B.A. PART-II **PAPER : BAP-201**

SEMESTER-III

COMPUTER SCIENCE **C PROGRAMMING AND DATA STRUCTURES**

UNIT - 2

Department of Distance Education
Punjabi University, Patiala

LESSON NOs :

Note:-The students can download syllabus from departmental website **www.dccpbi.com**

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.1 **AUTHOR : DR DHARAMVIR SHARMA**

INTRODUCTION TO DATA STRUCTURES

2.1.1 Objective of the lesson

In this lesson we will discuss the meaning, types, characteristics and operations on data structures.

2.1.2 Introduction

Software engineering is the study of ways in which to create large and complex computer applications and that generally involve many programmers and designers. At the heart of software engineering is with the overall design of the applications and on the creation of a design that is based on the needs and requirements of end users. While software engineering involves the full life cycle of a software project, is includes many different components - specification, requirements gathering, design, verification, coding, testing, quality assurance, user acceptance testing, production and ongoing maintenance.

Having an in-depth understanding on every component of software engineering is not mandatory; however, it is important to understand that the subject of data structures and algorithms is concerned with the coding phase. The use of data structures and algorithms is the nuts-and-blots used by programmers to store and manipulate data.

When thinking about a particular application or programming problem, many developers find themselves most interested in writing the algorithm to tackle the problem at hand, or adding cool features to the application to enhance the user's experience. Rarely, if ever, will you hear someone excited about what type of data structure they are using. However, the data structures used for a particular algorithm can greatly impact its performance. A common example is finding an element in a data structure. With an array, this process takes time proportional to the number of elements in the array. With binary search trees or SkipLists, the time required is sub-linear. When searching large amounts of data, the data structure chosen can make a difference in the application's performance that can be visibly measured in seconds or even minutes.

Since the data structure used by an algorithm can greatly affect the algorithm's performance, it is important that there exists a rigorous method by which to compare the efficiency of various data structures. What we, as developers utilizing a data structure, are primarily interested in is how the data structures performance changes as the amount of data stored increases. That is, for each new element stored by the data structure, how are the running times of the data structure's operations effected?

2.1.3 Definition of Data Structures & Algorithms

A data structure is an arrangement of data in a computer's memory or even disk storage. It is a specialized format for organizing and storing data. An example of several common data structures are arrays, linked lists, queues, stacks, binary

trees and hash tables. Algorithms, on the other hand are used to manipulate the data contained in these data structures as in searching and sorting.

The choice of the data structure often begins from the choice of an abstract data type. A well-designed data structure allows a variety of critical operations to be performed, using as few resources, both execution time and memory space, as possible. Data structures are implemented by a programming language as data types and the references and operations they provide.

Different kinds of data structures are suited to different kinds of applications, and some are highly specialized to certain tasks. For example, B-trees are particularly well-suited for implementation of databases, while networks of machines rely on routing tables to function.

In the design of many types of computer program, the choice of data structures is a primary design consideration. Experience in building large systems has shown that the difficulty of implementation and the quality and performance of the final result depends heavily on choosing the best data structure. After the data structures are chosen, the algorithms to be used often become relatively obvious. Sometimes things work in the opposite direction — data structures are chosen because certain key tasks have algorithms that work best with particular data structures. In either case, the choice of appropriate data structures is crucial.

Many algorithms apply directly to a specific data structures. When working with certain data structures you need to know how to insert new data, search for a specified item, and deleting a specific item.

2.1.4 Some other Definitions

2.1.4.1 Algorithm: A finite sequence of instructions, each of which has a clear meaning and can be executed with a finite amount of effort in finite time. Whatever the input values, an algorithm will definitely terminate after executing a finite number of instructions.

2.1.4.2 Data Type: Data type of a variable is the set of values that the variable may assume.

Basic data types in C:

- int, char, float and double

2.1.4.3 Four Fundamental Data Structures: The following four data structures are used ubiquitously in the description of algorithms and serve as basic building blocks for realizing more complex data structures.

- Sequences (also called as lists)
- Dictionaries
- Priority Queues
- Graphs

Dictionaries and priority queues can be classified under a broader category called *dynamic sets*. Also, binary and general trees are very popular building blocks for implementing dictionaries and priority queues.

2.1.4.4 Recursion: It is a method call in which the method being called is the same as the one making the call. The recursive algorithm is implemented by using a method that makes recursive calls to itself. It can be:

- Direct recursion: Recursion in which a method directly calls itself
- Indirect recursion: Recursion in which a chain of two or more method calls returns to the method that originated the chain

2.1.5 Types of Data Structures

Data structures can be broadly divided into three categories

2.1.5.1 Base data structures: Basic data structures are the building blocks for composite and complex data structures.

General type	Specific types
Primitive types	• Character • Integer • String • Double • Float
Composite types	• Struct • Bit field • Union

2.1.5.2 Linear Data Structures: Where data is stored in sequential or linear order. The most commonly used linear data structures are given below:

General type	Specific types

List (or vector or sequence)	• Array
	  ○ Bitmaps
	  ○ Dynamic array
	  ○ Sparse array
	  ○ Matrix
	  ▪ Sparse matrix
	• Linked list
	  ○ Doubly linked list
	• Buffer
	  ○ Stack
	  ○ Queue
	  ▪ Priority queue
	  ▪ Double-ended priority queue
	  ○ Deque
	  ○ Circular buffer

Associative array (a.k.a. dictionary or map)	• Hash table • Self-balancing binary search tree • Skip list

2.1.5.3 Non-linear Data Structures: Where data is not stored in sequential or linear order like trees and graphs etc.

General type	Specific types
Graph data structures	• Adjacency list • Adjacency matrix • Disjoint-set data structure

Tree data structures

- Graph-structured stack
- Scene graph
- M-Way Tree
  - B-tree
    - B+ tree
      - Generalized search tree
    - B* tree
    - K-ary tree
- Binary tree
  - Binary heap
  - Binary search trees (each tree node compares entire key values)
    - Self-balancing binary search trees
      - AVL tree
      - Red-black tree
      - Scapegoat tree
      - Splay tree
      - Interval tree
      - Treap
  - Exponential tree
- Heap
  - Binary heap
  - Binomial heap
  - Fibonacci heap
- Syntax tree
  - Abstract syntax tree

- Parse tree

- Decision theory
  - Binary decision diagram
  - Decision tree
    - Alternating decision tree
    - Minimax tree
      - Expectiminimax tree

2.1.6 Characteristics of Data Structures

Data Structure	Advantages	Disadvantages
Array	Quick inserts Fast access if index known	Slow search Slow deletes Fixed size
Ordered Array	Faster search than unsorted array	Slow inserts Slow deletes Fixed size
Stack	Last-in, first-out access	Slow access to other items
Queue	First-in, first-out access	Slow access to other items
Linked List	Quick inserts Quick deletes	Slow search
Binary Tree	Quick search Quick inserts Quick deletes	Deletion algorithm is complex

	(If the tree remains balanced)	
Red-Black Tree	Quick search Quick inserts Quick deletes (Tree always remains balanced)	Complex to implement
2-3-4 Tree	Quick search Quick inserts Quick deletes (Tree always remains balanced) (Similar trees good for disk storage)	Complex to implement
Hash Table	Very fast access if key is known Quick inserts	Slow deletes Access slow if key is not known Inefficient memory usage
Heap	Quick inserts Quick deletes Access to largest item	Slow access to other items
Graph	Best models real-world situations	Some algorithms are slow and very complex
NOTE: The data structures shown above (with the exception of the array) can be thought of as Abstract Data Types (ADTs).		

Commonly used algorithms include are useful for:

Searching for a particular data item (or record).

Sorting the data. There are many ways to sort data. Simple sorting, Advanced sorting

Iterating through all the items in a data structure. (Visiting each item in turn so as to display it or perform some other action on these items)

2.1.7 Operation on Data Structures

The data present in data structure is processed by means of certain operations. The choice of a particular data structures depends specifically on the frequency of specific operations. The following are some common operations performed on the data structures:

- **Insertion**: Adding new elements to the data structure
- **Deletion**: Removing an element from the data structure
- **Traversal**: Accessing each record only once so that certain items in the record may be processed. (This is also termed as visit the record)
- **Searching**: Finding the location of a record with a given key or finding the locations of all records satisfying one or more given conditions.

Some times two or more operations are used in a given situation, e.g. for deleting a record with a specific key requires first finding the location of the record and then deleting it.

The following operations are used in some specific situations only.

- **Sorting**: Arranging the elements in some logical order, e.g. alphabetically according to some name or in numerical order according to some number key, or chronologically according to dates of events etc.)
- **Merging**: Combining two sets of records into one.

2.1.8 Abstract Data Types

An Abstract Data Type (ADT) is more a way of looking at a data structure: focusing on what it does and ignoring how it does its job. A stack or a queue is an example of an ADT. It is important to understand that both stacks and queues can be implemented using an array. It is also possible to implement stacks and queues using a linked list. This demonstrates the "abstract" nature of stacks and queues: how they can be considered separately from their implementation.

To best describe the term Abstract Data Type, it is best to break the term down into "data type" and then "abstract".

2.1.8.1 data type

When we consider a primitive type we are actually referring to two things: a data item with certain characteristics and the permissible operations on that data. An int in Java, for example, can contain any whole-number value from -2,147,483,648 to +2,147,483,647. It can also be used with the operators +, -, *, and /. The data type's permissible operations are an inseparable part of its identity; understanding the type means understanding what operations can be performed on it.

In C++, any class represents a data type, in the sense that a class is made up of data (fields) and permissible operations on that data (methods). By extension, when a data storage structure like a stack or queue is represented by a class, it too can be referred to as a data type. A stack is different in many ways from an int but they are both defined as a certain arrangement of data and a set of operations on that data.

2.1.8.2 abstract

Now lets look at the "abstract" portion of the phrase. The word abstract in our context stands for "considered apart from the detailed specifications or implementation". In C++, an Abstract Data Type is a class considered without regard to its implementation. It can be thought of as a "description" of the data in the class and a list of operations that can be carried out on that data and instructions on how to use these operations. What is excluded though, is the details of how the methods carry out their tasks. An end user (or class user), you should be told what methods to call, how to call them and the results that should be expected, but not HOW they work.

We can further extend the meaning of the ADT when applying it to data structures such as a stack and queue. In C++, as with any class, it means the data and the operations that can be performed on it. In this context, although, even the fundamentals of how the data is stored should be invisible to the user. Users not only should not know how the methods work, they should also not know what structures are being used to store the data.

Consider for example the stack class. The end user knows that push() and pop() (amoung other similar methods) exist and how they work. The user doesn't and shouldn't have to know how push() and pop() work, or whether data is stored in an array, a linked list, or some other data structure like a tree.

2.1.8.3 The Interface

The ADT specification is often called an interface. It's what the user of the class actually sees. In C++, this would often be the public methods. Consider for example, the stack class - the public methods push() and pop() and similar methods from the interface would be published to the end user.

2.1.9 Summary

A data structure is an arrangement of data in a computer's memory or even disk storage. Algorithms, on the other hand, are used to manipulate the data contained in these data structures as in searching and sorting. The data structure used by an algorithm can greatly affect the algorithm's performance, therefore, it is important that there exists a rigorous method by which to compare the efficiency of various data structures.

Data structures can be basic (character, integer, float etc.), linear (array, stacks, queues, linked lists etc.) or non-linear (trees, graphs etc). The operations that are performed on data structures include insertion, deletion, searching, traversing, sorting and merging.

2.1.10 Questions

1. Define data structures.
2. What are the characteristics of data structures?
3. What are the operations performed on data structures?
4. Define Abstract Data Structure.

2.1.11 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.2
AUTHOR : DR DHARAMVIR SHARMA

ALGORITHM ANALYSIS

2.2.1 Objective of the lesson

In this lesson we will discuss the various operations applied on the data structures, time and space tradeoffs using different types of data structures and the representation of algorithm complexity in terns of big 'O' notation.

2.2.2 Introduction

In computer science, often the question is not how to solve a problem, but how to solve a problem well. For instance, take the problem of sorting. Many sorting algorithms are well-known; the problem is not to find a way to sort words but to find a way to efficiently sort words. This article is about understanding how to compare the relative efficiency of algorithms and why it's important to do so.

If it's possible to solve a problem by using a brute force technique, such as trying out all possible combinations of solutions (for instance, sorting a group of words by trying all possible orderings until you find one that is in order), then why is it necessary to find a better approach? The simplest answer is, if you had a fast enough computer, maybe it wouldn't be. But as it stands, we do not have access to computers fast enough. For instance, if you were to try out all possible orderings of 100 words, that would require 100! (100 factorial) orders of words. (Explanation) That's a number with a 158 digits; were you to compute 1,000,000,000 possibilities were second, you would still be left with the need for over 1x10^149 seconds, which is longer than the expected life of the universe. Clearly, having a more efficient algorithm to sort words would be handy and of course, there are many that can sort 100 words within the life of the universe.

Before going further, it's important to understand some of the terminology used for measuring algorithmic efficiency. Usually, the efficiency of an algorithm is expressed as how long it runs in relation to its input. For instance, in the above example, we showed how long it would take our naive sorting algorithm to sort a certain number of words. Usually we refer to the length of input as n; so, for the above example, the efficiency is roughly n!. You might have noticed that it's possible to come up with the correct order early on in the attempt -- for instance, if the words are already partially ordered, it's unlikely that the algorithm would have to try all n! combinations. Often we refer to the average efficiency, would be in this case n!/2. But because the division by two is nearly insignificant as n grows larger (half of 2 billion is, for instance, still a very large number), we usually ignore constant terms (unless the constant term is zero).

Now that we can describe any algorithm's efficiency in terms of its input length (assuming we know how to compute the efficiency), we can compare algorithms based on their "order". Here, "order" refers to the mathematical method used to compare the efficiency -- for instance, n^2 is "order of n squared," and n! is "order of n factorial." It should be obvious that an order of n^2 algorithm is much less efficient than an algorithm of order n. But not all orders are polynomial -- we've already seen n!, and some are order of log n, or order 2^n.

Often order is abbreviated with a capital O: for instance, O(n^2). This notation, known as big-O notation (explain later in this lesson), is a typical way of describing algorithmic efficiency; note that big-O notation typically does not call for inclusion of constants. Also, if you are determining the order of an algorithm and the order turns out to be the sum of several terms, you will typically express the efficiency as only the term with the highest order. For instance, if you have an algorithm with

efficiency
n^2 + n, then it is an algorithm of order O(n^2).

2.2.3 Asymptotic Notation

A problem may have numerous algorithmic solutions. In order to choose the best algorithm for a particular task, you need to be able to judge how long a particular solution will take to run. Or, more accurately, you need to be able to judge how long two solutions will take to run and choose the better of the two. You don't need to know how many minutes and seconds they will take but you do need some way to compare algorithms against one another.

Asymptotic complexity is a way of expressing the main component of the cost of an algorithm, using idealized units of computational work. Consider, for example, the algorithm for sorting a deck of cards, which proceeds by repeatedly searching through the deck for the lowest card. The asymptotic complexity of this algorithm is the square of the number of cards in the deck. This quadratic behavior is the main term in the complexity formula, it says, e.g., if you double the size of the deck, then the work is roughly quadrupled.

The exact formula for the cost is more complex and contains more details than are needed to understand the essential complexity of the algorithm. With our deck of cards, in the worst case, the deck would start out reverse-sorted, so our scans would have to go all the way to the end. The first scan would involve scanning 52 cards, the next would take 51, etc. So the cost formula is 52 + 51 + ... + 1. generally, letting N be the number of cards, the formula is 1 + 2 + ... + N, which equals (N+1) * (N/2) = (N^2 + N)/2 = (1/2)N^2 + N/2. But the N^2 term dominates the expression and this is what is key for comparing algorithm costs. (This is in fact an expensive algorithm; the best sorting algorithms run in sub-quadratic time.)

Asymptotically speaking, in the limit as N tends towards infinity, 1 + 2 + ... + N gets closer and closer to the pure quadratic function (1/2) N^2. And what difference does the constant factor of 1/2 make at this level of abstraction. So the behavior is said to be O(n2).

Now let us consider how we would go about comparing the complexity of two algorithms. Let f(n) be the cost, in the worst case, of one algorithm, expressed as a function of the input size n, and g(n) be the cost function for the other algorithm. E.g., for sorting algorithms, f(10) and g(10) would be the maximum number of steps that the algorithms would take on a list of 10 items. If, for all values of n >= 0, f(n) is less than or equal to g(n), then the algorithm with complexity function f is strictly faster. But, generally speaking, our concern for computational cost is for the cases

with large inputs; so the comparison of f(n) and g(n) for small values of n is less significant than the "long term" comparison of f(n) and g(n), for n larger than some threshold.

Note that we have been speaking about bounds on the performance of algorithms, rather than giving exact speeds. The actual number of steps required to sort our deck of cards (with our naive quadratic algorithm) will depend upon the order in which the cards begin. The actual time to perform each of our steps will depend upon our processor speed, the condition of our processor cache, etc., etc. It's all very complicated in the concrete details and moreover not relevant to the essence of the algorithm.

2.2.4 Big-O Notation

Big-O notation is used to express an ordering property among functions. In the context of our study of algorithms, the functions are the amount of resources consumed by an algorithm when the algorithm is executed. This function is usually denoted T(N). T(N) is the amount of the resource (usually time or the count of some specific operation) consumed when the input to the algorithm is of size N.

Sometimes it is not obvious what the ``size'' of the input is, so here are few examples:

- **Multiplication**: If we multiply two numbers together using the algorithm we learned in grade school, the number of single-digit multiplications and additions that we do depends on the number of digits in the two numbers being multiplied. (If we multiply two numbers together using a calculator, then the number of keystrokes we need to type also depends on the number of digits in the two numbers.) Therefore, for an algorithm that performs multiplication (or any other arithmetic operation) the ``size'' of the input is usually the number of digits in the input numbers.
- **Searching:** If we have a set of N unknown items, arranged in an unknown order and we want to do something involving all of them, then the size of the problem is N. For example, if we want to find the smallest element in an array of N numbers, then the size of the problem is N.

Big-O is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of the longest amount of time it could possibly take for the algorithm to complete.

More formally, for non-negative functions, f(n) and g(n), if there exists an integer n0 and a constant c > 0 such that for all integers n > n0, f(n) ≤ cg(n), then f(n) is Big O of g(n). This is denoted as "f(n) = O(g(n))". If graphed, g(n) serves as an upper bound to the curve you are analyzing, f(n).

Theory Examples

So, let's take an example of Big-O. Say that f(n) = 2n + 8, and g(n) = n2. Can we find a constant c, so that 2n + 8 <= n2? The number 4 works here, giving us 16 <= 16. For any number c greater than 4, this will still work. Since we're trying to generalize this for large values of n and small values (1, 2, 3) aren't that important, we can say that f(n) is generally faster than g(n), i.e., f(n) is bound by g(n) and will always be less than it. It could then be said that f(n) runs in O(n2) time: "f-of-n runs in Big-O of
n-squared time".

To find the upper bound - the Big-O time - assuming we know that f(n) is equal to (exactly) 2n + 8, we can take a few shortcuts. For example, we can remove all constants from the runtime; eventually, at some value of c, they become irrelevant. This makes f(n) = 2n. Also, for convenience of comparison, we remove constant multipliers; in this case, the 2. This makes f(n) = n. It could also be said that f(n) runs in O(n) time; that lets us put a tighter (closer) upper bound onto the estimate.

Practical Examples

O(n): printing a list of n items to the screen, looking at each item once. O(ln n): also "log n", taking a list of items, cutting it in half repeatedly until there's only one item left. O(n2): taking a list of n items and comparing every item to every other item.

Big-Omega Notation

For non-negative functions, f(n) and g(n), if there exists an integer n0 and a constant c > 0 such that for all integers n > n0, f(n) ≥ cg(n), then f(n) is omega of g(n). This is denoted as "f(n) = Ω(g(n))".

This is almost the same definition as Big Oh, except that "f(n) ≥ cg(n)", this makes g(n) a lower bound function, instead of an upper bound function. It describes the best that can happen for a given data size.

Theta Notation

For non-negative functions, f(n) and g(n), f(n) is theta of g(n) if and only if f(n) = O(g(n)) and f(n) = Ω(g(n)). This is denoted as "f(n) = Θ(g(n))".

This is basically saying that the function, f(n) is bounded both from the top and bottom by the same function, g(n).

Little-O Notation

For non-negative functions, f(n) and g(n), f(n) is little o of g(n) if and only if f(n) = O(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = o(g(n))". This represents a loose bounding version of Big O. g(n) bounds from the top but it does not bound the bottom.

Little Omega Notation

For non-negative functions, f(n) and g(n), f(n) is little omega of g(n) if and only if f(n) = Ω(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = ω(g(n))". Much like Little Oh, this is the equivalent for Big Omega. g(n) is a loose lower boundary of the function f(n); it bounds from the bottom, but not from the top.

How asymptotic notation relates to analyzing complexity

Temporal comparison is not the only issue in algorithms. There are space issues as well. Generally, a trade off between time and space is noticed in algorithms. Asymptotic notation empowers you to make that trade off. If you think of the amount of time and space your algorithm uses as a function of your data over time or space (time and space are usually analyzed separately), you can analyze how the time and space is handled when you introduce more data to your program.

This is important in data structures because you want a structure that behaves efficiently as you increase the amount of data it handles. Keep in mind though that algorithms that are efficient with large amounts of data are not always simple and efficient for small amounts of data. So if you know you are working with only a small amount of data and you have concerns for speed and code space, a trade off can be made for a function that does not behave well for large amounts of data.

A few examples of asymptotic notation

Generally, we use asymptotic notation as a convenient way to examine what can happen in a function in the worst case or in the best case. For example, if you want to write a function that searches through an array of numbers and returns the smallest one:

```
function find-min(array a[1..n])
  let j := ∞
  for i := 1 to n:
```

```
    j := min(j, a[i])
  repeat
  return j
end
```

Regardless of how big or small the array is, every time we run find-min, we have to initialize the i and j integer variables and return j at the end. Therefore, we can just think of those parts of the function as constant and ignore them.

So, how can we use asymptotic notation to discuss the find-min function? If we search through an array with 87 elements, then the for loop iterates 87 times, even if the very first element we hit turns out to be the minimum. Likewise, for n elements, the for loop iterates n times. Therefore, we say the function runs in time O(n).

What about this function:

```
function find-min-plus-max(array a[1..n])
 // First, find the smallest element in the array
 let j := ∞;
 for i := 1 to n:
   j := min(j, a[i])
 repeat
 let minim := j

 // Now, find the biggest element, add it to the smallest and
 j := −∞ ;
 for i := 1 to n:
   j := max(j, a[i])
 repeat
 let maxim := j

 // return the sum of the two
 return minim + maxim;
```

end

What's the running time for find-min-plus-max? There are two for loops, that each iterate n times, so the running time is clearly O(2n). Because 2 is a constant, we throw it away and write the running time as O(n). Why can you do this? If you recall the definition of Big-O notation, the function whose bound you're testing can be multiplied by some constant. If f(x) = 2x, we can see that if g(x) = x, then the Big-O condition holds. Thus O(2n) = O(n). This rule is general for the various asymptotic notations.

Finding the Big-O of a Function

Although the definitions given above completely define what the big-O of a function is, it is not always immediately obvious how to use them to actually discover the big-O of a function. Finding the order of many useful functions can be a challenging task, and there are even functions that have defied all attempts at a complete analysis! Luckily, since there has been so much work done in the area already, finding the big-O of many functions is simply a matter of finding a known result for a similar function and using it.

The following rules provide a jumping off point:

1. If T1(N) = O(f1(N)) and T2(N) = O(f2(N)) then:
2. T1(N) + T2(N) = max(O(f1(N)),O(f2(N)))
3. T1(N) X T2(N) = O(f1(N)Xf2(N))
4. If T(N) is a polynomial of degree k, then T(N) = Θ(Nk)

   The fact that T(N) = O(Nk) follows from the previous rule. Showing that T(N) = Θ(Nk) is slightly more complicated, but can be proven by showing that there is always a choice of c and n0 that satisfies the definition of Θ.
5. logk N = O(N), for any constant k.

   This is true, but of relatively limited use. Since O(N) is an upper bound, it is OK to say that anything with a lower order is O(N). This is handy as a last resort, when trying to find the big-O of a nasty function that includes logarithms, but whenever possible it is useful to draw a distinction between O(N) and O(logk N). For example, an algorithm that has a big-O of O(N log N) is (usually) much better than an algorithm that is O(N2).

Another method of finding the big-O of a function is to find the dominant term of the function, and find its order. The order of the dominant term will also be the order of the function.

The dominant term is the term that grows most quickly as n becomes large. Some rules of dominance include the following:

- Any exponential function of n dominates any polynomial function of n.
- Any polynomial function of n dominates any logarithmic function of n.
- Any logarithmic function of n dominates a constant term.
- A polynomial of degree k dominates a polynomial of degree l if and only if k>l.

There are additional rules that you can discover for yourself. The key is that term x(n) dominates function y(n) if and only if x(n)/y(n) grows as n grows large.

Using these rules can sometimes make finding the order of a complicated function easy. For example, the function f(n) = n4 + 2n + log n, is clumsy to manipulate algebraically. However, thanks to the rules of dominance, we know that we the 2n term will dominate the order of this function, so we can simply find the order of 2n, which is itself. Therefore, we can immediately say that f(n) = O(2n).

There are other methods that can be used find the order of functions. but we will not introduce them yet. Later in the semester, when we analyze several recursive algorithms, methods for analyzing the order of recursive functions will be introduced.

Properties of Orders

Since the orders of functions look like ordinary functions, it is tempting to treat them as such and manipulate them as though they were ordinary functions. This is usually a huge mistake.

For example, let f(n) = n2 and g(n) = 2n2. It should be clear that the both f(n) and g(n) are O(n2). What is the big-O of g(n) - f(n)? If we compute the order of the difference as the difference of the orders, then we would mistakenly conclude that (g(n) - f(n)) = the order of f(n) less the order of g(n), or (n2 – n2) = 0. This answer is obviously wrong. If we actually compute g(n) - f(n), however, we arrive at the correct conclusion:

(g(n) – f(n)) = (2n2 – n2) = n2 = O(n2).

2.2.5 Algorithm analysis

Now that we have seen the basics of big-O notation, it is time to relate this to the analysis of algorithms.

In our study of algorithms, nearly every function whose order we are interested in finding is a function that defines the quantity of some resource consumed by a

particular algorithm in relationship to the parameters of the algorithm. (This function is often referred to as a ***complexity*** of the algorithm or less frequently as the *cost function* of the algorithm.) This function is usually not the same as the algorithm itself but is a property of the algorithm. For example, when we are analyzing an algorithm that multiplies two numbers, the functions we might be interested in are the relationships between the number of digits in each number and the length of time or amount of memory required by the algorithm.

Although big-O notation is a way of describing the order of a function. It is also often meant to represent the time complexity of an algorithm. This is sloppy use of the mathematics, but unfortunately not uncommon. (For example, at the bottom of page 22 in Weiss, a factorial function is described as being O(N). Clearly, the factorial function itself is not O(N), it is O(N!) according to its definition. What is meant is that the particular algorithm given for computing the factorial of N requires O(N) time to execute.)

Note that this notation, like the orders themselves, doesn't tell us how quickly or slowly the algorithms actually execute for a given input. This information can be extremely important in practice- during the semester. We will study several algorithms that address the same problems and have the same order running time, but take substantially different amounts of time to execute.

Similarly, the order doesn't tell us how fast an algorithm will run for a small N. This may also be quite important in practice- during the semester, we will study at least one group of algorithms where the choice of the best algorithm depends on N- when N is small, one algorithm is best, but when N is large, a different algorithm is much better.

12.2.6 Time Space Tradeoffs

In computer science, a space-time or time-memory tradeoff is a situation where the memory use can be reduced at the cost of slower program execution or vice versa, the computation time can be reduced at the cost of increased memory use. As the relative costs of CPU cycles, RAM space and hard drive space change - hard drive space has for some time been getting cheaper at a much faster rate than other components of computers - the appropriate choices for space-time tradeoffs have changed radically. Often, by exploiting a space-time tradeoff, a program can be made to run much faster.

The most common situation is an algorithm involving a lookup table. An implementation can include the entire table, which reduces computing time but

increases the amount of memory needed or it can compute table entries as needed, increasing computing time but reducing memory requirements.

A space-time tradeoff can be applied to the problem of data storage. If data is stored uncompressed, it takes more space but less time than if the data were stored compressed (since compressing the data reduces the amount of space it takes but it takes time to run the compression algorithm). Depending on the particular instance of the problem, either way is practical. Another example is displaying mathematical formulae on primarily text-based websites. Storing only the LaTeX source and rendering it as an image every time the page is requested would be trading time for space - more time used but less space. Rendering the image when the page is changed and storing the rendered images would be trading space for time - more space used but less time. Note that there are also rare instances where it is possible to directly work with compressed data, such as in the case of compressed bitmap indices, where it is faster to work with compression than without compression.

Larger code size can be traded for higher program speed when applying loop unwinding. This technique makes the code longer for each iteration of a loop but saves the computation time required for jumping back to the beginning of the loop at the end of each iteration.

In the field of cryptography, where the adversary is trying to do better than the exponential time required for a brute force attack, the advantages of a space-time tradeoff can readily be seen. Rainbow tables use partially precomputed values in the hash space of a cryptographic hash function to crack passwords in minutes instead of weeks. Decreasing the size of the rainbow table increases the time required to iterate over the hash space. The meet-in-the-middle attack uses a space-time tradeoff to find the cryptographic key in only 2n + 1 encryptions (and O(2n) space) versus the expected 22n encryptions (but only O(1) space) of the naive attack.

Dynamic programming is another example where the time complexity of a problem can be reduced significantly by using more memory.

2.2.7 Algorithm efficiency

The ability to analyze a piece of code or an algorithm and understand its efficiency is vital for understanding computer science.

One approach to determining an algorithm's order is to start out assuming an order of O(1), an algorithm that doesn't do anything and immediately terminates no matter what the input. Then, find the section of code that you expect to have the highest order. From there, work out the algorithmic efficiency from the outside in --

figure out the efficiency of the outer loop or recursive portion of the code, then find the efficiency of the inner code; the total efficiency is the efficiency of each layer of code multiplied together. For instance, to compute the efficiency of a simple selection sort

```
for(int x=0; x<n; x++)
        {
          int min = x;
          for(int y=x; y<n; y++)
          {
                if(array[y]<array[min])
                  min=y;
          }
          int temp = array[x];
          array[x] = array[min];
          array[min] = temp;
        }
```

We compute that the order of the outer loop (for(int x = 0; ..)) is O(n); then, we compute that the order of the inner loop is roughly O(n). Note that even though its efficiency varies based on the value of x, the average efficiency is n/2 and we ignore the constant, so it's O(n). After multiplying together the order of the outer and the inner loop, we have O(n^2).

In order to use this approach effectively, you have to be able to deduce the order of the various steps of the algorithm. And you won't always have a piece of code to look at; sometimes you may want to just discuss a concept and determine its order. Some efficiencies are more important than others in computer science and on the next page, you'll see a list of the most important and useful orders of efficiency, along with examples of algorithms having that efficiency.

Algorithmic Efficiency -- Various Orders and Examples

Below is a list of some of the common orders and with example algorithms.

O(1)

An algorithm that runs the same no matter what the input. For instance, an

algorithm that always returns the same value regardless of input could be considered an algorithm of efficiency O(1).

O(logn)

Algorithms based on binary trees are often O(log n). This is because a perfectly balanced binary search tree has log n layers and to search for any element in a binary search tree requires traversing a single node on each layer.

The binary search algorithm is another example of a O(log n) algorithm. In a binary search, one is searching through an ordered array and, beginning in the middle of the remaining space to be searched, whether to search the top or the bottom half. You can divide the array in half only log n times before you reach a single element, which is the element being searched for, assuming it is in the array.

O(n)

Algorithms of efficiency n require only one pass over an entire input. For instance, a linear search algorithm, which searches an array by checking each element in turn is O(n). Often, accessing an element in a linked list is O(n) because linked lists do not support random access.

O(n log n)

Often, good sorting algorithms are roughly O(n log n). An example of an algorithm with this efficiency is merge sort, which breaks up an array into two halves, sorts those two halves by recursively calling itself on them and then merging the result back into a single array. Because it splits the array in half each time, the outer loop has an efficiency of logn and for each "level" of the array that has been split up (when the array is in two halves, then in quarters and so forth), it will have to merge together all of the elements, an operations that has order of n.

O(n^2)

A fairly reasonable efficiency, still in the polynomial time range, the typical examples for this order come from sorting algorithms, such as the selection sort example on the previous page.

O(2^n)

The most important non-polynomial efficiency is this exponential time increase. Many important problems can only be solved by algorithms with this (or worse) efficiency. One example is factoring large numbers expressed in binary; the only known way is by trial and error and a naive approach would involve dividing every number less than the number being factored into that number until one divided in evenly. For every increase of a single digit, it would require twice as many tests.

2.2.8 Summary

Efficiency of algorithm is of prime concern in computer science. The complexity of an algorithm can measured in terms of number of operations that are required to be performed for solving the problem called time complexity or the space required by the algorithm called space complexity. The notation used for representing complexity is Big-O. The time and space trade off of an algorithm maintains a balance in the time and space complexity of the algorithm.

2.2.9 Questions

1. What is the need of writing efficient algorithms?
2. Define asymptotic notation.
3. What do you mean by Big-O notation? How is it computed?
4. What are the various notations used for representing algorithm complexity?
5. What do you mean by algorithm analysis?
6. Define time and space trade off.
7. Define algorithm efficiency. How efficiency of an algorithm is measured?

2.2.10 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.3 **AUTHOR : DR DHARAMVIR SHARMA**

ARRAYS

2.3.1 Objective of the lesson We will explore arrays, multidimensional arrays, sparse arrays and matrices in this lesson.

2.3.2 Introduction

An Array is a data structure that contains a group of elements. Array is a collection of same type elements under the same variable identifier referenced by index number. An array is a way to reference a series of memory locations using the same name. Each memory location is represented by an array element. An array

element is similar to one variable except it is identified by an index value instead of a name. An index value is a number used to identify an array element.

Now we'll show you what an array looks like, with the three array elements shown next. The array is called grades. The first array element is called grades[0]. The zero is the index value. The square bracket tells the computer that the value inside the square bracket is an index.

```
grades[0]

grades[1]

grades[2]
```

Each array element is like a variable name. For example, the following variables are equivalent to array elements. There is no difference between array elements and variables—well, almost no difference but we'll get to the differences in a moment. For now, let's explore how they are the same. Here are three integer variables:

```
int maryGrade;

int bobGrade;

int amberGrade;
```

You probably recall from your programming class that you store a value into a memory location by using an assignment statement. Here are two assignment statements. The first assigns a value to a variable and the other assigns a value to an array element. Notice that these statements are practically the same except reference is made to the index of the array element in the second statement:

```
int grades[1];

maryGrade = 90;

grades[0] = 90;
```

Suppose you want to use the value stored in a memory location. There are a number of ways to do this in a program but a common way is to use another assignment statement like the ones shown in the next example. The first assignment statement uses two variables, the next assignment statement uses two array elements and the last assignment statement assigns the value referenced by a variable name and assigns that value to an array element:

```
bobGrade = maryGrade;

grades[0] = grades[1];

grades[0] = bobGrade;
```

You've probably noticed a pattern developing. You use an array element the same way you use a variable.

There are two important differences between an array element and a variable, and those differences make working with large amounts of data a breeze. Suppose you had to work with 100 grades to calculate the average grade. How would you do this?

The challenge isn't applying the formula for calculating an average. You know how that's done. The challenge is to come up with 100 variable names and then reference all those variable names in a program. Ouch!

First, you'd need to sum all the grades by writing a statement similar to the following. (We'll stop at three variables because it's difficult to identify 100 variables—and we'd run out of space on this page.)

```
sum = maryGrade + bobGrade + amberGrade;
```

Now, here's how a smart programmer meets this challenge using an array:

```
sum = 0;
for (int i = 0; i < 100; i++)
   sum = sum + grades[i];
```

Big difference. The control variable of the for loop is the index for the array element, enabling the program to quickly walk through all array elements in two lines of code. (The first statement has nothing to do with walking through all the array elements. It only initializes the sum variable with the total grades.)

The other difference between an array and a variable is that all the array elements are next to each other in memory. Variables can be anywhere in memory. For example, grades[0] is next to grades[1] in memory, grades[1] is next to grades[2] in memory and so on. In contrast, maryGrade and bobGrade variables can be anywhere in memory, even if they are declared in the same declaration statement.

The location of array elements is important when pointers are used to manipulate data stored in memory. It is more efficient to point to array elements than variables because the computer moves to the next memory location when you point to the next array element.

Some programmers might say that arrays are the backbone of data structures because an array enables a programmer to easily reorganize hundreds of values stored in memory by using an array of pointers to pointers.

So we drew a picture to show you the importance of arrays in data structures. Figure 2.3.1 shows memory. Each block is a byte. We'll say that two bytes are needed to store a memory address in memory. You need to store a memory address in memory because you'll use it to refer to other memory addresses in the "An Array of Pointers" section of this lesson.

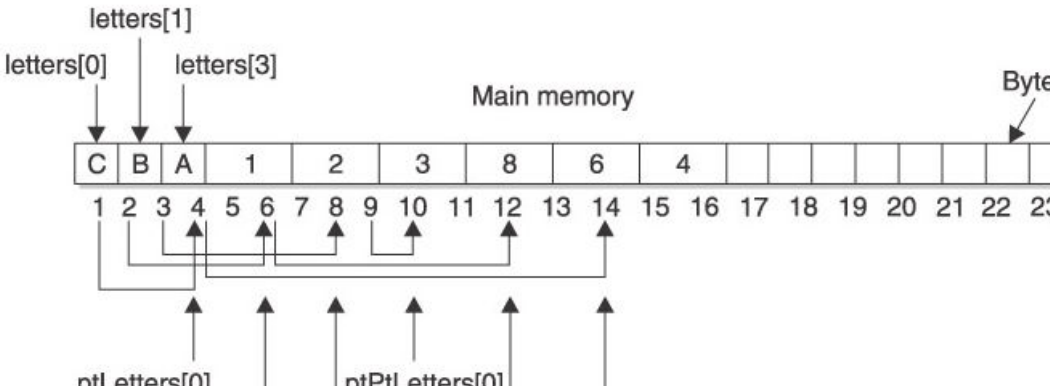


Figure 2.3.1: Elements of an array are stored sequentially in memory.

First, create an array called letters and assign characters to it, as shown here:

```
char letters[3];
letters[0] = 'C';
letters[1] = 'B';
letters[2] = 'A';
```

You'll notice in Figure 2.3.1 that each letter appears one after the other in memory. This is because these values are assigned to elements of an array, and each array element is placed sequentially in memory.

Next, create an array of pointers. A pointer is a variable that contains a memory address of another variable. In this example, you'll use an array of pointers instead of a pointer variable.

An array of pointers is nearly identical to a pointer variable except each array element contains a memory address. Assign the memory address of each element of the letters array to elements of the ptLetters array, which is an array of pointers. Here's how this is done in C and C++:

```
char * ptLetters[3];
for (int i = 0; i < 3; i++)
```

```
ptLetters[i] = &letters[i];
```

The ampersand (&), which is called the address operator, tells the computer to assign the memory address of the element of the letters array and not the contents of the element.

The final step is to create an array of pointers to pointers and then use it to change the order of the letters array when printing the letters array on the screen. A pointer to a pointer is a variable that contains the address of another pointer. In the example, you use an array of pointers to a pointer where each element of the array is like a pointer to a pointer variable. That is, each element is assigned an address of a pointer.

Use the following code to assign the memory address of each element of the ptLetters pointer array to the ptPtLetters pointer to pointer array. Notice that you don't use a for loop. This is because you need to change the order of the letters array without changing the letters array itself. Figure 2.3.1 shows how memory looks after the following code executes. If you printed elements of the ptPtLetters array, what would be displayed on the screen?

```
char ** ptPTLetters[3];

ptPtLetters[0] = &ptLetters[2];

ptPtLetters[1] = &ptLetters[1];

ptPtLetters[2] = &ptLetters[0];
```

Here is the code that prints the ptPtLetters array:

```
for ( i = 0; i <3; i++)

  cout << **ptPtLetters[i] << endl;
```

The answer to the question is A B C. Follow Figure 2.3.1 as we explain how this works. The first element of the ptPtLetters array is located at memory address 10. The content of memory address 10 is 8, which is memory address 8 because memory address 10 is the last element of the array ptLetters—a pointer. The value of memory address 8 is 3, which is the memory address of the third element of the array letters.

When the computer sees the ptPtLetters[i] statement for the first time, it goes to the array element ptPtLetters[0] and reads its value, which is 8. The computer then goes to memory address 8 and reads its content because memory address 8 is a pointer. The content of memory address 8 is 3, which is the memory address of the

third element of the letters array. The computer reads the content of memory address 3 and displays the content on the screen.

This can be a bit tricky to follow unless you use Figure 2.3.1 as a guide; you can also use Figure 2.3.1 to explain how the computer displays the other letters.

The importance of using arrays for data structures is that you can easily change the order of data by using pointers and pointers to pointers without having to touch the original data. Some smart programmer might tell you that you're not saving any time or memory by using pointers and pointers to pointers to rearrange an array of characters. The programmer is correct. However, we're juggling characters to illustrate how arrays and pointers to pointers work. In the real world, pointers typically point to a whole group of information such as a client's name, address, phone number and other pertinent data. Instead of juggling all that information, you need only to juggle memory addresses.

2.3.3 One Dimensional Arrays

The way to declare an array depends on the programming language used to write your program. In Java, there are two techniques for declaring an array. You can declare and initialize an array either where memory is allocated at compile time or where memory is dynamically allocated at runtime. Allocation is another way of saying reserving memory.

Let's begin by declaring an array where memory is reserved when you compile your program. This technique is similar in Java, C and C++, except in Java you must initialize the array when the array is declared. There are four components of a statement that declares an array. These components are a data type, an array name, the total number of array element to create and a semicolon (;). The semicolon tells the computer that the preceding is a statement. Here's the declaration statement in C and C++:

```
int grades[10];
```

In Java, you must initialize the array when the array is declared as shown here. The size of the array is automatically determined by counting the number of values within the braces. Therefore, there is n't any need to place the size of the array within the square brackets:

```
int[] grades = {0,0,0,0,0,0,0,0,0,0};
```

The data type is a keyword that tells the computer the amount of memory to reserve for each element of the array. In this example, the computer is told to reserve enough memory to store an integer for each array element.

The array name is the name you use within a program to reference an array element. The array name in this example is grades. The number within the square brackets is the total number of elements that will be in the array. The previous statements tell the computer to create an array of 10 elements. Previously in this chapter you learned that the index for the first array element is zero, not one. Therefore, the tenth array element has the index value 9, not 10.

Some programs confuse an index with the total number of array elements. That is, they use the value 9 within the square brackets when declaring an array because they assume they are declaring 10 elements. In reality, they are declaring an array of 9 elements. The confusion stems from the fact that 9 is the index to reference the tenth array element.

With a little practice you can avoid making this mistake. Remember that the value within the square brackets in the statement that creates an array is not an index, although it resembles an index. This value is the number of array elements you need. That is, you insert the number 10 within the square brackets if you need 10 array elements. You use the index value of 9 if you want to access the tenth element.

In order to allocate memory at compile time, you must know the number of array elements that you need. Sometimes you don't know this, especially if your program loads the array with data stored in a database. The amount of data stored in a database typically fluctuates.

2.3.4 Multi-dimensional Arrays

The array described in this chapter is referred to as a one-dimensional array because the array consists of one series of elements. However, an array can have more than one series of elements. This is called a multidimensional array.

A multidimensional array consists of two or more arrays defined by sets of array elements, as shown in Figure 2.3.2. Each set of array elements is an array. The first set of array elements is considered the primary array and the second and subsequent sets of array elements are considered subarrays.

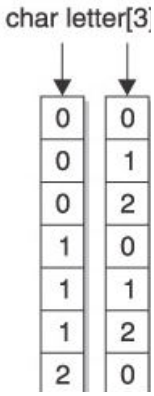


Figure 2.3.2: A two-dimensional array is a multidimensional array consisting of two arrays.

There are two arrays in the multidimensional array shown in Figure 2.3.2. Each element of the first array points to a corresponding array. For example, letters[1] in Figure 2.3.2 points to the array beginning with array element letters[1][0] where the zero is the first element of the second array.

Although you can create an array with any size multidimension, many programmers limit an array to two dimensions. Any size greater than two dimensions become unwieldy to manage.

An analogy we find helpful is visualizing a table (rows and columns) for a two-dimensional array and a cube (or similar figure) for a three-dimensional array.

2.3.4.1 Need of a Multidimensional Array

A multidimensional array can be useful to organize subgroups of data within an array. Let's say that a student has three grades, a mid-term grade, a final exam grade, and a final grade. You can store all three grades for an endless number of students in a two-dimensional array, as shown in Figure 2.3.3.

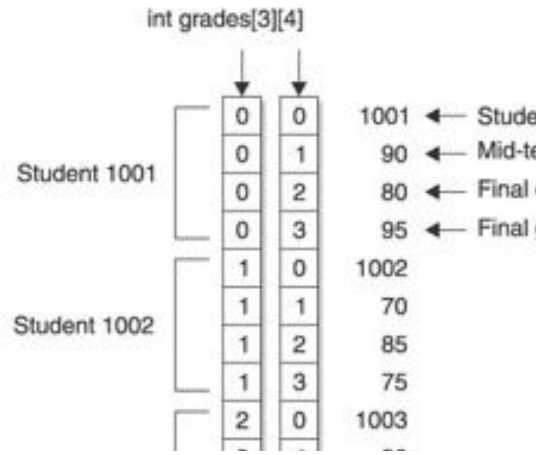


Figure 2.3.3: All three grades can be stored in a multidimensional array.

Figure 2.3.3 declares a multidimensional array of integers. The first set of array elements contains three array elements, one for each student. The second set of array elements has four array elements. The first of the four elements contains the student ID and the other three contain the three grades for that student ID.

Data stored in a multidimensional array is stored sequentially by sets of elements, as shown in Figure 2.3.4. The first set of four array elements is placed in memory, followed by the second set of four array elements, and so on.

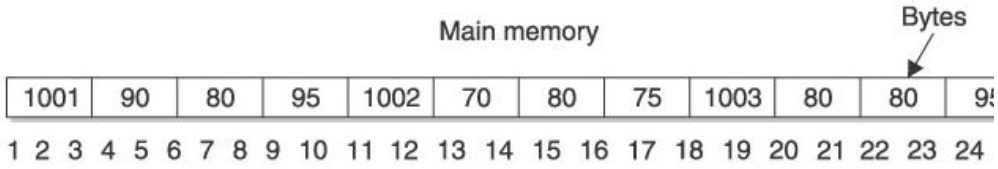


Figure 2.3.4: Elements of a multidimensional array are stored sequentially in memory.

The name of a multidimensional array references the memory address of the first element of the first set of four elements. That is, grades is the equivalent of using memory address 1 in Figure 2.3.4. You can use the name of a multidimensional array as a pointer to the entire array.

The index of the first element of the first set of array elements points to the memory address where values assigned to array elements are stored.

Referencing the index of the first dimension points to the memory address of the first element of that dimension. For example, referencing grades[1] points to

memory address 9 in Figure 2.3.4. Memory address 9 is the first memory address of contiguous memory where values of the second set of array elements that are associated with grades[1] are stored.

A multidimensional array is declared similar to the way you declare a one-dimensional array except you specify the number of elements in both dimensions. For example, the multidimensional array shown in Figure 2.3.3 is declared as follows in C or C++:

```
int grades[3][4];
```

The first bracket ([3]) tells the compiler that you're declaring 3 pointers, each pointing to an array. This concept might be confusing because the term "pointer" may make some programmers think of pointer variable or pointer array, which you'll learn about later in this lesson. However, we are not talking about a pointer variable or pointer array. Instead, we are saying that each element of the first dimension of a multidimensional array reference a corresponding second dimension, which is an array.

In this example, all the arrays pointed to by the first index are of the same size. The second index can be of variable size. For example, the previous statement declares a two-dimensional array where there are 3 elements in the first dimension and 4 elements in the second dimension.

The element grades[0] is said to "point" (just as you use your finger to point) to the second dimension of the array, which is referenced as grades[0][0]. The second dimension is considered an array. Therefore, programmers say that the first element of a multidimensional array points to another array (that is, the second dimension).

The data type tells the computer that each element of the array will contain an integer data type. The data type is followed by the array name and two values that indicate the size of each dimension used for the array. In this case, there are three sets of four array elements.

You declare a multidimensional array and initialize its elements by using French braces, as shown in Figure 2.3.5. There are three sets of inner French braces. Each of these sets represents the first dimension of the array. There are four values within each set of inner French braces. These values are assigned to each element of the second dimension of the array.

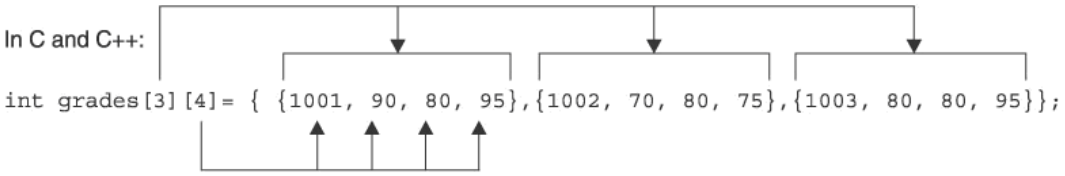


Figure 2.3.5: Braces define sets of values to be assigned to array elements.

You assign a value to an element of a multidimensional array with an assignment statement similar to the assignment statement that assigns a value to a single-dimensional array, as shown here:

```
grades[0][0] = 1001;
```

You must specify the index for both dimensions. In this example, the integer 1001, which is a student ID, is assigned to the first element of the first set of elements in the grades array.

2.3.4.2 Referencing Contents of a Multidimensional Array

The contents of elements of a multidimensional array can be used in a program by referencing the index of both dimensions of the array element. Figure 2.3.6 shows you how to display the final exam grade for the second student.

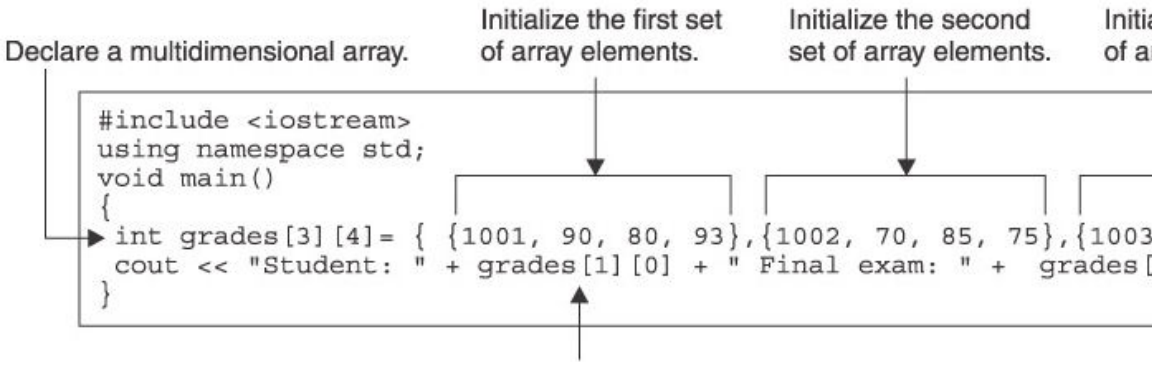


Figure 2.3.6: Display the contents of array elements by referencing the index of both sets of array elements.

In this example, the student ID is displayed by referencing the first element of the second set, and the grade for the final exam is displayed by referencing the third element of the second set.

Index of the first element

The index of the first element (sometimes called the "origin") varies by language. There are three main implementations: zero-based, one-based and n-based arrays, for which the first element has an index of zero, one or a programmer-specified value. The zero-based array is more natural in the root machine language and was popularized by the C programming language, where the abstraction of array is very weak and an index n of a one-dimensional array is simply the offset of the element accessed from the address of the first (or "zeroth") element (scaled by the size of the element). One-based arrays are based on traditional mathematics notation for matrices and most but not all, mathematical sequences. n-based is made available so the programmer is free to choose the lower bound, which may even be negative, which is most naturally suited for the problem at hand.

Index of the last element

The relation between numbers appearing in an array declaration and the index of that array's last element also varies by language. In some languages (e.g. C) the number of elements contained in the arrays must be specified, whereas in others (e.g. Visual Basic .NET) the numeric value of the index of the last element must be specified.

2.3.5 Operations on Arrays

An array is the simplest of all the data structures to implement and use. The various operation that can be applied on array are:

i. **Insertion of an element:** An element can be inserted in the array by specifying the location where it is to be stored. The location should not be greater than the size of the array. e.g. A[i] = 10 this will insert value 10 at the i^{th} location in the array and value of i in this case should not be greater than size of the array.

ii. **Deletion of an element:** Since all element of an array are store in sequential order in the memory. Deletion of an element can be performed by sifting the element appearing after the element to be deleted from the array.

iii. **Searching:** Searching on array can be performed in linear order if the array elements are not arranged in ascending or descending order. If elements are arranged then binary search can be applied which is far superior to linear search. Complexity of linear search in worst case is O(n) as compared to that of binary search which is $\log_2 n$. Consider search for a number, which is not in the array, in an array of 10000 numbers. Linear search needs 10000

comparisons to declare that the number is not present. Compare this to the comparisons needed by binary search, which will need about 15 comparisons to make the same declaration.

vi. **Sorting:** It means arranging the elements of the array in ascending or descending order. There are various sorting algorithms for performing this task. These algorithms have been discussed in detail in lessons 2.5 and 18.

2.3.6 Sparse Arrays and Matrices

A sparse array is an array in which most of the elements have the same value (known as the default value -- usually 0 or null).

A naive implementation of an array may allocate space for the entire array but in the case where there are few non-default values, this implementation is inefficient. Typically the algorithm used instead of an ordinary array is determined by other known features (or statistical features) of the array, for instance if the sparsity is known in advance, or if the elements are arranged according to some function (e.g. occur in blocks).

As an example, a spreadsheet containing 100x100 mostly empty cells might be more efficiently stored as a linked list rather than an array containing ten thousand array elements.

2.3.6.1 Representation of Sparse matrix

The naive data structure for a matrix is a two-dimensional array. Each entry in the array represents an element ai, j of the matrix and can be accessed by the two indices i and j. For an m×n matrix we need at least enough memory to store (m×n) entries to represent the matrix.

Many if not most entries of a sparse matrix are zeros. The basic idea when storing sparse matrices is to store only the non-zero entries as opposed to storing all entries. Depending on the number and distribution of the non-zero entries, different data structures can be used and yield huge savings in memory when compared to a naïve approach.

A simple method of representing sparse matrix is a two dimensional array of NNZ+1*3, where NNZ is the number of non-zero elements and number of columns is fixed to 3.

The first row of the matrix contains number of rows, number of columns and number of non-zero elements in the source matrix. The remaining NNZ rows store the row, column and value of the non-zero element.

For example, consider the source matrix

[1 2 0 0]

[0 3 9 0]

[0 1 4 0]

The simple sparse representation of this matrix is

[3 4 6]

[1 1 1]

[1 2 2]

[2 2 3]

[2 3 9]

[3 2 1]

[3 3 4]

Another example of such a sparse matrix format is the Yale Sparse Matrix Format. It stores an initial sparse m×n matrix, M, in row form using three one-dimensional arrays. Let NNZ denote the number of nonzero entries of M. The first array is A, which is of length NNZ and holds all nonzero entries of M in left-to-right top-to-bottom order. The second array is IA, which is of length m + 1 (i.e., one entry per row, plus one). IA(i) contains the index in A of the first nonzero element of row i. Row i of the original matrix extends from A(IA(i)) to A(IA(i+1)-1). The third array, JA, contains the column index of each element of A, so it also is of length NNZ.

Consider the source matrix given above, the Yale sparse matrix format is

A = [1 2 3 9 1 4]

IA = [1 3 5 7]

JA = [1 2 2 3 2 3]

2.3.7 Row Major Ordering

Row major ordering assigns successive elements, moving across the rows and then down the columns, to successive memory locations. This mapping is emonstrated in Figure 2.3.7 :

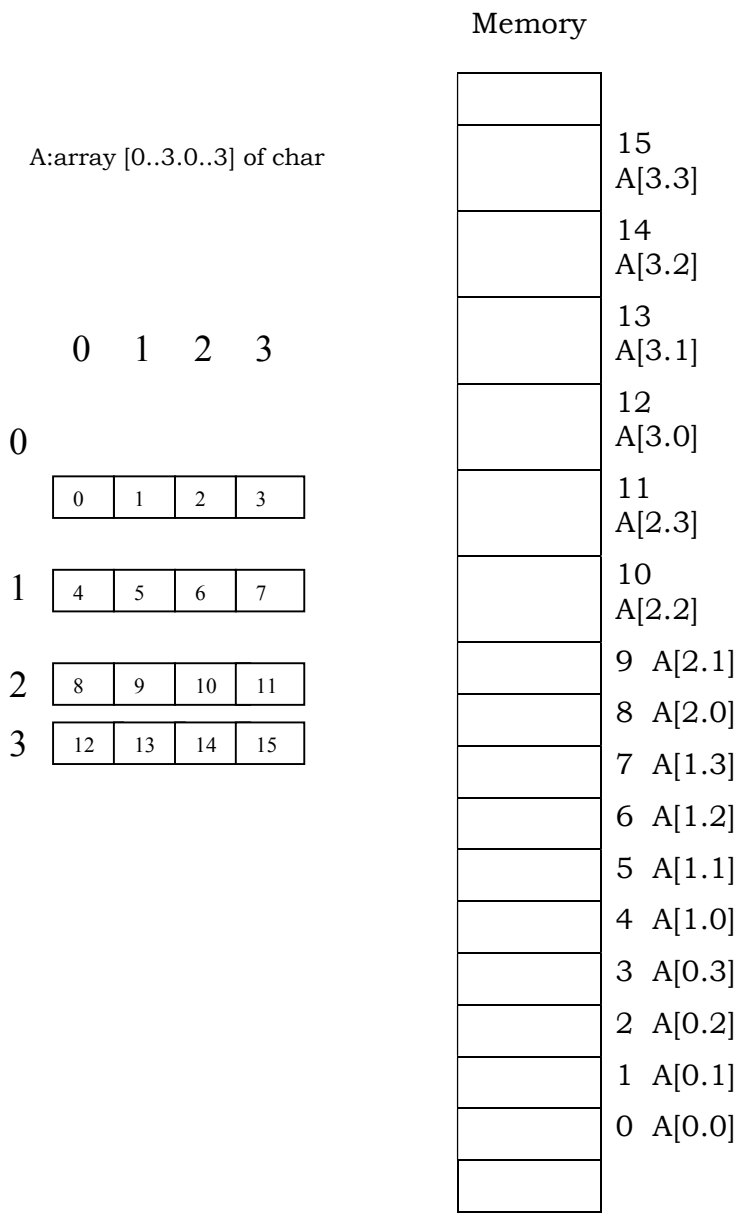


Figure 2.3.7 Row Major Array Element Ordering

Row major ordering is the method employed by most high level programming languages including Pascal, C/C++, Java, Ada, Modula-2, etc. It is very easy to implement and easy to use in machine language. The conversion from a two-dimensional structure to a linear array is very intuitive. Start with the first row (row number zero) and then concatenate the second row to its end. Then concatenate the third row to the end of the list, then the fourth row, etc. (see Figure 15.8)

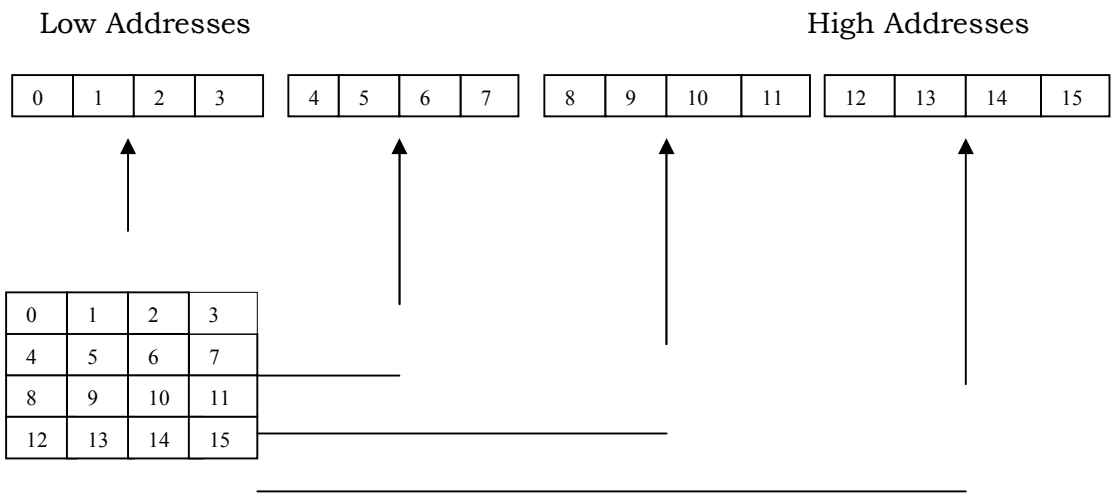


Figure 2.3.8 Another View of Row-Major Ordering for a 4x4 Array

For those who like to think in terms of program code, the following nested Pascal loop also demonstrates how row major ordering works:

```
index := 0;
 for colindex := 0 to 3 do
        for rowindex := 0 to 3 do
        begin
                memory [index] := rowmajor [colindex][rowindex];
                index := index + 1;
        end;
```

The important thing to note from this code, that applies regardless of the number of dimensions, is that the rightmost index increases the fastest. That is, as you allocate successive memory locations you increment the rightmost index until you reach the end of the current row. Upon reaching the end, reset the index back to the beginning of the row and increment the next successive index by one (that is, move down to the next row.). This works equally well for any number of dimensions. The following Pascal segment demonstrates row major organization for a 4x4x4 array:

```
index := 0;
 for depthindex := 0 to 3 do
```

```
for colindex := 0 to 3 do
  for rowindex := 0 to 3 do begin
        memory [index] := rowmajor [depthindex][colindex][rowindex];
        index := index + 1;
  end;
```

The actual function that converts a list of index values into an offset doesn't involve loops or much in the way of fancy computations. Indeed, it's a slight modification of the formula for computing the address of an element of a single dimension array. The formula to compute the offset for a two-dimension row major ordered array declared in Pascal as "A:array [0..3,0..3] of integer" is

Element_Address = Base_Address + (colindex * row_size + rowindex) * Element_Size

As usual, *Base_Address* is the address of the first element of the array (*A[0][0]* in this case) and *Element_Size* is the size of an individual element of the array, in bytes. *Colindex* is the leftmost index, *rowindex* is the rightmost index into the array. *Row_size* is the number of elements in one row of the array (four, in this case, since each row has four elements). Assuming *Element_Size* is one, this formula computes the following offsets from the base address:

Column index	**Row index**	**Offset into Array**
0	0	0
0	1	1
0	2	2
0	3	3
1	0	4
1	1	5
1	2	6
1	3	7
2	0	8
2	1	9

2	2	10
2	3	11
3	0	12
3	1	13
3	2	14
3	3	15

For a three-dimensional array, the formula to compute the offset into memory is the following:

Address = Base + ((depthindex*col_size+colindex) * row_size + rowindex) * Element_Size

Col_size is the number of items in a column, *row_size* is the number of items in a row. In C/C++, if you've declared the array as "*type* A[i] [j] [k];" then *row_size* is equal to *k* and *col_size* is equal to *j*.

For a four dimensional array, declared in C/C++ as "type A[i] [j] [k] [m];" the formula for computing the address of an array element is

Address =

Base + (((LeftIndex * depth_size + depthindex)*col_size+colindex) * row_size +

rowindex) * Element_Size

Depth_size is equal to *j*, *col_size* is equal to *k*, and *row_size* is equal to *m*. *LeftIndex* represents the value of the leftmost index.

By now you're probably beginning to see a pattern. There is a generic formula that will compute the offset into memory for an array with *any* number of dimensions, however, you'll rarely use more than four. Another convenient way to think of row major arrays is as arrays of arrays. Consider the following single dimension Pascal array definition:

A: array [0..3] of *sometype*;

Assume that *sometype* is the type "sometype = array [0..3] of char;".

A is a single dimension array. Its individual elements happen to be arrays, but you can safely ignore that for the time being. The formula to compute the address of an element of a single dimension array is

Element_Address = Base + Index * Element_Size

In this case *Element_Size* happens to be four since each element of *A* is an array of four characters. So what does this formula compute? It computes the base address of each row in this 4x4 array of characters (see Figure 15.9):

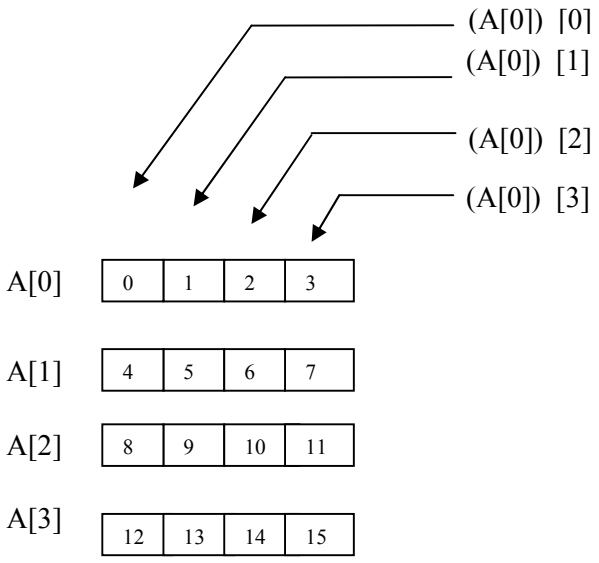


Figure 12.3.9 Viewing a 4x4 Array as an Array of Arrays

Of course, once you compute the base address of a row, you can reapply the single dimension formula to get the address of a particular element. While this doesn't affect the computation at all, conceptually it's probably a little easier to deal with several single dimension computations rather than a complex multidimensional array element address computation.

Consider a Pascal array defined as "A:array [0..3] [0..3] [0..3] [0..3] [0..3] of char;" You can view this five-dimension array as a single dimension array of arrays. The following Pascal code demonstrates such a definition:

```
type
        OneD = array [0..3] of char;
        TwoD = array [0..3] of OneD;
        ThreeD = array [0..3] of TwoD;
        FourD = array [0..3] of ThreeD;
var
```

```
A : array [0..3] of FourD;
```

The size of *OneD* is four bytes. Since *TwoD* contains four *OneD* arrays, its size is 16 bytes. Likewise, *ThreeD* is four *TwoDs*, so it is 64 bytes long. Finally, *FourD* is four *ThreeDs*, so it is 256 bytes long. To compute the address of "A [b, c, d, e, f]" you could use the following steps:

- Compute the address of *A [b]* as "Base + *b* * size". Here size is 256 bytes. Use this result as the new base address in the next computation.
- Compute the address of *A [b, c]* by the formula "Base + *c**size", where *Base* is the value obtained immediately above and size is 64. Use the result as the new base in the next computation.
- Compute the address of *A [b, c, d]* by "Base + *d**size" with *Base* coming from the above computation and size being 16.
- Compute the address of *A [b, c, d, e]* with the formula "Base + *e**size" with Base from above and size being four. Use this value as the base for the next computation.
- Finally, compute the address of *A [b, c, d, e, f]* using the formula "Base + *f**size" where base comes from the above computation and size is one (obviously you can simply ignore this final multiplication). The result you obtain at this point is the address of the desired element.

Not only is this scheme easier to deal with than the fancy formulae given earlier, but it is easier to compute (using a single loop) as well. Suppose you have two arrays initialized as follows

A1 = [256, 64, 16, 4, 1] and A2 = [b, c, d, e, f]

then the Pascal code to perform the element address computation becomes:

```
for i := 0 to 4 do
        base := base + A1[i] * A2[i];
```

Presumably *base* contains the base address of the array before executing this loop. Note that you can easily extend this code to any number of dimensions by simply initializing *A1* and *A2* appropriately and changing the ending value of the for loop.

As it turns out, the computational overhead for a loop like this is too great to consider in practice. You would only use an algorithm like this if you needed to be able to specify the number of dimensions at run time. Indeed, one of the main reasons you won't find higher dimension arrays in assembly language is that assembly language displays the inefficiencies associated with such access. It's easy

to enter something like "*A [b,c,d,e,f]*" into a Pascal program, not realizing what the compiler is doing with the code. Assembly language programmers are not so cavalier - they see the mess you wind up with when you use higher dimension arrays. Indeed, good assembly language programmers try to avoid two dimension arrays and often resort to tricks in order to access data in such an array when its use becomes absolutely mandatory. But more on that a little later.

2.3.8 Column Major Ordering

Column major ordering is the other function frequently used to compute the address of an array element. FORTRAN and various dialects of BASIC (e.g., older versions of Microsoft BASIC) use this method to index arrays.

In row major ordering the rightmost index increased the fastest as you moved through consecutive memory locations. In column major ordering the leftmost index increases the fastest. Pictorially, a column major ordered array is organized as shown in Figure 2.3.8:

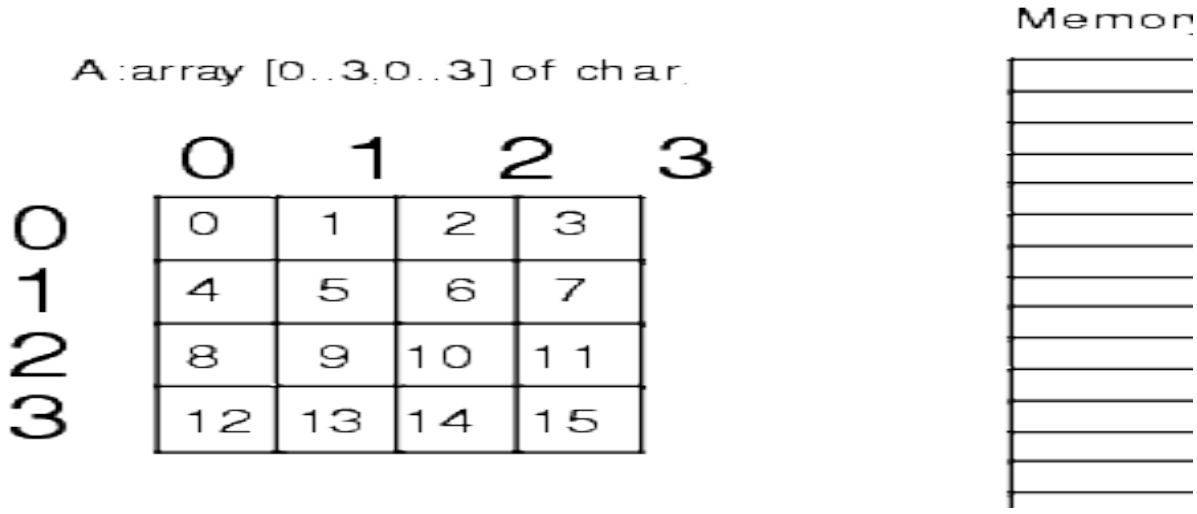


Figure 2.3.8 Column Major Array Element Ordering

The formulae for computing the address of an array element when using column major ordering is very similar to that for row major ordering. You simply reverse the indexes and sizes in the computation:

For a two-dimension column major array:

Element_Address = Base_Address + (rowindex * col_size + colindex) * Element_Size

For a three-dimension column major array:

Address = Base + ((rowindex*col_size+colindex) * depth_size + depthindex) * Element_Size

For a four-dimension column major array:

Address =

Base + (((rowindex * col_size + colindex)*depth_size + depthindex) *

Left_size + Leftindex) * Element_Size

The single Pascal loop provided for row major access remains unchanged (to access A[b][c][d][e][f]):

```
for i := 0 to 4 do
        base := base + A1[i] * A2[i];
```

Likewise, the initial values of the *A1* array remain unchanged:

A1 = {256, 64, 16, 4, 1}

The only thing that needs to change is the initial values for the *A2* array, and all you have to do here is reverse the order of the indices:

A2 = { f, e, d, c, b}

2.3.7 Summary

Array is a linear data structure. Elements of an array are stored sequentially in memory. Each element of an array can be addressed by an index. Therefore, accessing an element of array does not require any traversal through the other element of it. All languages provide arrays as inbuilt data structures.

Arrays can be one dimensional or multi-dimensional. Two dimensional array is called matrix, which has wide application in mathematics as well as computer science.

A matrix with very few non-zero elements is called a sparse matrix and there are different methods of storing the sparse matrix other than the normal two dimensional matrix.

2.3.8 Questions

1. What is the difference between an array element and a variable?
2. How do you declare an array?
3. How are elements of an array stored in memory?
4. Define a matrix.
5. Define sparse matrix. What are the various methods of representing a sparse matrix?
6. Write the program for converting a two dimensional matrix to Yale's form of sparse matrix representation.

2.3.9 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, "Data Structures Using C++", PHI.
2. M. A. Weiss, "Data Structures and Algorithm Analysis in C++", Pearson Education.
3. R. Sedgewick, "Algorithms in C++", Pearson Education.
4. S. Lipschutz, "Data Structures", Tata McGraw Hill.
5. Donald E. Knuth, "The Art Of Computer Programming", Vol 1-4. 3rd ed., Addison Wesley.

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.4 **AUTHOR : DR DHARAMVIR SHARMA**

STACKS

2.4.1 Objective of the lesson

A stack is the way you groups things together by placing one thing on top of another and then removing things one at a time from the top of the stack. It is amazing that something this simple is a critical component of nearly every program

that is written. In this lesson, you'll learn how to create and use a stack in your programs.

2.4.2 Introduction

A stock is a type of data Structure – a means of storing information in a computer. When a new object is inserted in a Stock, it is placed on top of all the previously entered objects. When you hear the term "stack" used outside the context of computer programming, you might envision a stack of dishes on your kitchen counter. This organization is structured in a particular way: the newest dish is on top and the oldest is on the bottom of the stack.

Each dish in a stack is accessed using fifo: first in, first out. The only way to access each dish is from the top of the stack. If you want the third dish (the third oldest on the stack), then you must remove the first two dishes from the top of the stack. This places the third dish at the top of the stack making it available to be removed.

There's no way to access a dish unless the dish is at the top of the stack. You might be thinking stacks are inefficient and you'd be correct if the objective was to randomly access things on the stack. There are other data structures that are ideal for random access, which you'll learn about throughout this lesson.

However, if the object is to access things in the order in which they were placed on the stack, such as computer instructions, stacks are efficient. In these situations, using a stack makes a lot of sense.

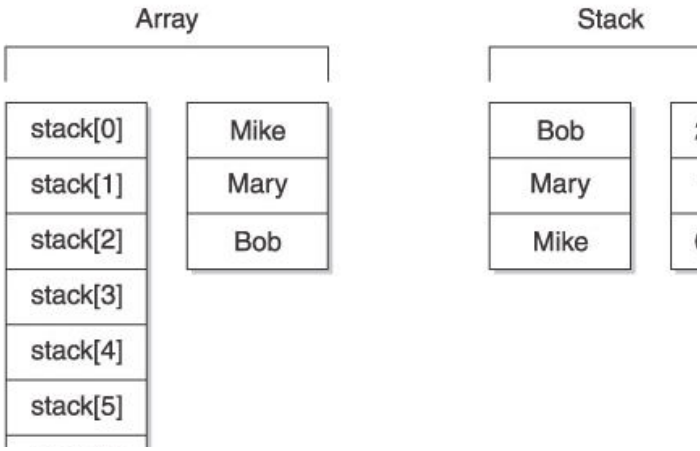


Figure 2.4.1: A stack and an array are two different things: an array stores values in memory; a stack tracks which of the array elements is at the top of the stack.

2.4.3 Operations on Stacks

Programmers use arrays to store values that are referenced by a stack. An array consists of a series of array elements, each of which is similar in concept to a variable. The stack contains the index of the array element that is at the top of the stack.

Figure 2.4.1 is the way some programmers envision an array used with a stack. This example shows an array called stack with 8 array elements. The entire array contains values that are referenced by the stack. Three array elements are assigned values, while the other array elements are empty and can be used when new items are placed on the stack (see the upcoming section "Push").

Mike is the first value placed on the stack. You know this because Mike is at the bottom of the stack. Bob is the last item placed on the stack because Bob is the top item on the stack.

2.4.3.1 Push

Programmers use the term "push" to mean placing an item on a stack. Push is the direction that data is being added to the stack. Think of this as pushing items down on the stack to move the items already on the stack down to make room for the next item.

Here's what actually happens. The new value is assigned to the next available array element and the index of that array element becomes the top of the stack, as shown in Figure 2.4.2. The program increments the current index of the stack by 1. In this example, the index is incremented by 1, resulting in index 3 being at the top of the stack, which is the index of the new values assigned to the array.

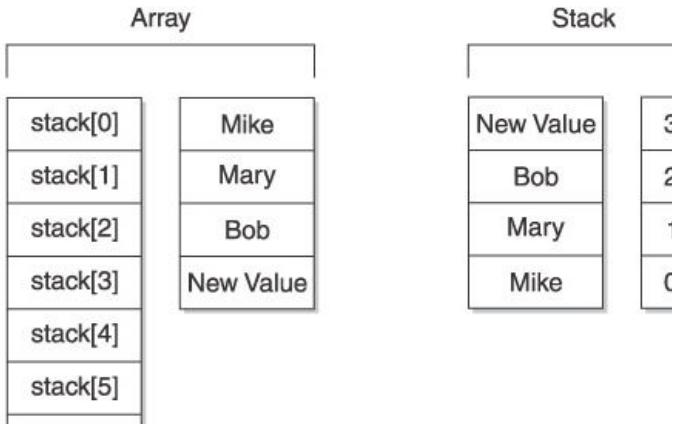


Figure 2.4.2: The new value is assigned to the next array element and its index becomes the top of the stack.

2.4.3.2 Pop

Popping is the reverse process of pushing: it removes an item from the stack. It is important to understand that popping an item off the stack doesn't copy the item. Once an item is popped from the stack, the item is no longer available on the stack, although the value remains in the array.

Here's what really happens. Remember that the top of the stack contains an index of the array element whose value is at the top of the stack. In Figure 2.4.2, index 3 is at the top of the stack, which means New Value in array element 3 is at the top of the stack.

When you pop New Value from the stack, you decrement the index at the top of the stack. That is, you make its index 2 instead of 3. This makes Bob the new value at the top of the stack (see Figure 2.4.3). Notice that New Value and array element 3 remain untouched in the array because popping a value from the stack only alters the stack, not the underlying array.

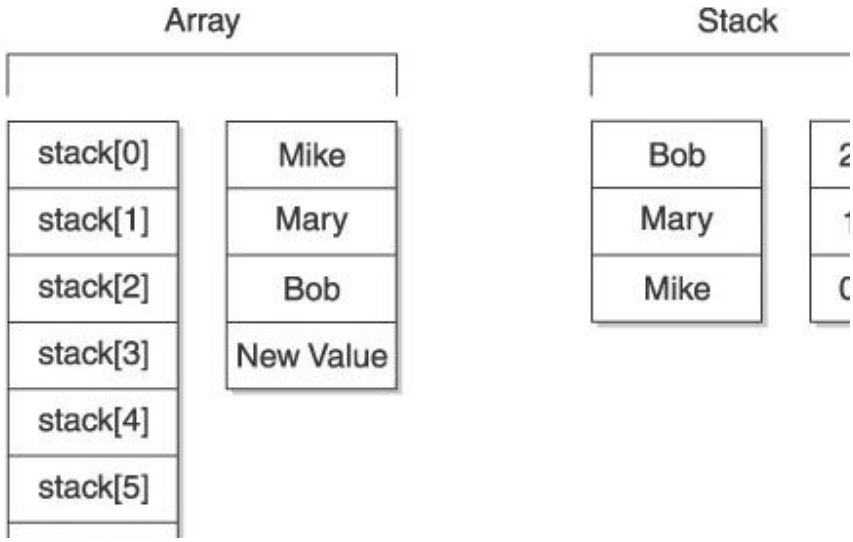


Figure 2.4.3: All values move toward the top of the stack when the top item is popped off the stack.

2.4.3.3 Checking overflow and underflow

Deletion of stack element is not possible when the stack is empty. This is called underflow checking. Similarly an element can not be inserted when the stack is full. This is called overflow checking.

While implementing Push operation, before actual insertion of the element in the stack, its overflow condition must be checked. And while implementing Pop operation, before actual deletion of stack element, its underflow condition must be checked.

2.4.4 Creating a Stack in C++

You can create a stack in C++ by defining a stack class and declaring an instance of the class. The Stack class requires three attributes and several member functions, which are defined as you learn about them. You'll begin by defining a basic stack class that has only the components needed to create the stack.

The class is called Stack but you can call it any name you wish. This class definition is divided into a private access specifier section and a public access specifier section. The private access specifier section has attributes and member functions (although not in this example) that are accessible only by a member function defined in this class. The public access specifier section has attributes (although not in this example) and member functions that are accessible by using an instance of the class.

The private access specifier section of the Stack class defines three attributes: size, top, and values, all of which are integers. The size attribute stores the number of elements in the stack, the top attribute stores the index of the top element of the stack, and the values attribute is a pointer to the stack, which is an array. The stack in this example is a stack of integer values, but you can use an array of any data type, depending on the nature of your program.

Only one member function is defined in the Stack class, although we'll define other member functions for the class in upcoming sections of this lesson. For now, let's keep the example simple and easy to understand. This member function is called Stack, which is the constructor of the class. A constructor is a member function that has the same name as the class and is called when an instance of the class is created. The code for this is on the next page.

Several things are happening in the constructor. First, the constructor receives an integer as an argument that is passed when the instance of the Stack class is declared. The integer determines the number of elements in the stack and is assigned to the size variable.

The first statement might look a bit confusing. It appears that the value of the size variable from the argument list is being assigned to itself but that's not the case. Actually, the size variable from the argument list is local to the Stack member

function. The this->size combination refers to the size attribute of the Stack class, as shown here:

```
this->size = size;
```

Programmers use the this pointer within a member function to refer to the current instance of the class. In this example, the 'this' pointer uses the pointer reference (->) to tell the computer to use the size attribute of the class. As you'll remember from your C++ programming class, the pointer reference is used when indirectly working with a class member, and the dot operator is used when you are directly working with a class member.

This allows the compiler to distinguish between a class variable and local variable that have the same name. This means that the value of the size variable that is passed as an argument to the Stack member function is assigned to the size attribute, making the value available to other members of the Stack class.

You can see how the size attribute is used in the next statement. This statement does two things. First, it allocates memory for the stack by using the new operator (new int[size]). The new operator returns a pointer to the reserved memory location. The size is the size attribute of the class and determines the size of the array. The array is an array of integers.

Next, the pointer to the array of integers is assigned to the values attribute of the class. The values attribute is a pointer variable that is defined in the private attribute section of the Stack class.

The last statement in the Stack member function assigns a –1 to the top attribute. The value of the top attribute is the index of the top element of the stack. A –1 means that that stack doesn't have any elements. Remember from your programming class that index values are memory offsets from the start of the array. Index 0 means "move 0 bytes from the start of the array." So index –1 is just a convenient way to say that the stack is empty.

We'll expand on the definition of the Stack class in the next section but for now let's create an instance of the Stack class. The instance is declared within the main() function of this example. Three things are happening here. First, the new operator is creating an instance of the stack in memory. The new operator returns a pointer to that memory location.

Next, the statement declares a reference to the stack, which is called myStack. The reference is a pointer. The final step is to assign the pointer returned by the new

operator to the reference. You then use the reference (myStack) as the name of the instance of the Stack class throughout the program.

```
public class Stack{
  private:
    int size;
    int top;
    int* values;
  public:
    Stack(int size){
      this->size = size;
      values = new int[size];
      top = -1;
  }
};
void main(){
  Stack *myStack = new Stack(10);
}
```

2.4.4.1 Implementing IsFull

We'll create different member functions for each step, beginning by defining a member function that determines if there is room on the stack i.e. it check the overflow condition. We'll call it IsFull() and define it in the following code. The IsFull() member function is simple. It compares the value of the top attribute with the one less than the value of the size attribute.

```
bool IsFull(){
  if(top < size-1){
    return false;
  }
  else{
    return true;
```

```
    }
  }
```

With the IsFull() member function defined, we'll move on to defining the push() member function, as shown in section 2.5.4.3.

2.4.4.2 Implementing IsEmpty

The IsEmpty() member function determines if there are any values on the stack i.e. the underflow condition. Let's define the IsEmpty() member function in this next example. The IsEmpty() member function contains an if statement. The condition expression of the if statement compares the value of the top attribute to –1. Remember that –1 is the initial value of the top attribute when the instance of a stack is declared. If the top attribute is equal to –1, then a true is returned because the stack is empty; otherwise, a false is returned.

```
  bool IsEmpty(){
    if(top == -1){
      return true;
    }
    else{
      return false;
    }
}
```

2.4.4.3 Creating a Push Member Function in C++

Now that you've seen how to define a class that creates a stack, we'll show you how to define additional member functions that enable the class to push values onto the stack. Pushing a value onto the stack is a two-step process. First, you must determine if there is room on the stack for another value. If there is, you push the value onto the stack; otherwise, you don't.

The value of the top attribute is –1 when the instance of the stack is declared. Suppose the value of size is 10. The condition expression in the if statement of the IsFull() member function determines if the value of top, which is –1, is 1 less than the value of size. Since the value of size is 10, the condition expression compares –1 < 9. If top is greater than or equal to 9, then a true is returned; otherwise, a false is returned.

Why do you subtract 1 from the size of the stack? The value of the top attribute is an index of an array element. Remember that the index begins with zero. In contrast, the size is actually the number of array elements in the stack. Therefore, the tenth array element on the stack has an index of 9.

The push() member function pushes a value onto the stack. The value being pushed onto the stack is passed as an argument to the push() member function and is assigned to the variable x in this example.

Before doing anything else, the push() member function determines if there is room on the stack by calling the IsFull() member function in the condition expression of the if statement. The condition expression might look a little strange because the call is preceded by an exclamation point (!) so we'll take apart the condition expression to explain what is really happening here.

Remember from your programming classes that statements within an if statement execute only if the condition expression is true. This means the condition expression must be true for the value passed to the push() member function to be placed on the stack.

Here's a slight problem. We're calling the IsFull() member function to determine if there is room on the stack for another value. However, the IsFull() member function returns false if there is room and true if there isn't room. A false causes the push() member function to skip statements that place the value on the stack. We really need the IsFull() member function to return a true if there is room available, not a false. Rather than rewrite the IsFull() member function, we use the exclamation point to reverse the logic. As you remember from your programming class, the exclamation point is the not operator—that is, a false is treated as a true, which causes the value to be placed on the stack.

There are two statements within the if statement. The first statement increments the value of the top attribute, which is the index of the last value placed on the stack. If the stack is empty, then the current value of the top attribute is –1. Incrementing –1 changes the value of the top attribute to 0, which is the index of the first array element of the stack. The last statement in the if statement assigns the value passed to the push() member function to the next available array element.

```
void push(int x){
  if(!IsFull()){
    top++;
    values[top] = x;
```

```
    }
}
```

2.4.4.4 Creating a Pop Member Function in C++

The pop() member function of the Stack class has the job of changing the index that is at the top of the stack and returning the value of the corresponding array to the statement that calls the pop() member function. The next example defines the pop() member function.

The first statement in the definition declares an integer variable called retVal that stores the value returned by the pop() member function. The retVal is initialized to zero. Next, the IsEmpty() member function is called in the condition expression of the if statement to determine if there is a value at the top of the stack. Notice the exclamation point reverses the logic as it did in the pop() member function.

Statements within the if statement should only execute if the IsEmpty() member function returns a false, meaning the stack is not empty. Therefore, we need to use the exclamation point to reverse the logic of the condition expression to make the condition expression true if the IsEmpty() member function returns a false.

Two steps occur within the if statement. First, the value at the top of the stack is assigned to the retVal variable by referencing the values array using the index contained in the top attribute. Next, the value of the top attribute is decremented. The return retVal is then returned by the pop() member function.

```
int pop(){
  int retVal = 0;
  if(!IsEmpty()){
     retVal = values[top];
     top--;
  }
  return retVal;
}
```

2.4.5 An Example of Stack Implementation

Now that you understand how to create and use a stack, we'll pick up the pace and explore an industrial-strength stack. You've may have heard the term industrial

strength used in relation to programming and may be curious what this really means.

Industrial strength is a term used in industry that implies a product is designed to withstand stress. Industrial strength can be used to describe any kind of product but in this case the product is the program that creates and uses a stack.

Programs used to illustrate the concepts of a stack in this lesson are bare bones and lack the robust features that are found in industrial-strength programs. A bare-bones program is what you need when you're learning the concepts of stacks and other data structures because the program contains only statements that pertain to what you are learning. However, once you learn the concept, you need to see how it's applied in a real-world program. In this section, you'll take a look at how a stack is created and used in an industrial-strength C++ program.

We'll use as an example an industrial-strength C++ program that creates and uses a stack. The program is contained within three files, stack.h, stack.cpp, and stackDemo.cpp. The stack.h file is a header file that contains the definition of the Stack class, which is the "blueprint" of the Stack class. The stack.cpp file is a source code file that contains the implementation of the member functions of the Stack class. The stackDemo.cpp file contains the source code for the C++ program that declares the instance of the Stack class and calls its member functions. Let's begin by taking a look at the stack.h header file, which is shown in the next code example. As you'll recall from your C++ classes, a header file typically contains definitions and preprocessor instructions. A preprocessor is a program that applies preprocessor instructions to source code before the code is compiled.

The stack.h header file contains one preprocessor instruction, #define, which defines a symbol. Here we've defined the symbol DEFAULT_SIZE and given it a value of 10. The preprocessor then replaces all occurrences of DEFAULT_SIZE with 10 before the code is compiled. The DEFAULT_SIZE is the default size of the stack if the no argument is passed to the constructor. Function parameters in C and C++ can be assigned default values in the function prototype as long as those arguments are at the end of the argument list. If the size value is not passed in, it gets defaulted to the value of DEFAULT_SIZE, which is 10 in our example.

The stack.h file also contains the definition of the Stack class. The Stack class definition has the same size, top and values attributes you saw in the previous C++ example. However, the definition of member functions is different from what you saw because member functions are implemented outside the class definition in the stack.cpp source code file. The header file contains only the prototype of the functions, which make up the blueprint for the class.

From your C++ class, you'll remember that only the prototype or signature of a member function needs to be included in a class definition. The implementation of the member function can be outside the class definition. There are two important reasons for keeping the definition (header file) and implementation (source) in separate files:

It allows you to provide a commercial software application programmer interface to a programmer without handing over your source code. You provide the programmer with your header files, which they will use to compile their code (they only need header files to compile the code). You provide your source code in the form of precompiled libraries that are referenced by the programmer's program during linking.

The class definition contains signatures of six member functions. The first member function is called Stack, which is the constructor that you learned about previously in this lesson. Previously, you learned that the constructor is passed an integer representing the size of the stack. In the real-world version, the program sets a default size that can be overridden when an instance of the class is created in the program. The default size is specified by using the DEFAULT_SIZE, which is 10 (see #define). The next member function is ~Stack() and is the destructor of the class. A destructor is the last member function that is called when the instance of the class goes out of scope and dies. A constructor must always be the same name as the class and begin with a tilde (~). By definition, destructors cannot accept any arguments. The purpose of the destructor is to free memory that is used by the stack or do any other sort of cleanup that's required.

The remaining member functions are the same functions that you learned about previously in this lesson.

```
//stack.h
#define DEFAULT_SIZE 10
class Stack {
  private:
     int size;
     int top;
     int* values;
  public:
     Stack(int size = DEFAULT_SIZE);
```

```
    virtual ~Stack();
    bool IsFull();
    bool IsEmpty();
    void push(int);
    int pop();
};
```

The stack.cpp file is a source code file that contains the implementation of the Stack class's member functions. We placed these in a different file from the class definition because it is easier to read and maintain as well as for other reasons explained previously. The file begins with the preprocessor instruction #include that tells the computer to evaluate the contents of the stack.h file before compiling the stack.cpp file so it "knows" about the Stack class definition before compiling the program. Member functions in the stack.cpp file will be familiar to you because all except one are the same member functions that you learned about previously in the lesson. However, the names of the member functions might be confusing at first glance because each name begins with the name of the class followed by two colons (::). The two colons are called the scope resolution operator.

You must precede the name of a member function with the class name and scope resolution operator if the member function is defined outside the class definition. Think of this as telling the computer that the member function belongs to the Stack class. The ~Stack() member function frees memory used by the stack. It does this by using the delete operator and referencing the name of the array used for the stack. In this example, values is the name of the array.

To avoid memory leaks, freeing memory is important whenever memory is dynamically allocated. The square brackets ([]) are used with delete because the object being removed from memory was dynamically created.

The stack.cpp is compiled as you would compile any source code. The result is an object file that is joined together with the compiled stackDemo.cpp source code file by the linker to create an executable program called a load module.

```
//stack.cpp
#include "stack.h"
Stack::Stack(int size){
  this->size = size;
```

```
    values = new int[size];
    top = -1;
}
Stack::~Stack(){
    delete[] values;
}
bool Stack::IsFull(){
    if(top < size-1){
        return false;
    }
    else{
        return true;
    }
}
bool Stack::IsEmpty(){
    if(top == -1){
        return true;
    }
    else{
        return false;
    }
}
void Stack::push(int x){
    if(!IsFull()){
        top++;
        values[top] = x;
    }
}
```

```
int Stack::pop(){
    int retVal = 0;
    if(!IsEmpty()){
        retVal = values[top];
        top--;
  }
  return retVal;
}
```

Finally, we come to the stackDemo.cpp program, which is the C++ program that creates the instance of the Stack class. The first statement creates the stack in a three-step process. The first step is to use the new operator to allocate space in memory for the Stack class by calling the constructor of the class. The new operator returns the memory location of the stack. The second step is to declare a pointer called stack. The last step is to assign the memory location returned by the new operator to the stack pointer. In this example, we used the default size for the stack, which is 10 elements. We can pass the Stack() constructor an integer to change the size of the stack. The push() member function is called three times. Each time a different value is placed on the stack. Notice that the -> pointer is used instead of the dot operator. You must do this because stack is a pointer to an instance of the class and not the instance itself.

The last portion of the stackDemo.cpp program calls the pop() member three times. Each time a value is removed from the top of the stack and displayed on the screen.

```
//stackDemo.cpp
void main() {
  Stack *stack = new Stack();
  stack->push(10);
  stack->push(20);
  stack->push(30);
  for(int i=0; i<3; i++)  {
    cout << stack->pop() << endl;
  }
```

```
}
```

2.4.6 Summary

Stack is a linear data structures extensively used in varied application of computer programming. It is based in Last In First Out (LIFO) principle. A top pointer maintains the count of elements present in the stack. Push operation is used to insert elements in stack and Pop operation is used to remove an element from top of the stack. Underflow and overflow conditions are check using IsEmpty and IsFull functions, respectively.

2.4.7 Questions

1. What is a stack?
2. What is the purpose of the push() member method?
3. What is the purpose of the pop() member method?
4. What is the purpose of the IsFull() member method?
5. What is the purpose of the IsEmpty() member method?
6. What kind of value is assigned to the top attribute?
7. Why is the top attribute initialized to –1?

2.4.8 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley.

LESSON NO. 2.5 **AUTHOR : DR DHARAMVIR SHARMA**

APPLICATIONS OF STACKS

2.5.1 Objective of the lesson

Stacks have wide applications in computers programming. In this lesson we shall discuss some most common applications of stacks.

2.5.2 Introduction

Stacks are used in various applications of computers. These may be used for reversal of a string, matching or parenthesis in expression or programs, conversion of infix expression to postfix, evaluation of postfix expression, simulating recursion etc.

2.5.3 Parenthesis Matching

The simplest application of stacks deals with removing the parentheses, this task ensures that all paired symbols match correctly, including parentheses ( ), brackets [ ] and braces { }. This program will read strings of input from the user, indicating whether the parentheses, brackets and braces match up correctly. Use a character stack to store the necessary characters and make comparisons. All characters other

than parentheses, brackets and braces are ignored for purposes of matching. Many applications of stacks deal with determining whether an input string (or file or sequence of symbols) is a member of a context-free language. Many programming languages are context free.

As an example, consider the language of balanced parentheses. A sequence of symbols involving ()+* and integers, is said to have balanced-parentheses if each right parenthesis has a corresponding left parenthesis that occurs before it. For example: These expressions have balanced parentheses:

```
2*7                 // no parens - still balanced
(1+3)
((2*16)+1)*(44+(17+9))
```

These expressions do not:

```
(44+38
)                   // a right paren with no left
(55+(12*11)         // missing a )
```

How do we tell if a sequence of characters represents a balanced-parentheses expression? Use stacks. Idea:

- Start with an empty stack.
- For each left parenthesis, push.
- For each right parenthesis, pop.
- For each non-parenthesis character, do nothing.
- The expression is balanced if the stack is empty and there are no more characters to process.
- It is not balanced if either after the last character the stack is not empty (too many left parens), or if the stack is empty and a right paren is encountered (a right without a left).

What items do we push onto the stack? If we are just interested in knowing if the expression is balanced, it does not matter. For clarity, we might choose a stack of characters, pushing "(" onto the stack for each left parenthesis.

We can do more. We can match multiple types of delimiters. For example, we might want to match (), [], and {}. In that case we can push the left delimiter onto the stack and when we pop, do a check, as in the pseudocode:

```
if (next character is a right delimiter) then {
  if (stack is empty) then
    return false
  else {
    pop the stack
    if (right delimiter is corresponding version
        of what was popped off the stack) then
          continue processing
    else
       return false
  }
}
```

Other variations on parentheses matching include:

- valid prefix: return true if the expression is a valid prefix for a balanced parentheses expression, for example "((3+4)+" is a valid prefix. Same idea, but we can return true as long as there is no attempt to pop an empty stack.
- identifying matches: it is often useful to not only know that an expression is balanced, bust to see which pair correspond. For example, in some editors (like Emacs), when you type a right paren the corresponding left paren is momentarily highlighted. Think about how you might adapt a paren matching function to do this.

```
int main(){
  char c;
  int k;
ifstream f;
 f.open( "input.txt" );
 if( ! f ){
   cout << "Error opening file. Quitting.\n";
   exit(-1);
 }
  //initialize
```

```
    stk_init();
      c = getchar();
      putchar(c);
    // loop through file one char at a time
    while (c!=EOF)
    {
      // process one character push (,  pop if ),
      // check stack empty if \n, ignore others
      // if error, move to next line
       if (c == '('){
         stk_push(c);
       }
       else if (c == ')') {
            stk_top (c);
            if (c == '(')
            stk_pop (c);
            else// pop stack, if empty have missing (
       }
       else if (c == '\n'){
         // check on missing ) set up for next line
         stk_init();
       }
       if (stk_error()) //skip rest of line
       else{ // get next char
       }
     } // endwhile
    return 0;

}
```

2.5.4 Conversion from Infix to Postfix expression

Any expression in the standard form like "2*3-4/5" is an Infix (In order) expression. The Postfix (Post order) form of the above expression is "23*45/-". In a postfix expression the operands precede the operator.

In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for the conversion is as follows:

- Scan the Infix string from left to right.

- Initialise an empty stack.
- If the scanned character is an operand, add it to the Postfix string. If the scanned character is an operator and if the stack is empty Push the character to stack.
  - If the scanned character is an Operand and the stack is not empty, compare the precedence of the character with the element on top of the stack (topStack). If topStack has higher precedence over the scanned character Pop the stack else Push the scanned character to stack. Repeat this step as long as stack is not empty and topStack has precedence over the character.

    Repeat this step till all the characters are scanned.
- (After all characters are scanned, we have to add any character that the stack may have to the Postfix string.) If stack is not empty add topStack to Postfix string and Pop the stack. Repeat this step as long as stack is not empty.
- Return the Postfix string.

Example :

Let us see how the above algorithm will be implemented using an example. Infix String : a+b*c-d

Initially the Stack is empty and our Postfix string has no characters. Now, the first character scanned is 'a'. 'a' is added to the Postfix string. The next character scanned is '+'. It being an operator, it is pushed to the stack.

Next character scanned is 'b' which will be placed in the Postfix string. Next character is '*' which is an operator. Now, the top element of the stack is '+' which has lower precedence than '*', so '*' will be pushed to the stack.

Stack

The next character is 'c' which is placed in the Postfix string. Next character scanned is '-'. The topmost character in the stack is '*' which has a higher precedence than '-'. Thus '*' will be popped out from the stack and added to the Postfix string. Even now the stack is not empty. Now the topmost element of the stack is '+' which has equal priority to '-'. So pop the '+' from the stack and add it to the Postfix string. The '-' will be pushed to the stack.

Stack

abc*+

Postfix String

Next character is 'd' which is added to Postfix string. Now all characters have been scanned so we must pop the remaining elements from the stack and add it to the Postfix string. At this stage we have only a '-' in the stack. It is popped out and added to the Postfix string. So, after all characters are scanned, this is how the stack and Postfix string will be:

Stack

abc*+d-

Postfix String

End result :

Infix String : a+b*c-d

Postfix String : abc*+d-

Example program:

```
#include <iostream.h>

#include <conio.h>

#include <string.h>

#include <ctype.h>
```

```
#include "CHRStack.h"
#define MAXSIZE 100
int IsOperator(char c){
    if (ch == '^' || ch == '*' || ch == '/' || ch == '+' || ch == '-')
        return 1;
    return 0;
}
void main(){
    Stack s;
    char c,ch;
    char str[MAXSIZE];
    int l,i;
    int HP;
    clrscr();
    cout << "Enter the input expression in infix notation -> ";
    cin.getline(str, MAXSIZE);
    l = strlen(str);
    str[l++] = ')';
    str[l] = '\0';
    s.Push('(');
    cout << "The corresponding expression in postfix notation is -> ";
    for (i = 0;i < l;i++){
        if (isalpha(str[i])){
            cout << str[i];
            continue;
        }
        if (str[i] == '('){
            s.Push(str[i]);
```

```
        continue;
    }
    if (IsOperator(str[i])){
        HP = 1;
        while (HP){
            ch = s.GetTopElement();
            if (IsOperator(ch)){
                switch(str[i]){
                    case '^':
                            if (ch == '^'){
                                cout << s.Pop();
                                HP = 1;
                            }
                            else HP = 0;
                            break;
                    case '*':
                    case '/':
                            if (ch == '^' || ch == '*' || ch == '/'){
                                cout << s.Pop();
                                HP = 1;
                            }
                            else HP = 0;
                            break;
                    case '+':
                    case '-':
                            if (IsOperator(ch)){
                                cout << s.Pop();
                                HP = 1;
```

```
                        }
                        else HP = 0;
                        break;
                    }
                }
                else HP = 0;
            }
            s.Push(str[i]);
        }
        if (str[i] == ')'){
            while (1){
                if (IsOperator(s.GetTopElement())) cout << s.Pop();
                else break;
            }
            s.Pop();
        }
    }
    getch();
}
```

2.5.5 Evaluation of Postfix Expression

In normal algebra we use the infix notation like a+b*c. The corresponding postfix notation is abc*+. The algorithm for the conversion is as follows:

- Scan the Postfix string from left to right.
- Initialise an empty stack.
- If the scanned character is an operand, add it to the stack. If the scanned character is an operator, there will be atleast two operands in the stack.
  - If the scanned character is an Operator, then we store the top most element of the stack(topStack) in a variable temp. Pop the stack. Now evaluate

topStack(Operator)temp. Let the result of this operation be retVal. Pop the stack and Push retVal into the stack.

Repeat this step till all the characters are scanned.

- After all characters are scanned, we will have only one element in the stack. Return topStack.

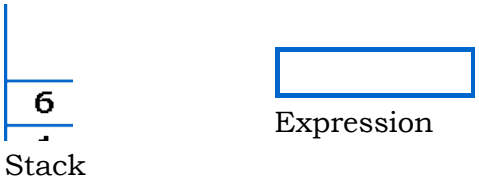
Example:

Let us see how the above algorithm will be implemented using an example. Postfix String: 123*+4-

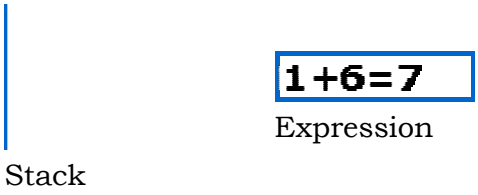
Initially the Stack is empty. Now, the first three characters scanned are 1, 2 and 3, which are operands. Thus they will be pushed into the stack in that order.

Next character scanned is "*", which is an operator. Thus, we pop the top two elements from the stack and perform the "*" operation with the two operands. The second operand will be the first element that is popped.

The value of the expression(2*3) that has been evaluated(6) is pushed into the stack.



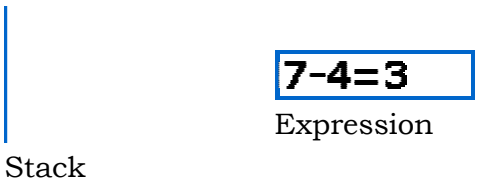
Next character scanned is "+", which is an operator. Thus, we pop the top two elements from the stack and perform the "+" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(1+6) that has been evaluated(7) is pushed into the stack.

Next character scanned is "4", which is added to the stack.

Next character scanned is "-", which is an operator. Thus, we pop the top two elements from the stack and perform the "-" operation with the two operands. The second operand will be the first element that is popped.



The value of the expression(7-4) that has been evaluated(3) is pushed into the stack.

Now, since all the characters are scanned, the remaining element in the stack (there will be only one element in the stack) will be returned.

End result:

Postfix String: 123*+4-

Result: 3

Example Program

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
#include <ctype.h>
#include "INTStack.h"
#define MAXSIZE 100
int IsOperator(char c){
   if (ch == '^' || ch == '*' || ch == '/' || ch == '+' || ch == '-')
      return 1;
   return 0;
}
void main(){
        Stack s;
        int i,l,t,a,b;
        char str[MAXSIZE];
        clrscr();
        cout << "Enter the expression in postfix (use comma as separator) -> ";
        cin.getline(str,MAXSIZE);
        l = strlen(str);
        for (i = 0;i < l;i++){
                if (isdigit(str[i])){
                        s.Push(str[i++]-'0');
                        while (isdigit(str[i])){ //For handling multi-digit numbers
                                int t = s.Pop() * 10 + str[i] - '0';
```

```
                        s.Push(t);
                        i++;
                    }
                    continue;
                }
                if (IsOperator(str[i])) {
                    b = s.Pop();
                    a = s.Pop();
                    switch(str[i]){
                        case '^':   t = a^b;
                                    break;
                        case '*':   t = a*b;
                                    break;
                        case '/':   t = a/b;
                                    break;
                        case '+':   t = a+b;
                                    break;
                        case '-':   t = a-b;
                                    break;
                    }
                    s.Push(t);
                }
            }
            cout << "Result of expression is -> " << s.Pop();
            getch();
}
```

2.5.6 Recursion

Recursion is calling a procedure from itself, directly (simple recursion) or via calls to other procedures (mutual recursion). The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions.

Recursion is usually less computationally efficient, compiler generated procedure calls introduce overhead when storing all variables before a procedure call and loading them back upon entering the procedure.

Automatic optimizations reduce the overhead associated with procedure calls by eliminating redundant store/load operations to some degree but of course hand optimizing may improve upon that. Efficiency is usually not the most important consideration when choosing an algorithm, in the name of good programming we already agreed to gladly suffer some measure of inefficiency.

Factorial

A classic example of a recursive procedure is the function used to calculate the factorial of an integer.

Function definition: $fact(n) = \begin{cases} 1 \\ n \times fact(n-1) \end{cases}$

Pseudocode (recursive):
function factorial is: input: integer n such that n >= 1 output: [n × (n-1) × (n-2) × … × 1] 1. if n is 0, return 1 2. otherwise, return [n × factorial(n-1)] end factorial

Simulating Recursion

Knowing the process by which recursion passes data upward and downward through the called modules, you can isolate and preserve the variables unique to each recursive step and simply loop a given piece of code to achieve simulated recursion. Since looping would start the code process at the same place each time, you must keep a place marker, indicating just where to begin the processing of the current iteration.

The data used by each iterative step can be in any form and is a function of the programming requirements. A binary switch is probably the best form for a place marker. A combination of the required variable data and a place marker form a snapshot of the conditions during any step of the recursion. You should construct a table where snapshots can be stored. A pointer to this stack of snapshots allows the program to select the appropriate set of values for a particular iteration of the recursive process. The depth of the stack must be such that it can contain all the steps necessary to achieve the objective of the program being executed.

Example Program

```
#include <iostream.h>
#include <conio.h>
#include <string.h>
#include "intstack.h"
void main(){
   Stack s;
   int n,t;
   clrscr();
   cout << "Enter the number for computing its factorial (less than 8) -> ";
   cin >> n;
   s.Push(1);
   while (n != 0){
      t = s.Pop() * n--;
      s.Push(t);
   }
```

```
        cout << "Factorial of n " << n << " is n is -> " << s.Pop();
        getch();
    }
```

2.5.7 Summary

Stack has many applications, including parenthesis matching, infix to postfix conversion, postfix evaluation and simulation of recursion among others. For matching program delimiters stacks are extensively used.

2.5.8 Questions

1. What do you mean by infix and postfix expressions?
2. Convert the following infix expressions to postfix
   a. a+b*c+d
   b. a*b+c*d
3. Write the pseudo code for converting an infix expression to postfix.
4. Write the pseudo code for evaluating a postfix expression.
5. Write the pseudo code for simulating recursion using stacks for computing factorial.

2.5.9 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.6 **AUTHOR: DHARAM VEER SHARMA**

QUEUES

2.6.1 Objective of the lesson

You probably never thought that waiting in line in the supermarket would help you become a whiz at data structures but it's a big help. The checkout line at a supermarket is similar to the way data structures are organized. We're the "things" organized by the supermarket line and the same kind of organization is used for data within your program. The checkout line in your program is called a queue. In

this lesson, you'll learn the ins and outs of implementing a queue within your program.

2.6.2 Introduction

A queue is a container of objects that are inserted and removed according to FIFO principle. A queue is like the checkout line at the supermarket where the first customer is at the front of the line, the second customer is next in line and so on until you reach the last customer who is at the back of the line. Customers check out of the supermarket in the order they arrive in the line. That is, the first customer is the first one to check out. This is referred to as first in, first out (FIFO).

The same concept applies to a queue in your program. A queue is a sequential organization of data. Data is accessible using FIFO. That is, the first data in the queue is the first data that is accessible by your program. In this lesson, you will explore the simplest type of queue, a fixed size, first in, first out queue using an array.

Programmers use one of two kinds of queues depending in the objective of the program, a simple queue or a priority queue. A simple queue organizes items in a line where the first item is at the beginning of the line and the last item is at the back of the line. Each item is processed in the order in which it appears in the queue. The first item in line is processed first, followed by the second item and then the third until the last item on the line is processed. There isn't any way for an item to cut the line and be processed out of order.

A priority queue is similar to a simple queue in that items are organized in a line and processed sequentially. However, items on a priority queue can jump to the front of the line if they have priority. Priority is a value that is associated with each item placed in the queue. The program processes the queue by scanning the queue for items with high priority. These are processed first regardless of their position in the line. All the other items are then processed sequentially after high priority items are processed.

Queues are very important in business applications that require items to be processed in the order they are received. The supermarket checkout line is a queue that most of us have experienced but you won't be creating a supermarket checkout line in a program unless the program is designed to simulate a checkout line.

In the real world, queues are used in programs that process transactions. A transaction is a set of information such as an order form. Transaction information

is received by a program and then placed in a simple queue waiting to be processed by another part of the program.

Many other applications use a simple queue to maintain the order in which to process items. These include programs that process stock and bond trades and those that process students registering for a course. Queues are also used within a computer to manage printing.

2.6.3 Operations on Queues

Data organized by a queue may be stored in an array. The queue determines the array element that is at the front and back of the queue. The array is not the queue. Likewise, the queue is not the array. Both are two separate things. This is an important concept to grasp and one that may be difficult to understand at first.

Take a look at Figure 2.6.1 and you'll see how an array and a queue are different and yet are linked together to organize data. The array is pictured as a block of elements. The queue is pictured as a circle. The empty boxes are where values are stored in the queue and the numbers correspond to the index of the array that is associated with the queue. To the right of the circle are three values. The front and back values store the index of the front and back of the queue. The size value is the number of elements in the queue, which is 8 in this example.

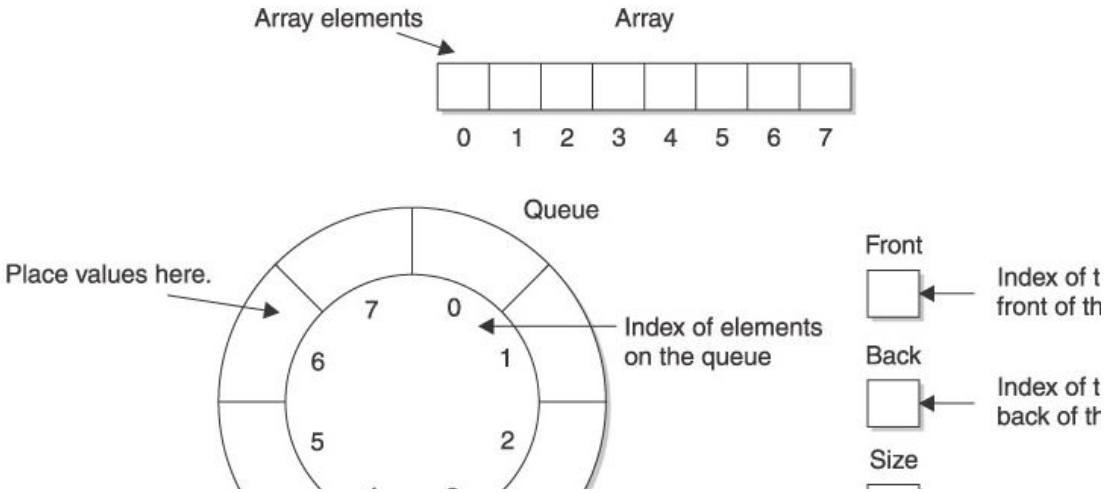


Figure 2.6.1: The queue is different from the array to store data that appears in the queue.

2.6.3.1 Inserting Element in Queue

A value is placed in the queue by performing the enqueue process, which consists of two steps. The first step is to identify the array element that is at the back of the queue. However, this is not necessarily the last element of the array. Remember

that the queue is not the array. The back of the queue is calculated by using the following formula:

back = (back+1) % size

Figure 2.6.2 shows how to use the formula and gives the values for the front, back and size of the queue. The front and back variables are set to zero because the queue is empty and size is set to 8 because the array has 8 elements.

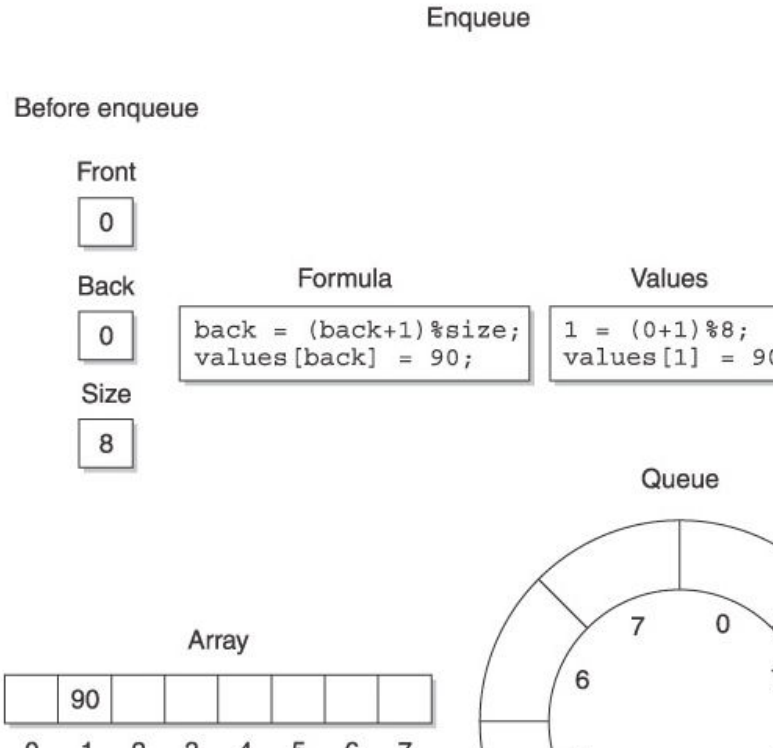


Figure 2.6.2: The enqueue process places a new value at the back of the queue.

The next box shows the formula that identifies the back of the queue and assigns it the value 90. To the right of this box is the same formula with variable names replaced by actual values. Let's take a closer look at this and see how the back of the queue is calculated.

The first operation occurs within the parentheses, where 1 is added to the value of the back variable. The modulus operator determines where the next element should be placed in the queue by performing an integer division and returning the remainder of the division. Although we've described a queue as a checkout line in the supermarket, a queue is actually circular. This is illustrated in the calculation used to determine the back of the queue, as shown here:

(7 + 1) % 8

When you get to the last element in the array at index 7, the calculation returns 0 (8 divided by 8 is 1 and the remainder is 0). So after the last element in the array, you come around to the beginning of the array as the back of the queue. You check to see if you're at the front of the queue before placing an item at the back of the queue so you don't overwrite the item at the front and corrupt the queue.

The second step is to assign the value 90 to array element 1. That is, place the value 90 at the back of the queue. Remember that values are added to the queue from the back just as you go to the back of the checkout line to wait your turn at the supermarket. Notice that the value 90 is assigned to the array in Figure 2.6.2.

2.6.3.2 Deleting Element from Queue

Dequeue is the process that removes a value from the front of the queue. It is important to understand that the value is removed from the queue, not the array. The value always remains assigned to the array until the value is either overwritten or the queue is abandoned.

There are two steps in the dequeue process, as illustrated in Figure 2.6.3.

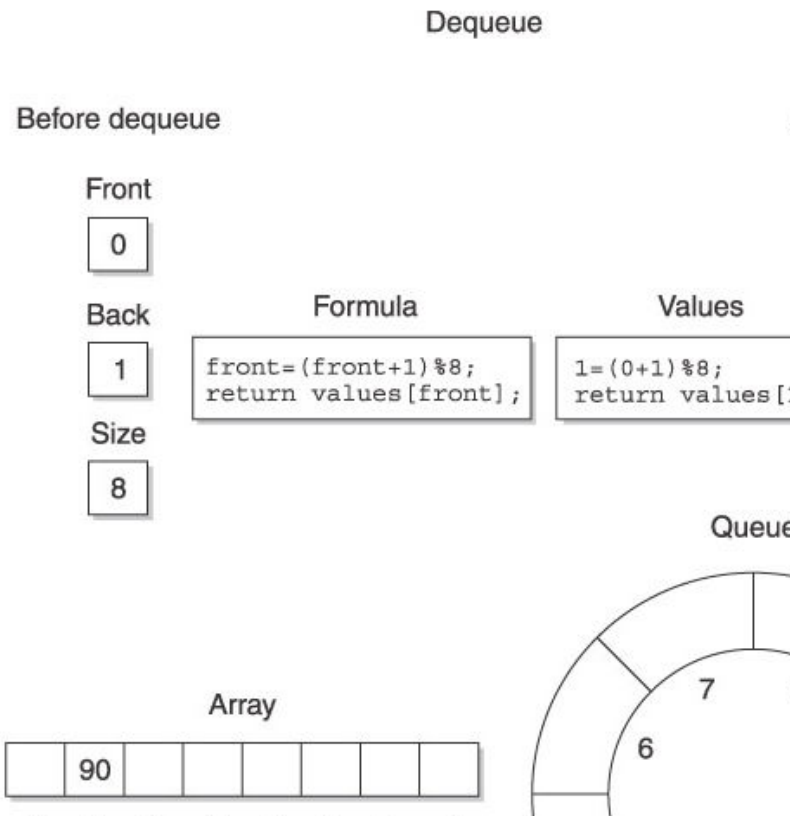


Figure 2.6.3: The dequeue process removes an item from the front of a queue.

The initial step is to calculate the index of the array element at the front of the queue using the following expression:

front = (front+1) % 8

Notice that this expression is very similar to the expression used in the enqueue process to calculate the index of the array element at the back of the queue. The first operation in this expression increments the value of the front variable. As you can see in Figure 2.6.3, the front variable is assigned the initial value zero. Therefore, the result of the first operation is 1. The next operation is to apply the modulus operator, which is identical to the modulus operation performed in the enqueue process. The result of this operation is 1, meaning that the front of the queue is the array element whose index is 1. This value is then assigned to the front variable. Previously in this lesson, you learned that if you were at index 7 in the array, the result of this calculation would be 0 ((7+1)%8 = 0), so you would chase the queue around in a circle.

The final step in the dequeue process is to use the value located at the front of the queue. Typically, the dequeue process is a method and the front of the queue is returned to the statement that called the method.

In Figure 2.6.3, the array element values[1] is at the front of the queue. The value assigned to this element is 90, which was placed at the back of the queue by the previous enqueue process.

Notice that the value 90 remains assigned to the values[1] array element in Figure 2.6.3 because values assigned to the array associated with a queue are not affected when a value is removed from the front of the queue. The queue keeps track of array elements that are at the front and back of the queue, not the front or back of the array. In this case, we're using a simple integer array to illustrate the principles behind implementing the queue data structure. You may come across more complex implementations, where each element in the array is a pointer to a class object or structure. In these cases, you should be concerned about memory management when you perform enqueue and dequeue operations.

2.6.3.3 Checking Underflow and Overflow

Deletion of queue element is not possible when the queue is empty. This is called underflow checking. Similarly an element can not be inserted when the queue is full. This is called overflow checking.

While implementing Enqueue operation, before actual insertion of the element in the queue, its overflow condition must be checked. And while implementing Dequeue operation, before actual deletion of queue element, its underflow condition must be checked.

2.6.4 Queues Using an Array in C++

Now that you understand how queues work with an array, it is time to create a real queue. In this section, you'll create a queue using C++.

The C++ queue program is organized into three files: the queue.h file, the queue.cpp file and the queueProgram.cpp file. The queue.h file, shown next, sets the default size of the array and defines the Queue class. The Queue class declares size, front and back attributes that store the array size and the index of the front and back of the queue. The Queue class also declares a pointer that will point to the array. In addition to these, the Queue class defines a set of member functions that manipulate the queues. These are explained later in this section.

```
//queue.h
#define DEFAULT_SIZE 8
```

```
class Queue{
  private:
    const int size;
    int front;
    int back;
    int* values;
  public:
    Queue(int size = DEFAULT_SIZE);
    virtual ~Queue();
    bool IsFull();
    bool IsEmpty();
    void enqueue(int);
    int dequeue();
};
```

The queue.cpp file contains the implementation of the member functions for the Queue class. There are six member functions defined in this file: Queue(), ~Queue(), IsFull(), IsEmpty(), enqueue() and dequeue().

The Queue() member function is a constructor, which is passed the size of the array when an instance of the Queue class is declared. If the constructor is called with no parameters, then the default size is used, otherwise, the value passed to the constructor is used. The value of the array size is assigned to the attribute size by the first statement within the constructor.

The second statement uses the new operator to declare an array of integers whose size is determined by the size passed to the constructor. The new operator returns a pointer to the array, which is assigned to the values pointer. The last two statements in the constructor initialize the front and back attributes to zero.

The ~Queue() member function is the destructor and uses the delete operator to remove the array from memory when the instance of the Queue class goes out of scope.

2.6.4.1 Implementing IsFull

The IsFull() member function (see Figure 2.6.4) determines if there is room available in the queue by comparing the calculated value of the back of the queue with the value of the front of the queue, as in shown Figure 2.6.4. Notice that the expression that calculates the back of the queue is very similar to the expression in the enqueue process (see the "Enqueue" section of this lesson) and both produce the same result. The queue is full when the back index is 1 behind the front. Placing another element in the queue would overwrite the front element and corrupt the queue. The modulus operator is used again to make this a circular queue, so when you're at element 7 on the back, the next element to look at is element 0.

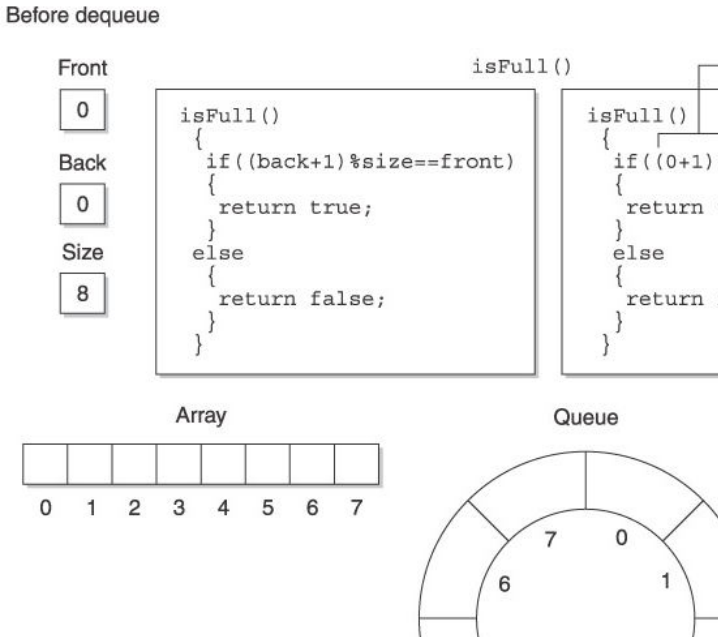


Figure 2.6.4: The IsFull() member function determines if there is room to place another item on the back of the queue.

The IsFull() member function is called by the enqueue() member function before an attempt is made to place a value on the back of the queue. The IsFull() member function returns a true if no more room is available in the queue or a false if there is room available.

2.6.4.2 Implementing IsEmpty

The IsEmpty() member function determines (see Figure 2.6.5) if the queue is empty by comparing the back and front variables. If they have the same values, a true is

returned, otherwise, a false is returned. The IsEmpty() member function is called within the dequeue() member function before it attempts to remove the front item from the queue.

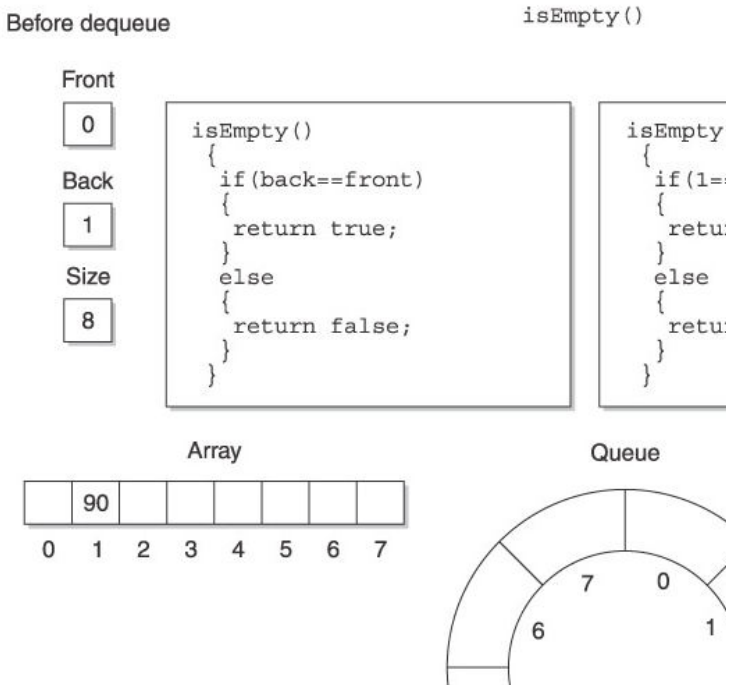


Figure 2.6.5: The IsEmpty() member function determines if the queue contains any values.

2.6.4.3 Implementing Enqueue

The enqueue() member function places an item at the back of the queue, as described in the "Enqueue" section of this lesson. The enqueue() member function is passed the value that is to be placed in the queue. However, before doing so, the IsFull() member function is called to determine if there is room available in the queue. Notice in the following example that the IsFull() member function is called as the condition expression of the if statement. Also notice that the not operator reverses the bool value returned by the IsFull() method. That is, a false is returned by the IsFull() member function if room is available in the queue. The condition expression in the if statement reverses this logic to true so that statements within the if statement execute to place the new item on the back of the queue.

2.6.4.4 Implementing Dequeue

The dequeue() member function removes an item from the queue and returns that item to the statement within the program that calls the dequeue() member function. However, the IsEmpty() member function is called in the condition expression of the if statement within the dequeue() member function, as shown in the next code listing.

The not operator in this expression reverses the logic returned by the IsEmpty() member function. The IsEmpty() member function returns a false if the queue is not empty. The not operator changes this to true, enabling statements within the if statement to remove the front item from the queue and return it to the statement that calls the dequeue() member function.

```
//queue.cpp
#include "queue.h"
Queue::Queue(int size){
   this->size = size;
   values = new int[size];
   front = 0;
   back = 0;
}
Queue::~Queue(){
   delete[] values;
}
bool Queue::IsFull(){
   if( (back+1) % size == front)
      return true;
   else
      return false;
}
bool Queue::IsEmpty(){
   if(back == front)
```

```
        return true;
    else
        return false;
}
void Queue::enqueue(int x){
    if(!IsFull()){
        back = (back+1) % size;
        values[back] = x;
    }
}
int Queue::dequeue(){
    if(!IsEmpty()){
        front = (front+1) % size;
        return queue[front];
    }
    return 0;
}
```

The queueProgram.cpp is where all the action takes place. It is here that an instance of the Queue class is declared and manipulated. As you can see in the next example, the first statement in the program uses the new operator to declare an instance of the Queue class and set the size to 8 elements. The new operator returns a pointer that is assigned to a pointer to an instance of the Queue class.

The next three statements call the enqueue() member function three times to place the values 10, 20 and 30 in the queue, respectively. The program concludes by calling the dequeue() member function three times to display the contents of the queue. Figure 2.6.6 shows the queue and the array after the last call to the enqueue() member function is made.

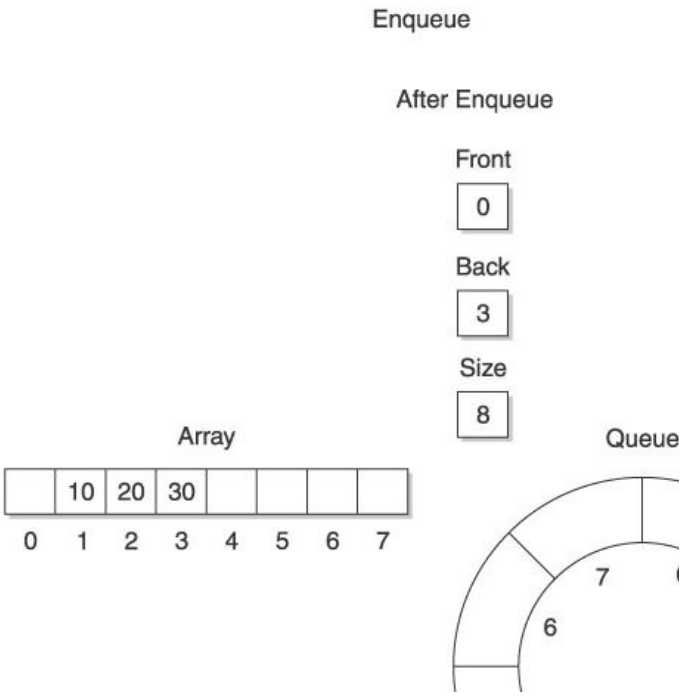


Figure 2.6.6: Here's the queue and the array after the last call to the enqueue() member function is made.

```
//queueProgram.cpp
#include <iostream>
using namespace std;
void main(){
   Queue *queue = new Queue(8);
   queue->enqueue(10);
   queue->enqueue(20);
   queue->enqueue(30);
   for(int i=0; i<3; i++){
      cout << queue->dequeue() << endl;
   }
}
```

2.6.5 Summary

Queue is a linear data structures extensively used in varied application of computer programming. It is based in First In First Out (FIFO) principle. A front pointer is used for deleting element from the beginning of the stack and a rear pointer is maintained for deleting elements from end of the stack. Enqueue operation is used to insert elements in the stack and dequeue operation is used for deleting an element from the stack. Underflow and overflow conditions are check using IsEmpty and IsFull functions, respectively.

2.6.6 Questions

1. What is a queue?
2. What is the relationship between a queue and its underlying array?
3. Explain how the index of the front and back of the queue is calculated.
4. What is the purpose of the enqueue process?
5. What is the purpose of the dequeue process?
6. Why is the IsFull() member method called?
7. Why is the IsEmpty() member method called?
8. What happens to the data stored on the array when the data is removed from the queue?
9. What is the purpose of setting the default size of the queue?

2.6.7 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.7 **AUTHOR: DHARAM VEER SHARMA**

TYPES OF QUEUES

2.7.1 Objective of the Lesson

In this lesson, we will discuss the various variants of queues including circular, double ended and priority queue. We shall also discuss the various applications of queues.

2.7.2 Introduction

There are different variants of queues for diverse areas of applications. These include circular queue, double ended queue also called dequeue or deque and priority queue as used in operating systems.

2.7.3 Circular Queue

A circular queue is a Queue but a particular implementation of a queue. It is very efficient. It is also quite useful in low level code, because insertion and deletion are totally independant, which means that you don't have to worry about an interrupt handler trying to do an insertion at the same time as your main code is doing a deletion.

Algorithm for Insertion:-

Step-1: If "rear" of the queue is pointing to the last position then go to step-2 or else step-3

Step-2: make the "rear" value as 0

Step-3: increment the "rear" value by one

Step-4:

1. if the "front" points where "rear" is pointing and the queue holds a not NULL value for it, then its a "queue overflow" state, so quit; else go to step-4.2
2. insert the new value for the queue position pointed by the "rear"

Algorithm for deletion:-

Step-1: If the queue is empty then say "empty queue" and quit; else continue

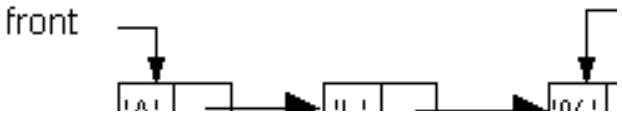
Step-2: Delete the "front" element

Step-3: If the "front" is pointing to the last position of the queue then step-4 else step-5

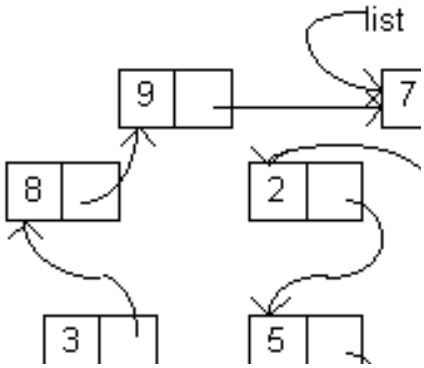
Step-4: Make the "front" point to the first position in the queue and quit

Step-5: Increment the "front" position by one

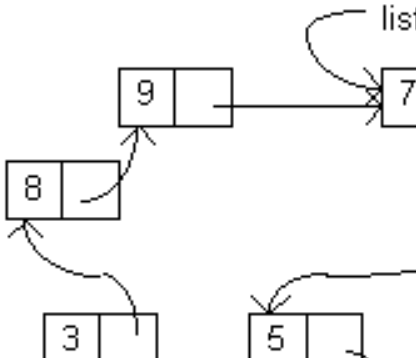
One of the nice applications of pointers is in implementing a circular queue. First let us consider the fact, that we only need one pointer and that is to the last element in the queue. In the example below, the node containing 2 is the head of the list and the node containing 9 is the rear of the list.



To enQueue a node containing 7 the queue would look like this:

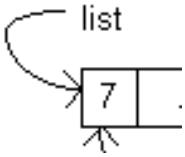


And then if we were to deQueue a node, we would dequeue the node containing the 2 and the result would look like this.



In this implementation, an empty list would be list = NULL.

A list containing one element would look like this. In coding enQueue, care must be taken with the first element.



To get rid of a queue or makeEmpty, the nodes must be disposed so that the memory is free again. Use the myqueue Class, a pointer implementation of a queue, to modify the code to be a circular queue. In your client program, to show that each function works, write a non-destructive function show that will display the contents of the queue.

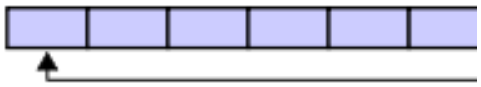
A key advantage of a circular queue is that it has a static size and elements need not be shuffled around when a portion of the queue is used. This means that only new data is written to the queue and the computational cost is independent of the length of the queue.

For example, in implementing a Transmission Control Protocol stack the windows used to hold the data can be circular queues. When the receiver acknowledges the reception of a packet then the amount of data acknowledged is used to advance the start of the queue. This allows the stack to restrict the amount of unacknowledged data by the transmitter. (This is an example of when it is not permitted for the start of the queue to be overwritten.)

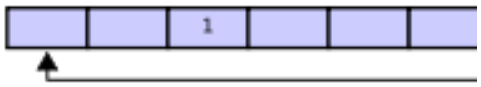
An example that could possibly use an overwriting circular queue is with multimedia. If the queue is used as the bounded queue in the producer-consumer problem then it is probably desired for the producer (e.g., an audio generator) to overwrite old data if the consumer (e.g., the sound card) is unable to momentarily keep up. Another example is the digital waveguide synthesis method which uses circular queues to efficiently simulate the sound of vibrating strings or wind instruments.

The "prized" attribute of a circular queue is that it does not need to have its elements shuffled around when one is consumed. (If a non-circular queue were used then it would be necessary to shift all elements when one is consumed.) In other words, the circular queue is well suited as a FIFO queue while a standard.

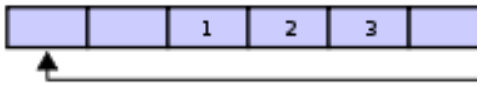
A circular queue first starts empty and of some predefined length. For example, this is a 7-element queue:



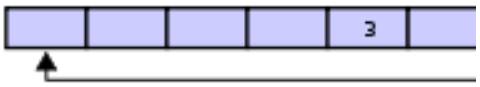
Assume that a 1 is written into the middle of the queue (exact starting location does not matter in a circular queue):



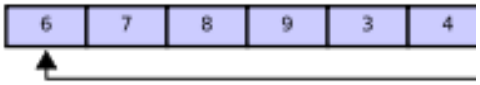
Then assume that two more elements are added — 2 & 3 — which get appended after the 1:



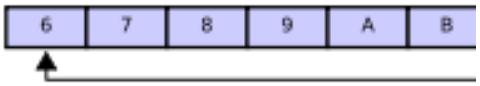
If two elements are then removed from the queue then they come from the end. The two elements removed, in this case, are 1 & 2 leaving the queue with just a 3:



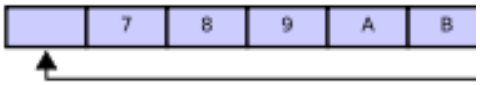
If the queue has 7 elements then it is completely full:



A consequence of the circular queue is that when it is full then a subsequent write is performed then it starts overwriting the oldest data. In this case, two more elements — A & B — are added and they overwrite then 3 & 4:



Alternatively, the routines that manage the queue could easily not allow data to be overwritten and return an error or raise an exception. Whether or not data is overwritten is up to the semantics of the queue routines or the application using the circular queue. Finally, if after overwriting elements two elements are removed then what would be returned is not 3 & 4 but 5 & 6 because A & B overwrote the 3 & the 4 yielding the queue with:



Circular queue mechanics

What is not shown in the example above is the mechanics of how the circular queue is managed.

Start / End Pointers

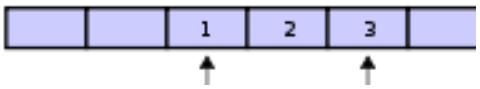
Generally, a circular queue requires three pointers:

- one to the actual queue in memory
- one to point to the start of valid data
- one to point to the end of valid data

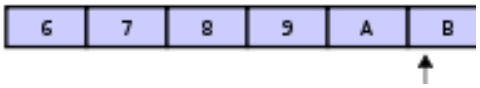
Alternatively, a fixed-length queue with two integers to keep track of indices can be used in languages that do not have pointers.

Taking a couple of examples from above. (While there are numerous ways to label the pointers and exact semantics can vary, this is one way to do it.)

This image shows a partially-full queue:



This image shows a full queue with two elements having been overwritten:



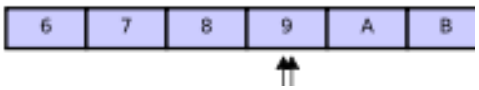
What to note about the second one is that after each element is overwritten then the start pointer is incremented as well.

Difficulties

Full / Empty Queue Distinction

Some small disadvantage of relying on pointers or relative indices of the start and end of data is, that in the case the queue is entirely full, both pointers pointing at the same element:

This is exactly the same situation as when the queue is empty:



To solve this problem there are a number of solutions:

- Always keep one byte open.
- Use a fill count to distinguish the two cases.
- Use read and write counts to get the fill count from.
- Use absolute indices.

Always Keep One Byte Open

This simple solution always keeps one byte unallocated. A full queue has at most (size − 1) bytes. If both pointers are pointing at the same location, the queue is empty.

The advantages are:

- Very simple and robust.
- You need only the two pointers.

The disadvantages are:

- You can never use the entire queue.
- If you cannot read over the queue border, you get a lot of situations where you can only read one element at once.

Use a Fill Count

The second simplest solution is to use a fill count. The fill count is implemented as an additional variable which keeps the number of readable bytes in the queue. This variable has to be increased if the write (end) pointer is moved, and to be decreased if the read (start) pointer is moved. In the situation if both pointers pointing at the same location, you consider the fill count to distinguish if the queue is empty or full.

The advantages are:

- Simple.
- Needs only one additional variable.

The disadvantage is:

- You need to keep track of a third variable. This can require complex logic, especially if you are working with different threads.

Alternately, you can replace the second pointer with the fill count and generate the second pointer as required by incrementing the first pointer by the fill count.

The advantages are:

- Simple.
- No additional variables.

The disadvantage is:

- Additional overhead when generating the write pointer.

Read / Write Counts

Another solution is to keep counts of the number of items written to and read from the circular queue. Both counts are stored in unsigned integer variables with numerical limits larger than the number of items that can be stored and are allowed to wrap freely from their limit back to zero.

The unsigned difference (write_count - read_count) always yields the number of items placed in the queue and not yet retrieved. This can indicate that the queue is empty, partially full, completely full (without waste of a storage location) or in a state of overrun.

The advantage is:

- The source and sink of data can implement independent policies for dealing with a full queue and overrun while adhering to the rule that only the source of data modifies the write count and only the sink of data modifies the read count. This can result in elegant and robust circular queue implementations even in multi-threaded environments.

The disadvantage is:

- You need two additional variables.

2.7.4 Double Ended Queue (de-queues)

This differs from the queue abstract data type or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

- An input-restricted deque is one where deletion can be made from both ends, but input can only be made at one end.
- An output-restricted deque is one where input can be made at both ends, but output can be made from one end only.

Both the basic and most common list types in computing, the queues and stacks can be considered specializations of deques and can be implemented using deques.

A deque (short for double-ended queue—usually pronounced deck) is an abstract list type data structure also called a head-tail linked list, for which elements can be added to or removed from the front (head) or back (tail).

Deque is sometimes written dequeue, but this use is generally deprecated in technical literature or technical writing because dequeue is also a verb meaning "to remove from a queue". Nevertheless, several libraries and some writers, such as Aho, Hopcroft, and Ullman in their textbook Data Structures and Algorithms, spell it dequeue. DEQ and DQ are also used.

This differs from the queue abstract data type or First-In-First-Out List (FIFO), where elements can only be added to one end and removed from the other. This general data class has some possible sub-types:

- An input-restricted deque is one where deletion can be made from both ends, but input can only be made at one end.
- An output-restricted deque is one where input can be made at both ends, but output can be made from one end only.

Both the basic and most common list types in computing, the queues and stacks can be considered specializations of deques, and can be implemented using deques.

Operations

The following operations are possible on a deque:

operation	C++
insert element at back	push_back
insert element at front	push_front
remove last element	pop_back
remove first element	pop_front
examine last element	back
examine first element	front

Implementations

There are at least two common ways to efficiently implement a deque: with a modified dynamic array or with a doubly-linked list. For information about doubly-linked lists, see the linked list article.

Dynamic array implementation

Deques are often implemented using a variant of a dynamic array that can grow from both ends, sometimes called array deques. These array deques have all the properties of a dynamic array, such as constant time random access, good locality of reference and inefficient insertion/removal in the middle, with the addition of amortized constant time insertion/removal at both ends, instead of just one end. Two common implementations include:

- Storing deque contents in a circular queue, and only resizing when the queue becomes completely full. This decreases the frequency of resizings, but requires an expensive branch instruction for indexing.
- Allocating deque contents from the center of the underlying array, and resizing the underlying array when either end is reached. This approach may require more frequent resizings and waste more space, particularly when elements are only inserted at one end.

Complexity

- In a doubly-linked list implementation, the time complexity of all operations is O(1), except for accessing an element in the middle using only its index, which is O(n).
- In a growing array, the amortized time complexity of all operations is O(1), except for removals and inserts into the middle of the array, which are O(n).

2.7.5 Priority Queue

A priority queue is similar to a simple queue in that items are organized in a line and processed sequentially. However, items on a priority queue can jump to the front of the line if they have priority. Priority is a value that is associated with each item placed in the queue. The program processes the queue by scanning the queue for items with high priority. These are processed first regardless of their position in the line. All the other items are then processed sequentially after high priority items are processed.

Unlike a standard queue where items are ordered in terms of who arrived first, a priority queue determines the order of its items by using a form of custom comparer to see which item has the highest priority. Other than the items in a priority queue being ordered by priority it remains the same as a normal queue, you can only access the item at the front of the queue.

A sensible implementation of a priority queue is to use a heap data structure. Using a heap we can look at the first item in the queue by simply returning the item at index 0 within the heap array. A heap provides us with the ability to construct a priority queue where the items with the highest priority are either those with the smallest value or those with the largest.

2.7.6 Application of Queues

An operating system often maintains a FIFO queue of processes that are ready to execute or that are waiting for a particular event to occur. The programmer who creates the operating system can use a Queue ADT to implement this.

Computer systems must often provide a "holding area" for messages between two processes, two programs or even two systems. This holding area is usually called a "buffer" and is often implemented as a FIFO queue. For example, if a large number of mail messages arrive at a mail server at about the same time, the messages are held in a buffer until the mail server can get around to processing them. It processes them in the order they arrived—in "first in, first out" order. (Some mail servers may be designed to handle the messages based on a priority system. In that case, the priority queue would be a more appropriate data structure.)

To demonstrate the use of queues, we look at a simpler problem i.e. identifying palindromes. A palindrome is a string that reads the same forwards as backwards. While we are not sure of their general usefulness, identifying them provides us with a good example for the use of both queues and stacks. Besides, palindromes can be entertaining.

Some famous palindromes are:

- A tribute to Teddy Roosevelt, who orchestrated the creation of the Panama Canal: "A man, a plan, a canal—Panama!"
- Allegedly muttered by Napoleon Bonaparte upon his exile to the island of Elba (although this is hard to believe since Napoleon mostly spoke French!): "Able was I ere, I saw Elba."
- Overheard in a Chinese restaurant: "Won ton? Not now!"
- And possibly the world's first palindrome: "Madam, I'm Adam."
- Followed immediately by one of the world's shorted palindromes: "Eve."

As you can see, the rules for what is a palindrome are somewhat lenient. Typically, we do not worry about punctuation, spaces or matching the case of letters. We

again follow the input/output model we have established for our test drivers-the same model used for the balanced parentheses example in the section on stacks.

An input file holds a separate string on each line. A corresponding output file is created, repeating each of the input lines and stating whether or not it is a palindrome.

This program assumes that no input line is more than 180 characters in length. If an input line is longer than that it is skipped. The names of the input and output files are passed to the program on the command line. Summary statistics are written to an output frame.

The program reads a line of input and checks to see how long it is. If it is too long, it moves on to the next line of input. Otherwise, it creates a new stack and a new queue and it repeatedly pushes each letter from the input line onto the stack and also enqueues it onto the queue. To simplify comparison later, the actual characters pushed and enqueued are the lowercase versions of the characters in the string. When all of the characters of the line have been processed, the program repeatedly pops a letter from the stack and dequeues a letter from the queue. As long as these letters match each other for the entire way through this process, we have a palindrome. Can you see why? Since the queue is a “first in, first out” structure, the letters are returned from the queue in the same order they appear in the string. But the letters taken from the stack, a “last in, first out” structure, are returned in the opposite order from the way they appear in the string. So, we are comparing the letters from the forward view of the string to the letters from the backward view of the string.

2.7.7 Summary

There are different variants of queues for diverse areas of applications. These include circular queue, double ended queue also called dequeue or deque and priority queue as used in operating systems. A circular queue is a Queue but a particular implementation of a queue. It is very efficient.

A double ended queue or dequeue may be:

- An input restricted where deletion can be made from both ends, but input can only be made at one end.
- An output restricted where input can be made at both ends but output can be made from one end only.

A priority queue is similar to a simple queue in that items are organized in a line and processed sequentially.

Queues are used by operating system for process management, by computer for buffering etc.

2.7.8 Questions

1. What are the various types of queues?
2. Define circular queue. What are its properties? Explain.
3. Write the procedure of implementing circular queue.
4. Discuss in detail the concept of double ended queue.
5. What are priority queues? How these are implemented? Explain.

2.7.9 Suggested Readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley.

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.8 **AUTHOR : DR DHARAMVIR SHARMA**

LINKED LIST

2.8.1 Objective of the lesson

In this lesson, we shall discuss meaning of linked list, creation of linked list and various operations like insertion, deletion, traversal and displaying of linked list.

2.8.2 Introduction

A linked list is a data structure that makes it easy to rearrange data without having to move data in memory. Sound a little confusing? If so, picture a classroom of students who are seated in no particular order. A unique number identifies each seat, as shown in Figure 2.8.1. We've also included the relative height of each student, which we'll use in the next exercise.

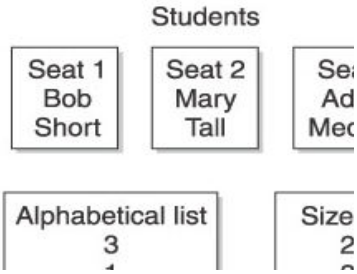


Figure 2.8.1: Students are seated in a classroom in random order.

Let's say that a teacher needs to place students names in alphabetical order so she can easily find a name on the list. One option is to have students change their seats so that Adam sits in seat 1, Bob sits in seat 2 and Mary in seat 3. However, this can be chaotic if there are a lot of students in the class.

Another option is to leave students seated and make a list of seat numbers that corresponds to the alphabetical order of students. The list would look something like this: 3, 1 and 2 as shown in Figure 2.8.1. The student in seat 3 is the first student who appears in alphabetical order followed by the student seated in seat 1 and so on. Notice how this option doesn't disrupt the class.

Suppose you want to rearrange students in size order. There's a pretty good chance that you won't move students about the classroom. Instead, you'd probably create another list of seat numbers that reflect each student's height. Here's the list: 1, 3 and 2 which is illustrated in Figure 2.8.1. The list can be read from bottom to top for the shortest to tallest or vice versa for tallest to shortest.

Once the list is created, the teacher can simply go down the list to see which seat contains the next student. To quiz students in alphabetical order, the teacher would use the alphabetical list to see that the student sitting in seat 3 is alphabetically first, followed by seat 1. The teacher can be tricky and call on the previous student by looking at the list to determine the student's seat.

Programmers call this sort of list a linked list because each item on the list is linked to the previous item and the next item. That is, the seat of the current student is linked to the seat of the previous student and to the seat of the next student by the list.

It is very important to keep the real world in mind as you learn how to use a linked list, otherwise, you'll fall into the trap of thinking that a linked list is an abstract concept that has little use in the real world. Actually, linked lists play a critical role in applications that help companies and governments manage data dynamically.

There are two versions of a linked list, a single link and a double link. A single link list enables a program to move through the list in one direction, which is usually from the front of the list moving to the end of the list. A doubly linked list enables the program to move through the list in both directions. We'll focus on the doubly linked list for most of the examples in this lesson and then discuss the single link list toward the end of the lesson.

Although we've mentioned that an entry in a linked list contains data and pointers to the previous and next entries in the list, this is an over simplification. The data we're talking about is typically a set of data such as customer information. Customer information could be a customer ID, customer first name, customer last name, customer street address, customer city, customer state, customer ZIP and so on. Programmers call this a record. This means that an entry in a linked list may contain several data elements. In our example, however, we'll store only a single value of an integer so that we can focus on the principle of how a linked list works. In reality, you can add as many additional attributes to each node as you need.

Programmers choose linked lists over an array because linked lists can grow or shrink in size during runtime. Another entry can be placed at the end of the last entry on the linked list simply by assigning reference to the new entry in the last entry on the linked list. Likewise, the last entry can be removed from the linked list by simply removing reference to it in the next element of the second-to-last entry on the linked list. This is more efficient than using an array and resizing at runtime.

If you change the size of the array, the operating system tries to increase the array by using memory alongside the array. If this location is unavailable, then the operating system finds another location large enough to hold elements of the array and new array elements. Elements of the array are then copied to the new location.

If you change the size of a linked list, the operating system changes references to the next item on the list, which is fewer steps than changing the size of an array.

2.8.3 Structure of a Linked List

Each entry in a linked list is called a node. Think of a node as an entry that has two subentries. One subentry contains the data, which may be one attribute or many attributes and the other points to the next node. When you enter a new item on a linked list, you allocate the new node and then set the pointer to next nodes.

Programmers create a node in C++ by using either a structure or a class object; our example uses a structure. As you'll recall from your C++ programming course, a structure is a user-defined data type. The following example is a structure used to define a node. Figure 2.8.2 shows a node.

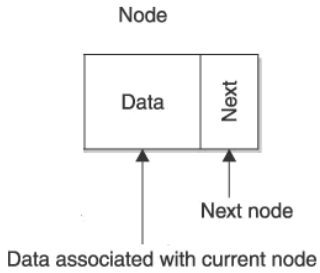


Figure 2.8.2: A node contains reference to the next node and the previous node in the linked list and contains data that is associated with the current node.

```
struct Node{
        int data;
        Node *next;
}*List;
```

The structure might look a bit strange even if you are familiar with structures because this example uses a pointer to the structure itself as one of its attribute. We'll clear up any confusion by taking apart this example. The structure is called Node. The name of the structure creates an instance of the structure similar to how you use a constructor to create an instance of a class and data type.

The first statement of the structure declares an integer that stores the current data of the node. The next statement declares pointers to the next node in the linked list.

2.8.4 Operations on Linked List

The various operations that can be performed on linked list are:

i. Creation (CreateList())

ii. Insertion of a node

a. Insertion at beginning (InsertAtBeg())

b. Insertion at end (InsertAtEnd())

c. Insertion at a position in the list (InsertAtPos())

d. Insertion after a specific value (InsertAfterVal())

iii. Deletion of a node

a. Deletion from beginning (DeleteFromBeg())

b. Deletion from end (DeleteFromEnd())

iv. Traversal of linked list or Displaying elements of a linked list (Disp())

v. Reverse nodes of the list (Reverse())

vi. Destroy list (Destroy())

2.8.4.1 The Linked List Class

Programmers use a LinkedList class to create and manage a linked list. C++ programmers define their own LinkedList class. For now, we'll focus on defining a LinkedList class in C++.

The LinkedList class definition consists of two data members and eleven function members, as shown in the example in this section. The two data members are: **data** which stores the actual data of the node and the pointer, **next** which references the next node on the linked list if it exists or NULL.

The twelve member functions manipulate the linked list. The first member function is the constructor of the LinkedList class and is called when an instance of the class is declared. Following the constructor is the destructor. When you return memory to the operating system by using the delete operator, the destructor is called. If you don't call the delete operator, then the destructor never gets called and the application causes a memory leak.

- The CreateList(int val) member function create a new list with the value of the first node supplied as argument to the function.
- The InsertAtBeg(int val) function inserts a new node in the beginning of the list making it the first node of the list.
- The InsertAtEnd(int val) function inserts a new node at the end of the list.
- The InsertAtPos(int pos, int val) function inserts a new node at the specified position in the list.

- The InsertAfterVal(int aval, int val) function inserts a new node after a specific value already existing in the list.
- The int DeleteFromBeg() function deletes the first node of the list.
- The int DeleteFromEnd() function deletes the last node of the list.
- The void Disp() functions displays the values of all nodes of the list.
- The void Destroy() function destroys the list.

The LinkedList class specification is defined in the header file, and the implementation is defined in the source file. We'll take a closer look at the implementation of these member functions in the next few sections of this lesson.

```
class LinkedList{
        struct Node{
                int data;
                Node *next;
        }*List;
        public:
                LinkedList(){NODE = NULL;}
                ~LinkedList(){Destroy();}
                void CreateList(int val);
                void InsertAtBeg(int val);
                void InsertAtEnd(int val);
                void InsertAtPos(int pos, int val);
                void InsertAfterVal(int aval, int val);
                int DeleteFromBeg();
                int DeleteFromEnd();
                void Reverse();
                void Disp();
                void Destroy();
};
```

2.8.4.2 Creation of a Linked List

The LinkedList constructor is a member function that is called when an instance of the LinkedList is declared. The purpose of the constructor in the linked list example is to initialize the List pointer, also known as header node, as shown in the following definition. The List pointer is assigned a NULL value, which is used by member functions to determine if the linked list is empty. You'll see how this is done in the next section.

```
LinkedList(){
  List = NULL;
}
```

The destructor is a member function called when the instance of the LinkedList class is deleted using the delete operator. In the example shown next, the destructor contains one statement that calls the Destroy() member function.

The Destroy() member function deletes the contents of the linked list but does not delete the linked list itself. That is, it removes all the nodes from the linked list. The Destroy() also resets the List pointer to NULL, signifying the linked list is empty of nodes. The destructor is responsible for deallocating all the memory that was allocated for the linked list. In this case, it would be all the nodes.

You might be wondering why we defined two member functions to perform basically the same task. We do so to enable the programmer to empty the linked list. This way you can reset the contents of the linked list without destroying the instance of the LinkedList class.

```
~LinkedList(){
   Destroy();
}
```

2.8.4.3 Insertion in a Linked List

Insertion in a linked list can be made by different ways. We shall consider four methods of inserting a node in the linked list.

i. **Insert at beginning:** The InsertAtBeg() member function places a new node at beginning of the list. If the list is empty the next pointer the newly inserted node points to NULL else the next node points to node of the list which was first node before insertion.

```
void LinkedList::InsertAtBeg(int val)
```

```
{
    Node *p;
    p = new Node;    //Create a new node
    p->data = val;
    p->next = List;  //Make it first node of the list
    List = p;        //Point List to the new node
}
```

An argument val is passed to the function which is the data for the new node to m be inserted. The function, first, creates a new node and points the next pointer of the new node to the beginning of the List and then makes it the first node of the list.

ii. **Insert at end:** The InsertAtEnd() member function insert a node at the end of the linked list. There are several steps that must be performed in order to add the node to the end of the list. These are shown in the following definition of the member function:

```
void LinkedList::InsertAtEnd(int val)
{
    Node *p, *Start;
    Start = List;
    p = new Node;
    p->data = val;
    p->next = NULL;
    if (Start == NULL)
            List = p;
    else
    {
            while (Start->next != NULL)
                    Start = Start->next;
            Start->next = p;
```

```
    }
}
```

The InsertAtEnd() member function requires one argument called val, which is the current data for the node. The first statement in the InsertAtEnd() member function declares two instances of the List structure, one (p) for creating the new node and the other (Start) for moving to the end of the List.

Once the new node is created, the InsertAtEnd() member function positions the new node in the linked list. First, it determines if the linked list is empty by comparing the List pointer to NULL. As you'll recall, the List pointer is assigned a NULL when an instance of the LinkedList class is declared and when the Destroy() member function removes all the nodes from the list. If the linked list is empty (checked by if condition), then the new node is assigned to the List pointer. This means that the linked list contains one node after the InsertAtEnd() member function is called, which is the new node. However, if there is at least one node on the linked list, then a little shifting of pointers must be performed. The while statement performs the required shifting and the new node is then assigned to the next pointer, making the new node the last node on the linked list. This can be a little confusing, so take a look at Figure 2.8.3. Figure 2.8.3 shows nodes of the linked list. Assume that the linked list has two nodes before the new node is appended to the list. This is represented in the top block.

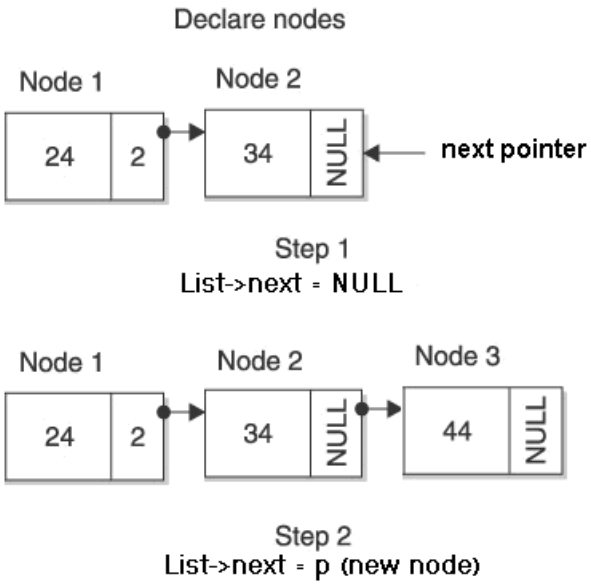


Figure 2.8.3: The appendNode() member function changes what nodes are pointed to in the linked list.

The first step moves the pointer to the end of the list, which is shown in the second block of memory in Figure 2.8.3. The second step assigns the memory address of the new node to the next member of the last node. This links both nodes.

iii. **Insert at position:** The InsertAtPos() function insert an element at the specified position in the list. If the specified position is greater than the number of nodes in the list then the node is inserted at the end.

```
void List::InsertAtPos(int pos, int val)
{
    int count = 1;
    Node *p, *q = List;
    p = new Node;
    p->data = val;

    while (q->next != NULL && count < pos-1)
    {
        count++;
        q = q->next;
    }
    p->next = q->next;
    q->next = p;
}
```

The pointer is first moved to the specified position using the while loop. If the number of nodes in the list is less than the specified position then the node is inserted at the end of the List.

iv. **Insert after value:** The InsertAfterVal() function inserts a new node after a node having the specified value.

```
void List::InsertAfterVal(int aval, int val)
{
    Node *p, *q = List;
```

```
        p = new Node;
        p->data = val;
        while (q != NULL && q->data != aval)
                q = q->next;
        p->next = q->next;
        q->next = p;
    }
```

The function is just like the previous function, the only difference is that the node is inserted after a specified value instead of position. If the specified value does not exist in the list then the new node is inserted at the end.

2.8.4.3 Deletion from beginning of the List

We have specified two different function for deleting a node from the List.

i. **Delete node from beginning:** The function DeleteFromBeg() deletes the first node of the List and the second node becomes the first node if it exists otherwise the list becomes empty.

```
int List::DeleteFromBeg()
{
    int val;
    Node *p = List;
    List = List->next;
    val = p->data;
    delete p;
    return val;
}
```

ii. **Deletion from end of the list:** This is an interesting situation. You will never cut the stem of the tree while sitting on it or break the stair while standing on it. Similarly for deleting a node from the end of the list we need to reach the last but one node of the list. The function DeleteFromEnd() does the same.

```
int List::DeleteFromEnd()
{
    int val;
    Node *p,*q = List;

    while (q->next->next != NULL)
        q = q->next;
    p = q->next;
    q->next = NULL;
    val = p->data;
    delete p;
    return val;
}
```

The above function reaches the last but one node of the list and from there deletes the last node.

2.8.4.5 Displaying elements of a Linked List

The Disp() member function displays each node of the linked list, beginning with the node at the beginning of the list and ending with the node at the end of the list. This is shown in the next example:

```
void List::Disp()
{
        Node *p = List;
        while (p != NULL)
        {
                cout << p->data << "\t";
                p = p->next;
        }
        cout << endl;
}
```

The Disp() member function begins by declaring a pointer p to the List. Before attempting to display data assigned to the node, Disp() determines if there is a node at the next position of the linked list. It does so by determining if the node pointed to by the p pointer is NULL. If so, the linked list is empty and there is nothing to display. If not, the member function proceeds and displays the data assigned to the node of the linked list.

The Disp() member function uses the node's next member to assign the pointer to the next node to the p pointer. The process continues by first determining if the node isn't NULL before displaying the data assigned to the node.

This process ends after the node at the end of the linked list is displayed because the next member of the node at the end of the list is NULL.

2.8.4.5 Destroying the Linked List

The Destroy() member function removes nodes from the linked list without removing the linked list itself, as shown in the following example. Each node is declared dynamically using the new operator. This enables you to remove the node by using the delete operator.

```
void List::Destroy()
{
        Node *p;
        while (List != NULL)
        {
                p = List;
                List = List->next;
                delete p;
        }
List = NULL;
}
```

The Destroy() member function begins by declaring a temporary pointer that is assigned the pointer to the node that is to be deleted and then the List points to the next node. However, before the node is removed, the member function determines if there is a node at the current position of the linked list by testing whether the List pointer is NULL. If so, then the Destroy() member function assumes there are no

nodes on the linked list. If the List pointer isn't NULL, then the member function proceeds to delete the node.

This process continues until all the nodes are removed from the linked list. The final step in the destroyList() member function is to assign NULL values to the front and back of the linked list, which indicates that the linked list is empty of any nodes.

2.8.5 Implementing Linked Lists in C++

Now that you know the parts of a linked list and how to create and manipulate the linked list using a class, we'll put those parts together and create a real-world C++ application that uses a linked list.

Professional programmers organized a linked list C++ application into three files. The first file is the header file that contains the definition of the NODE structure and the LinkedList class definition. The second file is a source code file containing the implementation of member functions of the LinkedList class. The last file is the application file that contains code that creates and uses the LinkedList class.

Let's begin with the header file. LinkedList.h, shown in the code in this section, is the header file for the C++ linked list example. This file contains the definition of the LinkedList class, which programmers called a class specification.

You'll notice that the LinkedList class definition does not contain the implementation of member functions. Instead, it contains prototypes of member functions that are implemented in the source file. Keeping the specifications and implementation in separate header and source files is common practice. Parts of the program that use the class only care about the interface functions defined in the header file; they don't care about the implementation. This also allows you to precompile your source code into library modules so the users of this class need only the headers and modules.

```
//LinkedList.h
#include <iostream.h>
        class LinkedList{
                struct Node{
                        int data;
                        Node *next;
                }*List;
```

```
        public:
                LinkedList(){List = NULL;}
                ~LinkedList(){Destroy();}
                void CreateList(int val);
                void InsertAtBeg(int val);
                void InsertAtEnd(int val);
                void InsertAtPos(int pos, int val);
                void InsertAfterVal(int aval, int val);
                int DeleteFromBeg();
                int DeleteFromEnd();
                void Disp();
                void Destroy();
};
```

Definitions of member functions for the LinkedList class are contained in the LinkedList.cpp file as shown in the next code in this section. The file begins with a preprocessor statement that tells the preprocessor to reference the contents of the LinkedList.h file during preprocessing. The LinkedList.h file contains the LinkedList class definition, which is required to resolve statements in the LinkedList.cpp that refer to the class.

Each member function definition in this example is practically the same definition as those discussed in the last several sections of this lesson. The only exception is that reference is made to the LinkedList class in the name of each member function definition. This associates each definition with the LinkedList class for the compiler.

```
//LinkedList.cpp
#include "LinkedList.h"
        void LinkedList::CreateList(int val)
        {
                Node *p;
                p = new Node;
                p->data = val;
```

```
        p->next = NULL;
        List = p;
}
void LinkedList::InsertAtBeg(int val)
{
        Node *p;
        p = new Node;
        p->data = val;
        p->next = List;
        List = p;
}
void LinkedList::InsertAtEnd(int val)
{
        Node *p, *Start;
        Start = List;
        p = new Node;
        p->data = val;
        p->next = NULL;
        if (Start == NULL)
                List = p;
        else
        {
                while (Start->next != NULL)
                        Start = Start->next;
                Start->next = p;
}
}
void List::InsertAtPos(int pos, int val)
```

```
        {
                int count = 1;
                Node *p, *q = List;
                p = new Node;
                p->data = val;

                while (q->next != NULL && count < pos-1)
                {
                        count++;
                        q = q->next;
                }
                p->next = q->next;
                q->next = p;
        }
        void List::InsertAfterVal(int aval, int val)
        {
                Node *p, *q = List;
                p = new Node;
                p->data = val;
                while (q != NULL && q->data != aval)
                        q = q->next;
                p->next = q->next;
                q->next = p;
}
        int List::DeleteFromBeg()
        {
                int val;
                Node *p = List;
```

```
        List = List->next;
        val = p->data;
        delete p;
        return val;
}
int List::DeleteFromEnd()
{
        int val;
        Node *p,*q = List;

        while (q->next->next != NULL)
                q = q->next;
        p = q->next;
        q->next = NULL;
        val = p->data;
        delete p;
        return val;
}
void List::Disp()
{
        Node *p = List;
        while (p != NULL)
        {
                cout << p->data << "\t";
                p = p->next;
        }
        cout << endl;
}
```

```
        void List::Destroy()
        {
                Node *p;
                while (List != NULL)
                {
                        p = List;
                        List = List->next;
                        delete p;
                }
        List = NULL;
}
```

The last file is the C++ application that uses the linked list. We call the file LinkedListDemo.cpp, which is shown next. It is amazing that the application itself is so small when compared to all the code used to define the NODE structure and the LinkedList class.

```
//LinkedListDemo.cpp
#include <iostream>
using namespace std;
        void main()
        {
                LinkedList list;
                clrscr();
                list.CreateList(10);
                list.InsertAtBeg(5);
                list.InsertAtEnd(20);
                list.InsertAtPos(3,15);
                list.InsertAfterVal(15,18);
                list.Disp();
                cout << "Element deleted is -> " << list.DeleteFromBeg() << endl;
```

```
        cout << "Element deleted is -> " << list.DeleteFromEnd() <<
endl;
        cout << "Final list if -> \n";
        list.Disp();
        list.Destroy();
}
```

The application begins by declaring an instance of the LinkedList class. As you recall from earlier in this lesson, the constructor initializes the front and back pointers. Next, the CreateList(), InsertAtBeg(), InsertAtEnd(), InsertAfterPos() and InsertAfterVal() member functions are called. Next the Disp() function display the values of nodes of the list, which is:

5 10 15 18 20

The next two statements delete nodes from the list. First DeleteFromBeg() function is called which deleted the first node of the list (node with value 5), then DeleteFromEnd() function is called which deletes the last node of the list (node with value 20). Then the Disp() function again displays the nodes of the list, which are:

10 15 18

Finally the list is destroyed by calling Destroy() function, which successively deleted all nodes of the list.

2.8.6 Summary

Linked list is a linear data structure, in which each node is a combination of two value, one for storing the data of the node and the other for pointing to the next node if there is any or else NULL. A linked list is a dynamic data structure in which nodes can be inserted or deleted during execution of the program. Size of the linked list is constrained by the amount of memory available. Insertion in the list can be done at any point. If a node is to be inserted within a linked list then no shifting of other nodes is required.

2.8.7 Questions

1. What is a linked list?
2. What is the benefit of using a linked list?
3. What is a node?
4. What are the elements of a node?

5. What advantage does a linked list have over an array?
6. Can a node reference more than one data element?
7. How can a node be inserted in the middle of a linked list?
8. Create a function which inserts nodes in ascending order in the list.

2.8.8 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley.

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.9 **Author : Dr Dharamvir Sharma**

TYPES OF LINKED LIST

2.9.1 Objective of the lesson

This lesson begins with three new implementations of reference based lists i.e. circular linked lists, doubly linked lists and lists with headers and trailers. We then introduce an array-based approach to implementing a linked list. This implementation is widely used in operating systems software.

2.9.2 Introduction

There are many variation of linked list but the most common of these are circular linked list and doubly linked list. Linked list can be implemented using arrays as well.

2.9.3 Circular Linked List

The linked lists that we implemented are characterized by a linear (linelike) relationship between the elements. Each element (except the first one) has a unique predecessor and each element (except the last one) has a unique successor. Let's consider a small change to our linked list approach and see how much it would affect our implementation and use of the Sorted List ADT. Suppose we change the linear list slightly, making the next reference of the last node point back to the first node instead of containing null (Figure 2.9.1). Now our list is a circular linked list rather than a linear linked list. We can start at any node in the list and traverse the whole list. Of course, we must now ensure that all of our list operations maintain this new property of the list that after the execution of any list operation, the last node continues to point to the front node. A quick consideration of each of the operations should convince us that we could continue to efficiently support all of them except when an operation changes the first element on the list. Consider, for example, if we try to delete the first element. Our previous delete approach would simply change the list reference to point to the second element on the list, effectively removing the first element. Now, however, we must also update the reference in the last element on the list, so that it points to the new first element. The only way to do that is to traverse the entire list to obtain access to the last element and then make the change. A similar problem arises if we insert an item into the front of the list. Circular linked list is a list in which every node has a successor the "last" element is succeeded by the "first" element.

Figure 2.9.1 A circular linked list

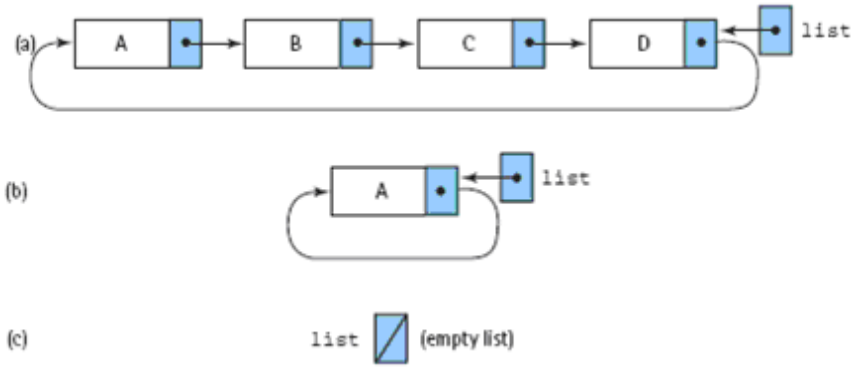


Figure 2.9.2 Circular linked lists with the external pointer pointing to the rear element

Inserting and deleting elements at the front of a list might be a common operation for some applications. Our linear linked list approach supported these operations very efficiently, but our circular linked list approach does not. We can fix this problem by letting our list reference point to the last element in the list rather than the first; now we have direct access to both the first and the last elements in the list (See Figure 2.9.2 where list.info references the information in the last node and list.next.info references the information in the first node.)

Deleting from a Circular List

We can use the same basic approach to deleting an element from a circular list as we used for a linear list. First, find the element that matches the targeted item and then delete it. To delete it we unlink it from the chain of elements by setting the next reference of the element previous to the identified element to reference the element after the identified element. Thus, when we delete an element we need a reference to the element that precedes it. Recall the “trick” we used for the linear list delete method, where we always looked at the element in the position after our current location. That way, when we found the element to delete, location held a reference to the previous node and we could “jump over” the node to be deleted with the statement

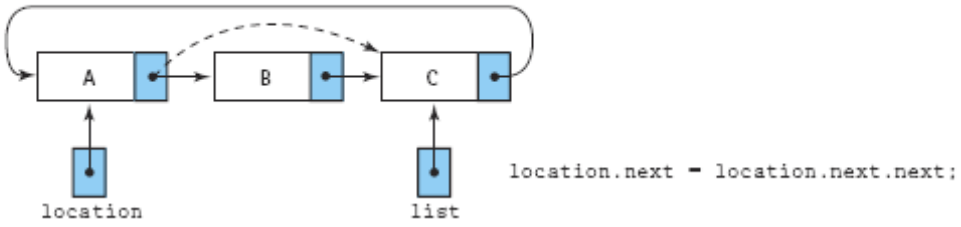
```
location.next = location.next.next;
```

In fact, using our trick with the circular list works very nicely. Since the original value of location is the node at the end of the list, the first information that we check is actually associated with the first node on the list. As with the linear approach, we are guaranteed to find our targeted element. When we do, location is

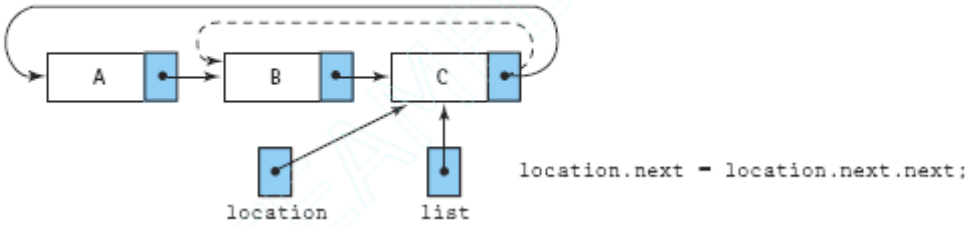
referencing its predecessor on the list. To remove the targeted element from the list, we simply reset location next as described above. We set it to jump over the node we are deleting. That works for the general case, at least (see Figure 2.9.3a).

However, the primary reason that there was a special case was that the overall list reference pointed to the first list element and had to be updated if that element was deleted. In the circular version the overall list reference points to the last list element, so it is very possible that deleting the first element is not a special case. Figure 2.9.3(b) shows that guess to be correct. However, deleting the only node in a circular list is a special case, as we see in Figure 2.9.3(c). The reference to the list must be set to null to indicate that the list is now empty. We can detect this situation by checking, at the start of the method, whether location is equal to location.next. If it is, since we know from the method preconditions that the item we are deleting is on the list, in the case of the single element list we can simply delete it immediately. It must be the element that we wish to delete. We delete it by setting the list reference to null.

(a) The general case (delete B)



(b) Special case (?): deleting the smallest item (delete A)



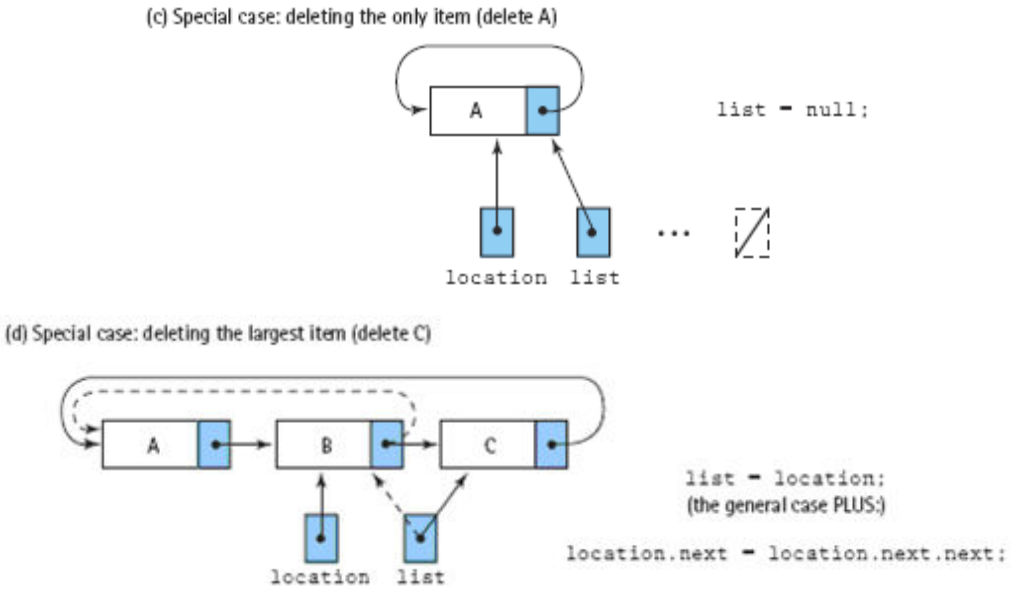


Figure 2.9.3 Deleting from a circular linked list

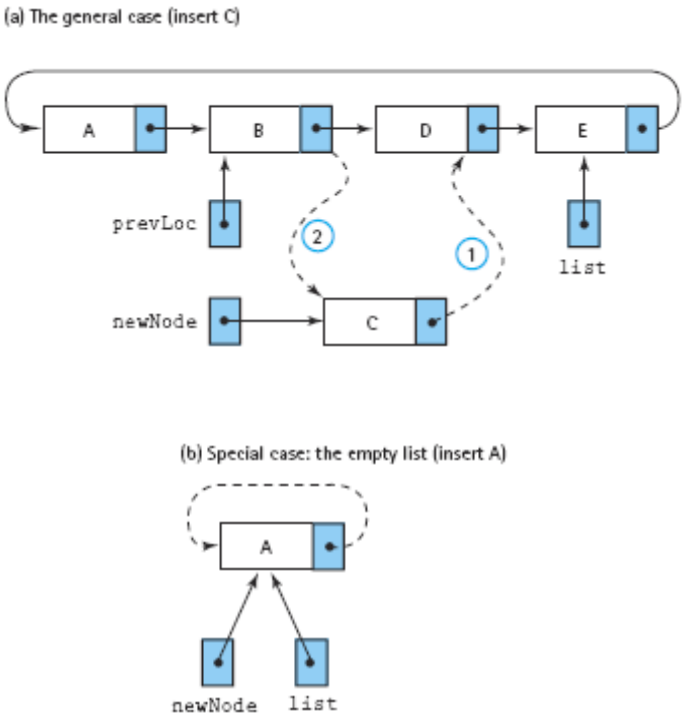
We might also guess that deleting the largest list element (the last node) from a circular list is a special case. After all, our reference to the list points to the last element, so if we delete it we must change our reference. As Figure 2.9.3(d) illustrates, when we delete the last node, we first update the overall list reference to point to the preceding element. We can detect this situation by checking whether location.next equals list after the search phase.

The insert Method

The algorithm to insert an element into a circular linked list is also similar to its linear list counterpart. Essentially, we find the insertion location by performing a search and insert the new item by rearranging some references. To do the insertion we need to have access to both the node preceding the insertion point and the node following the insertion point. And we need to handle special cases carefully. The task of creating a new node is the same as for the linear list. We allocate space for the node using the new operator and then store a copy of item into newNode.info.

The next task is one that we are used to by now. We search through the list maintaining two references, location and prevLoc, until we find an element larger than item or reach the end of the list. The new node is linked into the list immediately after prevLoc. To put the new element into the list we store location into newNode.next and newNode into prevLoc.next.

The general case is illustrated in Figure 2.9.4(a). What are the special cases? First, we have the case of inserting the first element into an empty list. In this case, we want to make list point to the new node and to make the new node point to itself (Figure 2.9.4b). We handle this special case first, before doing any other processing. In the insertion algorithm for the linear linked list we also had a special case when the new element key was smaller than any other key in the list. Because the new node became the first node in the list, we had to change the reference to point to the new node. The reference to a circular list, however, doesn't point to the first node in the list—it points to the last node. Therefore, inserting the smallest list element is not a special case for a circular linked list (Figure 2.9.4c). However, inserting the largest list element at the end of the list is a special case. In addition to linking the node to its predecessor (previously the last list node) and its successor (the first list node), we must modify the list reference to point to newNode—the new last node in the circular list (Figure 2.9.4d).



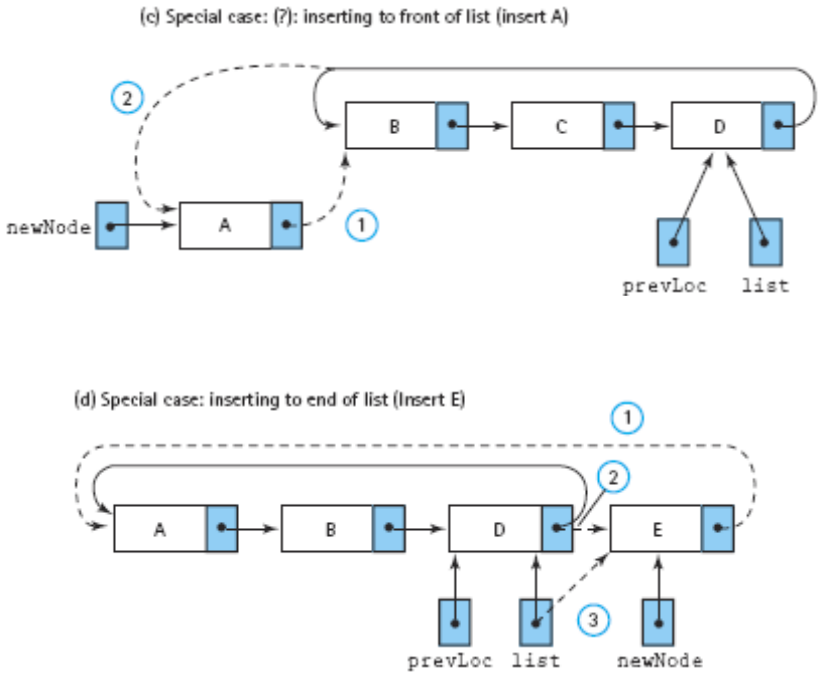


Figure 2.9.4 Inserting into a circular linked list

The statements to link the new node to the end of the list are the same as the general case, plus the assignment of the reference list. Rather than checking for this special case before the search, we can treat it together with the general case. We search for the insertion place and link in the new node. Then, if we detect that we have added the new node to the end of the list, we reassign list to point to the new node. To detect this condition, we compare item to list.info.

insert (item)

Create a node for the new list element. Find the place where the new element belongs. Put the new element into the list Increment the number of items

Circular Versus Linear

Studying circular linked lists provided good practice with using references and self-referential structures. You may have noticed that the only operation that is simpler for the circular approach, as compared to the linear approach, is getNextItem; that minimal advantage is counterbalanced by a more complicated reset operation. Why then might we want to use a circular, rather than linear, linked list? Circular lists are good for applications that require access to both ends of the list. Our Circular

Sorted Linked List class could be used as the basis for other classes that include operations that can take advantage of the new implementation. Perhaps we need a "maximum" operation that returns the largest list element; with the circular approach we have easy access to the largest element (through list). Or suppose we need an operation in Between that returns a boolean value indicating whether a parameter item is "in between" the largest and smallest element of the list; as just mentioned, with the circular approach we have easy access to the largest element (through list) and we also have easy access to the smallest element (through list.next). Therefore, with the circular list, we could implement inBetween in O(1), whereas with our linear approach it would take O(N).

In addition, it is common for the data we want to add to a sorted list to already be in order. Some times people manually sort raw data before turning it over to a data entry clerk. Data produced by other programs are often in sorted order. Given a Sorted List ADT and sorted input data, we always insert at the end of the list, the most expensive place to insert in terms of machine time. It is ironic that the work done manually to order the data now results in maximum insertion times. A circular list with the list reference to the end of the list, as developed in this section, can be designed to avoid this execution overhead. You may have realized that many of the benefits described here for circular lists could also be obtained by using the linear linked list defined in lesson 6 augmented with a reference to the last element of the list. This is yet another list variation as with the circular list, this variation requires changes to some of the linear list methods. We ask you to explore this variation in the exercises. The existence of many list variations is another reason for studying circular lists. It helps us understand how small changes in the structure underlying an ADT can require many subtle changes in the implementations of the ADT operations.

2.9.4 Doubly Linked List

We have discussed using circular linked lists to enable us to reach any node in the list from any starting point. Although this structure has advantages over a simple linear linked list for some applications, it is still too limited for others. Suppose we want to be able to delete a particular node in a list, given only a reference to that node. This task involves changing the next reference of the node preceding the targeted node. However, given only a reference to a node, it is not easy to access its predecessor in the list.

Another task that is difficult to perform on a linear linked list (or even a circular linked list) is traversing the list in reverse. For instance, suppose we have a list of student records, sorted by grade point average (GPA) from lowest to highest. The

Dean of Students might want a printout of the students' records, sorted from highest to lowest, to use in preparing the Dean's List. Consider the Real Estate application where the user can step through a list of house information, viewing the information house by house on the screen, by pressing a "next" button. Suppose the user requests an enhancement to the interface—the idea is to include a "previous" button so that the user can browse through the houses in either direction.

In cases like these, where we need to be able to access the node that precedes a given node, a doubly linked list is useful. In a doubly linked list, the nodes are linked in both directions. Each node of a doubly linked list contains three parts:

- info: the data stored in the node
- next: the reference to the following node
- back: the reference to the preceding node

A linear doubly linked list is pictured in Figure 2.9.5. Note that the back reference of the first node, as well as the next reference of the last node, contains a null. The following definition might be used to declare the nodes in such a list:

Doubly linked list A linked list in which each node is linked to both its successor and its predecessor

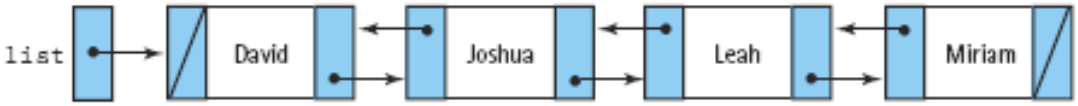


Figure 2.9.5 A linear doubly linked list

The Insert and Delete Operations

Using the definition of DLListNode, let's discuss the corresponding insert and delete methods. The first step for both is to find the location to do the insertion or deletion. This step was complicated in the singly linked list situation by the need to hold onto a reference to the previous location during the search. That is why we created our inchworm search approach. That approach is no longer needed; instead, we can get the predecessor to any node through its back reference. This means we can revert to the simpler search approaches used with our array-based lists.

Although our search phase is simpler, the algorithms for the insertion and deletion operations on a doubly linked list are somewhat more complicated than for a singly

linked list. The reason is clear: There are more references to keep track of in a doubly linked list.

For example, consider insert. To link a new node newNode, after a given node referenced by prevLoc, in a singly linked list, we need to change two references: newNode.next and prevLoc.next (see Figure 2.9.6a). The same operation on a doubly linked list requires four reference changes (see Figure 2.9.6b).

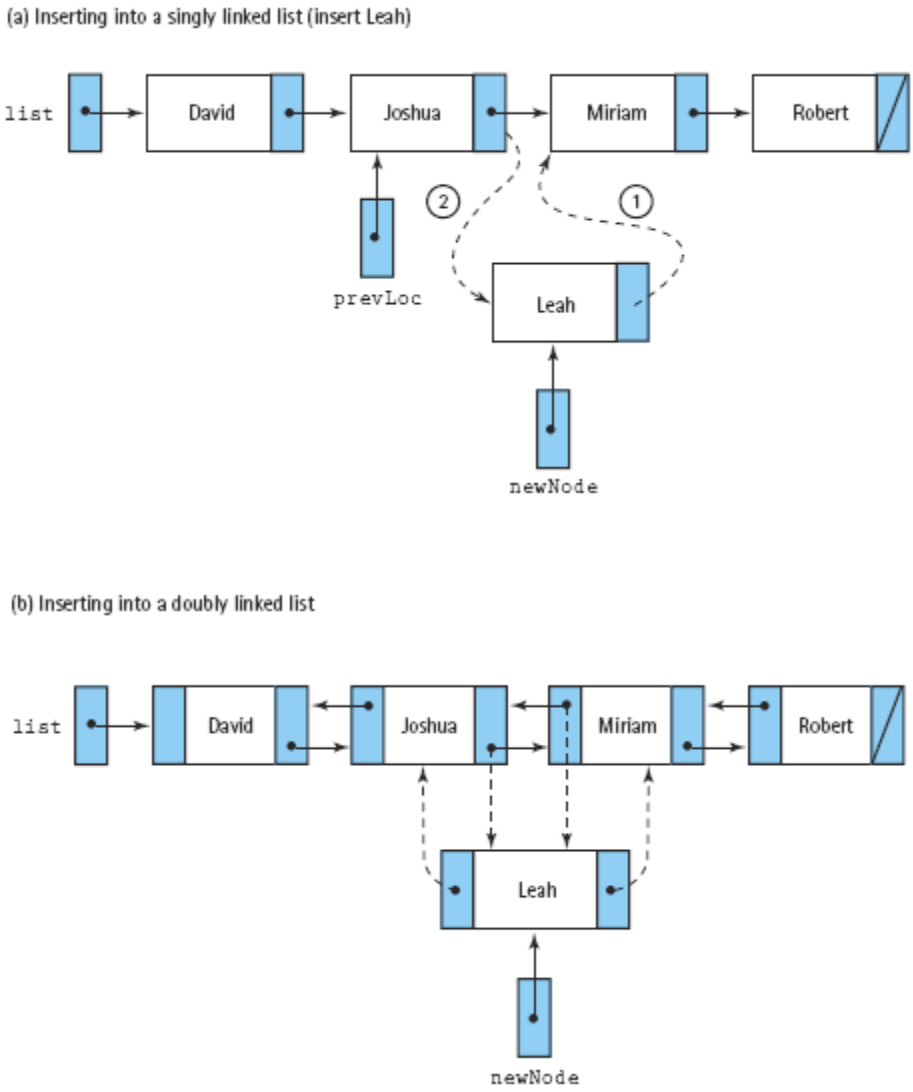


Figure 2.9.6 Insertions into single and doubly linked lists

To insert a new node we allocate space for the new node and search the list to find the insertion point. The result of our search is that location references the node that should follow the new node. Now we are ready to link the new node into the

list. Because of the complexity of the operation, it is important to be careful about the order in which you change the references. For instance, when inserting the new node before location, if we change the reference in location.back first, we lose our reference to the node that is to precede the new node. The correct order for the reference changes is illustrated in Figure 2.9.7. The corresponding code would be

```
newNode.back = location.back;

newNode.next = location;

location.back.next = newNode;

location.back = newNode;
```

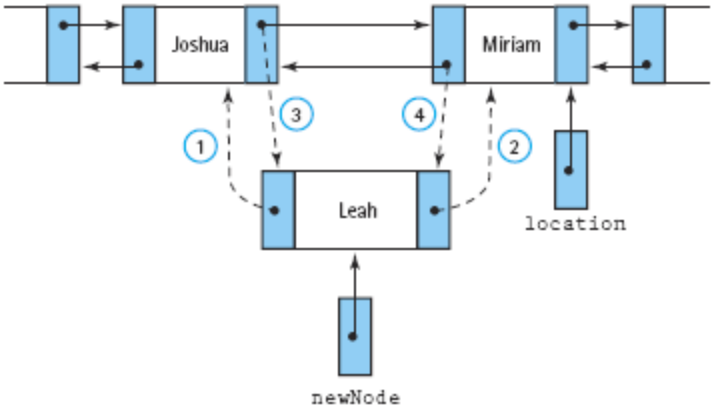


Figure 2.9.7 Inserting into a doubly linked list

We do have to be careful about inserting into an empty list, as it is a special case. Now let's consider the delete method. One of the useful features of a doubly linked list is that we don't need a reference to a node's predecessor in order to delete the node. Through the back reference, we can alter the next variable of the preceding node to make it jump over the unwanted node. Then we make the back reference of the succeeding node point to the preceding node. This operation is pictured in figure 2.9.8. We do, however, have to be careful about the end cases.

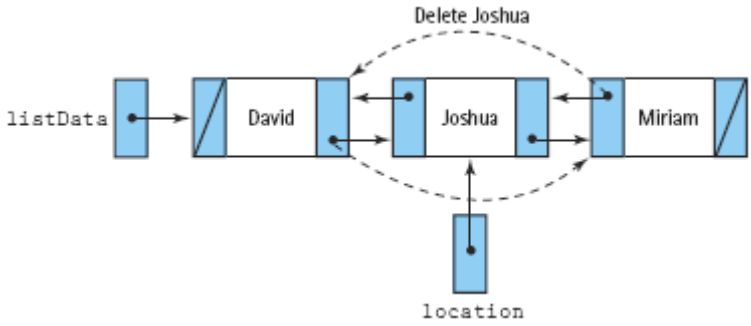


Figure 2.9.8 Deleting from a doubly linked list

If location.back is null, we are deleting the first node; if location.next is null, we are deleting the last node. If both location.back and location.next are null, we are deleting the only node. We leave the complete coding of the insert and delete methods for the doubly linked list as an exercise.

The List Framework

Before leaving the topic of doubly linked lists we should address the question of how they fit into our list framework. Although a doubly linked list is a form of linked list, it is based on a different underlying logical structure than our other linked lists. The relationship among its list elements is different from the case of the singly linked list.

Therefore, it does not make sense to have it extend the abstract LinkedList class as we did for the other implementations. Instead, there are several other viable options. We could create a new class, perhaps called DoublyLinkedList, and require it to implement our current ListInterface interface. In this case, we would probably want to add some additional methods to DoublyLinkedList that are not required by the interface. For example, we could add a getPreviousItem method; otherwise, what is the benefit of having the double links?

Alternately, we could create a new interface, perhaps called TwoWayListInterface that defines what we expect of lists that can be traversed in two directions. Then we could create a class based on doubly linked lists that implements the new interface. The new interface would probably require all of the operations that are part of our current ListInterface interface, plus a few more. Certainly, we would add the getPreviousItem operation. Of course, a doubly linked list is not the only possible way to implement such an interface. An array-based approach would also work well.

The fact that the proposed new interface would require all of the operations of our current ListInterface interface raises another possible approach. Java supports the inheritance of interfaces. (In fact, the language supports multiple inheritance of interfaces, so that a single interface can extend any number of other interfaces.) A good approach would be to define a new interface that extends ListInterface and adds a getPreviousItem method.

Single Linked List vs. Doubly Linked List

Programmers call this a doubly linked list or bidirectional because each node contains reference to the previous and next node on the linked list. This enables the programmer to traverse the linked list in both directions by referencing the previous and next nodes. The node can be transformed into a single linked list (Figure 2.9.9) by only having one pointer in the structure that contains the address of the next node. Typically, a node in a single linked list references the next node and not the previous node, although nothing stops you from creating a backward reference by using only the previous node reference.

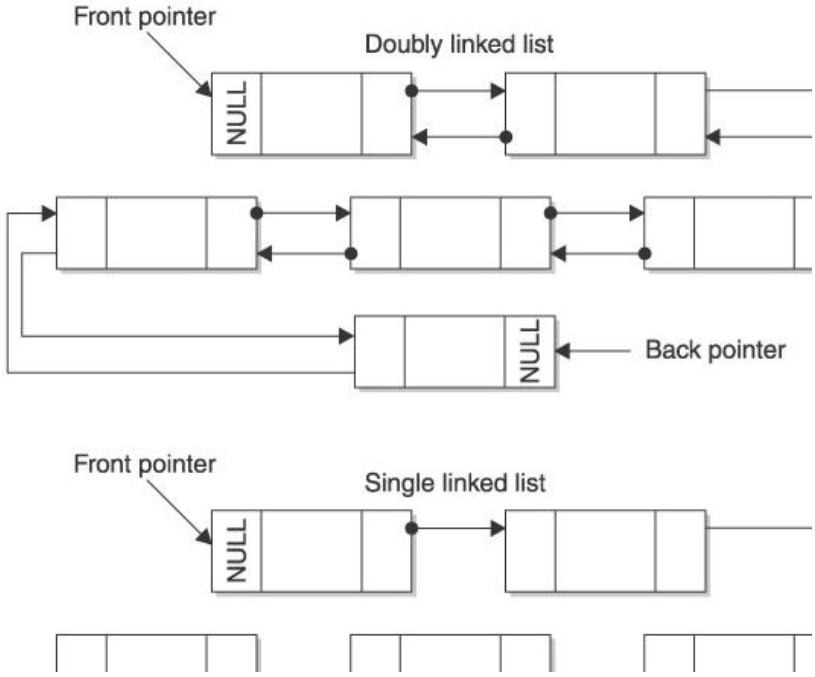


Figure 2.9.9 A doubly linked list contains next and previous members, and a single linked list contains only a next member.

The following example is nearly the same as the previous example except this is a single direction node. You'll notice that reference to the previous node is missing.

This means a programmer can only move down the linked list and not in both directions.

2.9.5 Linked List with Headers and Trailers

In writing the insert and delete algorithms for all implementations of linked lists, we see that special cases arise when we are dealing with the first node or the last node. One way to simplify these algorithms is to make sure that we never insert or delete at the ends of the list.

How can this be accomplished? Recall that the elements in the sorted linked list are arranged according to the value in some key—for example, numerically by identification number or alphabetically by name. If the range of possible values for the key can be determined, it is often a simple matter to set up dummy nodes with values outside of this range. A header node, containing a value smaller than any possible list element key, can be placed at the beginning of the list. A trailer node, containing a value larger than any legitimate element key, can be placed at the end of the list.

The header and the trailer are regular nodes of the same type as the real data nodes in the list. They have a different purpose, however, instead of storing list data, they act as placeholders.

Header node A placeholder node at the beginning of a list used to simplify list processing

Trailer node A placeholder node at the end of a list, used to simplify list processing

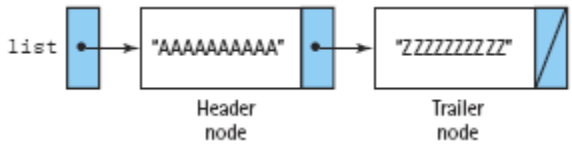


Figure 2.9.10 An “empty” list with a header and a trailer

If a list of students is sorted by last name, for example, we might assume that there are no students named “AAAAAAAAAA” or “ZZZZZZZZZZ”. We could, therefore, initialize our linked list to contain header and trailer nodes with these values as the keys, see Figure 2.9.10. How can we implement a general list ADT if we must know the minimum and maximum key values? We can use a parameterized class

constructor and let the user pass as arguments elements containing the dummy keys.

2.9.6 Linked List as an arrays of nodes

We tend to think of linked structures as being dynamically allocated as needed, using self-referential nodes as illustrated in Figure 2.9.11(a) but this is not a requirement. A linked list could be implemented in an array; the elements might be stored in the array in any order and "linked" by their indexes (see Figure 2.9.11b). In this section, we develop an array-based linked-list implementation.

For the array-based linked representation developed in this section, we can no longer rely on dynamic memory management support. Instead, we predetermine the maximum list size and instantiate an array of list nodes of that size. We then directly manage the nodes in the array. We keep a separate list of the available nodes and write routines to allocate and deallocate nodes, from and to this free list.

We have seen that dynamic allocation of list nodes has many advantages, so why would we even discuss using an array-of-nodes implementation instead? Remember that dynamic allocation is only one advantage of choosing a linked implementation another advantage is the efficiency of the insert and delete algorithms. Most of the algorithms that we have discussed for operations on a linked structure can be used for either an array-based or a reference-based implementation. The main difference is the requirement that we manage our own free space in an array-based implementation. Sometimes, managing the free space ourselves gives us greater flexibility.

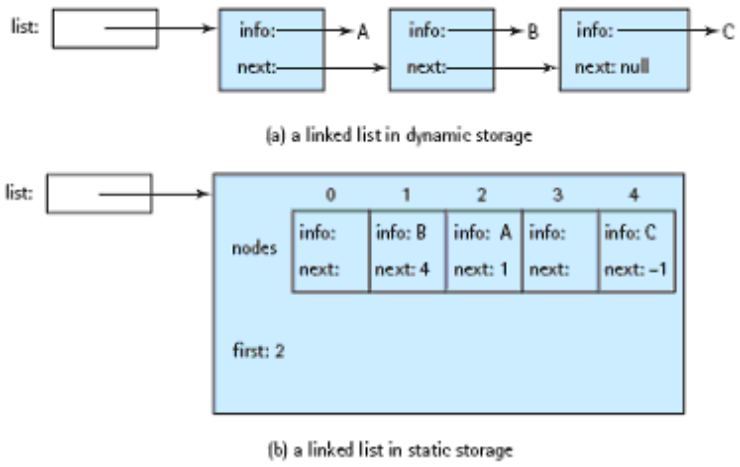


Figure 2.9.11 Linked lists in dynamic and static storage

Another reason to use an array of nodes is that there are programming languages that do not support dynamic allocation or reference types. You can still use linked structures if you are programming in one of these languages, using the techniques presented in this section.

With some languages, using references presents a problem when we need to save the information in a data structure between runs of a program. Suppose we want to write all the nodes in a list to a file and then use this file as input the next time we run the program to recreate the list. If the links are reference values—containing memory addresses—they are meaningless on the next run of the program because the program may be placed somewhere else in memory the next time. We must save the user data part of each node in the file and then rebuild the linked structure the next time we run the program. An array index, however, is still valid on the next run of the program. We can store the array information, including the next data index and then read it back in the next time we run the program.

Most importantly, there are times when dynamic allocation isn't possible or feasible or when dynamic allocation of each node, one at a time, is too costly in terms of time-especially in real-time system software such as operating systems, air traffic controllers and automotive systems. In such situations, an array-based linked approach provides the benefits of linked structures without the runtime costs.

Let's get back to our discussion of how a linked list can be implemented in an array. We can associate a next variable with each array node to indicate the array index of the succeeding node. The beginning of the list is accessed through a "reference" that contains the array index of the first element in the list. Figure 2.9.12 shows how a sorted list containing the elements "David," "Joshua," "Leah," "Miriam" and "Robert" might be stored in an array of nodes called nodes. Do you see how the order of the elements in the list is explicitly indicated by the chain of next indexes?

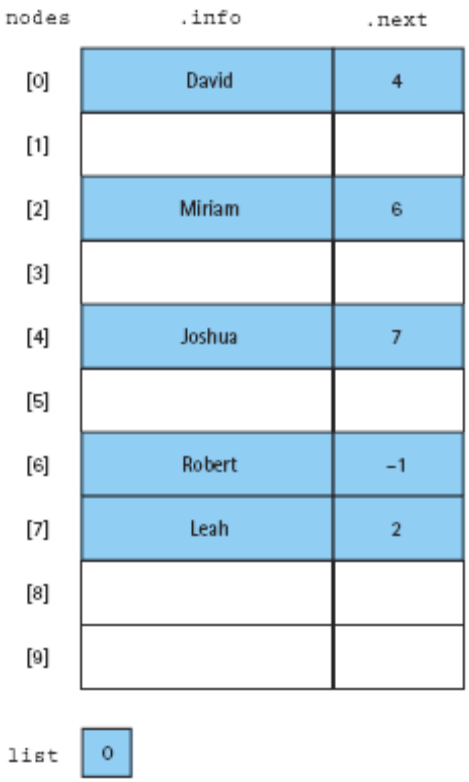


Figure 2.9.12 A sorted list stored in an array of nodes

What goes in the next index of the last list element? Its "null" value must be an invalid address for a real list element. Because the nodes array indexes begin at 0, the value _1 is not a valid index into the array, that is, there is no nodes[_1]. Therefore, _1 makes an ideal value to use as a "null" address. We could use the literal value _1 in our programs:

while (location != –1)

but it is better programming style to declare a named constant. We use the identifier

NUL and define it to be _1:

private static final int NUL = –1;

When an array-of-nodes implementation is used to represent a linked list, the programmer must write routines to manage the free space available for new list elements. Where is this free space? Look again at Figure 2.9.12. All of the array elements that do not contain values in the list constitute free space. Instead of the built-in allocator new, which allocates memory dynamically, we must write our own

method to allocate nodes from the free space. We call this method getNode. We use getNode when we insert new items onto the list.

When elements are deleted from the list, we need to free the node space, that is, to return the deleted node to the free space, so it can be used again later. We can't depend on a garbage collector the node we delete remains in the allocated array so it is not reclaimed by the run-time engine. We write our own method, freeNode, to put a node back into the pool of free space.

We need a way to track the collection of nodes that are not being used to hold list elements. We can link this collection of unused array elements together into a second list, a linked list of free nodes. Figure 2.9.13 shows the array nodes with both the list of elements and the list of free space linked through their next values. The list variable is a reference to a list that begins at index 0 (containing the value David). Following the links in next, we see that the list continues with the array slots at index 4 (Joshua), 9 (Leah), 2 (Miriam) and 6 (Robert), in that order. The free list begins at free, at index 1.

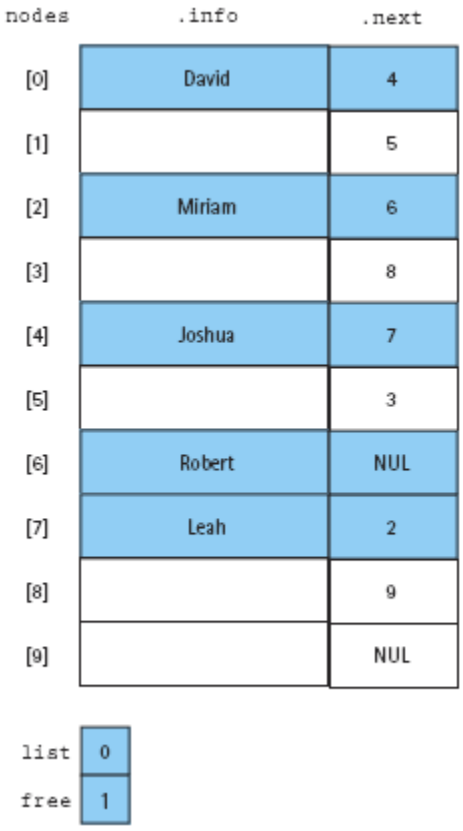


Figure 2.9.13 An array with linked list of values and free space

Following the next links, we see that the free list also includes the array slots at index 5, 3, 8 and 9. You see two NUL values in the next column because there are two linked lists contained in the nodes array so there are two end-of-list values in the array. There are two approaches to using an array-of-nodes implementation for linked structures. The first is to simulate dynamic memory with a single array. One array is used to store many different linked lists, just as the computer's free space can be dynamically allocated for different lists. In this approach, the references to the lists are not part of the storage structure but the reference to the list of free nodes is part of the structure. Figure 2.9.14 shows an array that contains two different lists. The list indicated by list1 contains the values "John," "Nell," "Susan" and "Suzanne" and the list indicated by list2 contains the values "Mark," "Naomi" and "Robert." The remaining three array slots in Figure 2.9.14 are linked together in the free list.

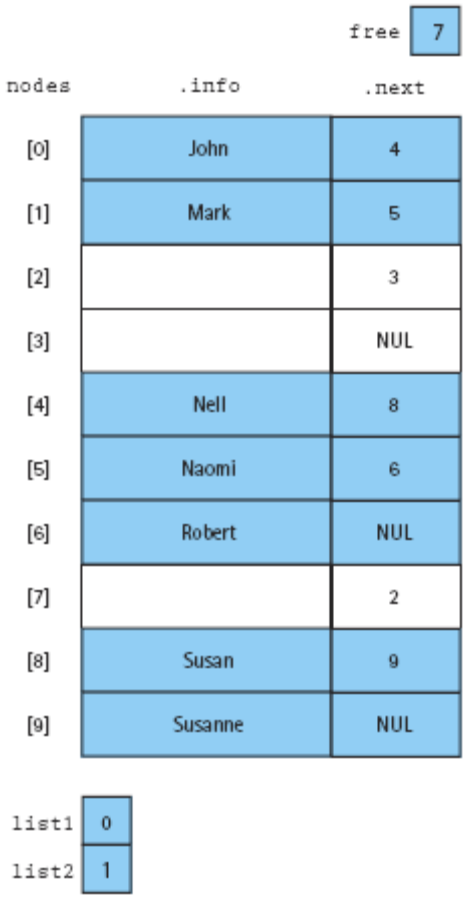


Figure 2.9.14 An array with three lists (including the free list)

Another approach is to have one array of nodes for each list. In this approach, the reference to the list is part of the storage structure itself (see Figure 2.9.15). This works since there is only one list. The list constructor has a parameter that specifies the maximum number of items to be on the list. This parameter is used to dynamically allocate an array of the appropriate size.

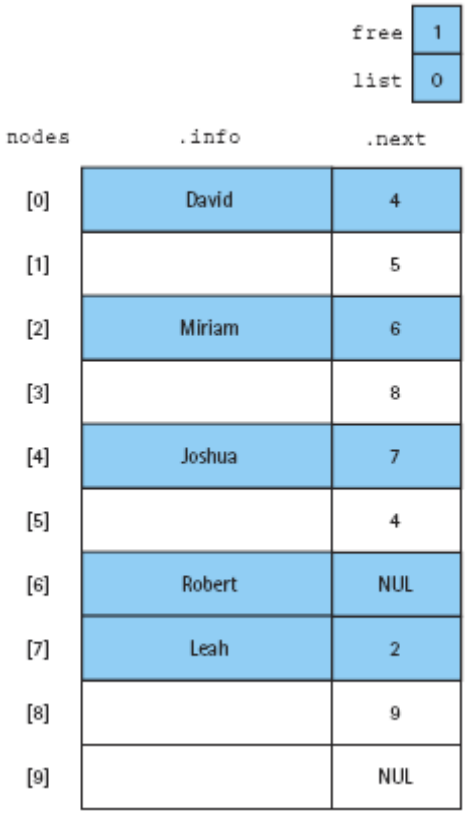


Figure 2.9.15 List and link structure are together

In this section, we implement this second approach. We call our new class ArrayLinkedList. In implementing our class methods, we need to keep in mind that there are two distinct processes going on within the array of nodes: bookkeeping relating to the space (such as initializing the array of nodes, getting a new node and a node) and the operations on the list that contains the user's data. The bookkeeping operations are transparent to the user. Our list interface does not change. In fact, our new class implements the ListInterface interface, just as all of our other list implementations have done. The private data, however, change. We need to include the array of nodes. Let's call this array nodes and have it hold elements of the class AListNode.

Objects of the AListNode class, then, contain two attributes: info of type Listable that holds a reference to a copy of the user's data and next, of the primitive type int, that holds the index of the next element on the list. In addition to the array of nodes, we need an integer "reference" to the first node of the list and another to the first free node. We call these list and free. And, of course, we still need our numItems and currentPos variables. Next is the beginning of our class file.

The class constructors for class ArrayLinkedList must allocate the storage for the array of nodes and initialize all of the private instance variables. They must also set up the initial free list of nodes. At the time of instantiation, all of the nodes are on the free list. So, the variable free is set to 0 to "reference" the first array node and the next value of that node is set to 1 and so on until all of the nodes are chained together. This initialization can be handled by a for loop, followed by a single assignment statement to set the last next value to NUL. To be consistent with our past array based implementation we should provide two constructors, one that accepts a size parameter and one that uses a default maximum size.

The methods that do the bookkeeping, getNode and freeNode are auxiliary ("helper") methods and therefore are private methods. The getNode method must return the index of the next free node. The easiest node to use is the one at the beginning of the free list, so getNode returns the value of free. Therefore, getNode must also update the value of free to indicate the next node on the free list. Other than the fact that we must be careful of our order of operations and use a temporary variable to hold the index we need to return.

The freeNode method must take the node index received as an argument and insert the corresponding node into the list of free nodes. As the first element in the list is the one that is directly accessible, we have freeNode insert the node being returned at the beginning of the free list, in the variable free.

The public methods are very similar to their reference-based linked list counterparts. From the point of view of the algorithm used, they are identical. Now we have a third implementation approach, the array-based linked approach. The following table shows equivalent expressions in each of our views of a list. It also shows the expressions for creating and deleting nodes, where appropriate.

Design Terminology	Array-Based	Reference-Based	Array Index Links
location.node()	`list[location]`	`location`	`nodes[location]`
location.info()	`list[location]`	`location.info`	`nodes[location].info`
location.next()	`list[location+1]`	`location.next`	`nodes[location].next`
allocate a node N	done by constructor	`ListNode N = new ListNode`	`N = getNode()`
free a node N	not applicable	remove links, garbage collector	`freeNode(N)`

We look here at some of the methods and leave the rest as an exercise. First an easy one is full. We have two ways of determining whether or not the list is full. As for our array-based list implementation, we can compare the number of items on the list to the size of the underlying array. If they are equal then the list is full. But there is an even easier way. If the entire array is being used to hold our list, then the list of free space must be empty.

The remaining methods can be implemented following the same scheme devised for the reference-based approach. You must be careful, however, to correctly transform the implementation. And don't forget that you have to handle the memory management yourself. Let's look at the delete method.

Think for a minute about how you would represent this statement using the approach of this section. It is not as simple as it might first appear. You need to compare the info value of the next element to item. Using the table above, you see that you access the info attribute of a location with nodes[location].info. However, you don't want the info value of the location, you want the info value of the next location. So, you must replace location with the expression that stands for the next location. In this case, that is nodes[location].next. Putting this altogether (see Figure 2.9.15), the corresponding code is:

```
while (item.compareTo(nodes[nodes[location].next].info) != 0)
location = nodes[location].next;
```

2.9.7 Applications of Linked Lists.

The main Applications of Linked Lists are

- For representing Polynomials

  It means in addition/subtraction /multiplication of two polynomials.

  Eg:p1=2x^2+3x+9 and p2=3x^3+5x+2

  p1+p2=3x^3+2x^2+8x+9

- In Dynamic Memory Management

- In Symbol Tables
- In Dynamic Memory Management
- Allocation and releasing memory at runtime.
- In Symbol Tables
- Balancing parenthesis
- Representing Sparse Matrix
- Representation and manipulation of long numbers.

2.9.8 Summary

The idea of linking the elements in a list has been extended to include lists with header and trailer nodes, circular lists and doubly linked lists. In addition to using dynamically allocated nodes to implement a linked structure, we looked at a technique for implementing linked structures in an array of nodes. In this technique the links are not references into the free store but indexes into the array of nodes. This type of linking is used extensively in systems software.

Although a linked list can be used to implement almost any list application, its real strength is in applications that largely process the list elements in order. This is not to say that we cannot do "random access" operations on a linked list. Our specifications include operations that logically access elements in random order. For instance, public methods retrieve and delete manipulate an arbitrary element in the list. However, at the implementation level the only way to find an element is to search the list, beginning at the first element and continuing sequentially to examine element after element. This search is O(N), because the amount of work required is directly proportional to the number of elements in the list. A particular element in a sequentially sorted list in an array, in contrast, can be found with a binary search, decreasing the search algorithm to $O(\log_2 N)$. For a large list, the O(N) sequential search can be quite time-consuming. There is a linked structure that supports $O(\log_2 N)$ searches the binary search tree.

2.9.9 Questions

1. Define circular linked list. What are its properties?
2. How insertion and deletion operations are performed on circular linked list?
3. Define doubly linked list. What are its properties?
4. How insertion and deletion operations are performed on doubly linked list?

5. What are the properties of linked list with header and trailers?
6. How linked list is implemented using arrays? Explain.
7. What are the various applications of linked list?

2.9.10 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley.

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.10 **AUTHOR: DHARAM VEER SHARMA**

LINKED REPRESENTATION OF STACKS AND QUEUES

2.10.9 Suggested readings

2.10.1 Objective of the lesson

You learned about stacks back in lesson 16 and about queues in lesson 18 when you discovered how to use an array to create your own stack or queue. However, using arrays presents a problem: you cannot adjust the size of the stack or queue when the program runs. The solution is to use a linked list to create a stack or queue. You learned about linked lists in general in lesson 20. In this lesson, you'll learn how to use a linked list to create a stack and a queue.

2.10.2 Introduction

A stack is a data structure that organizes data similar to how you organize dishes in a stack on your kitchen counter. The newest dish is on top and the oldest is on the bottom of the stack.

When accessing dishes, the last disk on the stack is the first dish removed from the stack. If you want the third dish, you must remove the first two dishes from the top of the stack first so that the third dish becomes the top of the stack and you can remove it. There is no way to remove a dish from anywhere other than the top of the stack. You'd need to use a different kind of data structure (or stacking system) if you wanted to randomly access dishes.

A stack is useful whenever you need to store and retrieve data in last in, first out order. For example, your computer processes instructions using a stack in which the next instruction to execute is at the top of the stack.

In lesson 2.6, you learned that a queue is a sequential organization of data where data is accessible on a first in, first out (fifo) basis, which is similar to the line that you stand in to buy concert tickets.

The queue in Lesson 18 was created using an array to store data. As you'll recall, the array is separate from the queue. Data is assigned to elements of the array. The queue itself consists of two variables called front and back. Each points to the array element that is at the front of the queue or at the back of the queue. When data is removed from the front of the queue, the program changes the value of the front variable to point to the next array element. However, the data removed from the queue remains assigned to the array. That is, data isn't removed from memory.

There is a serious problem with using arrays to store data for queues: you must know the size of the array when you write the program. An array can store only a

specific maximum number of elements at any point in time, similar to an architect designing a specific space for a box office that can accommodate a maximum number of fans at any point in time.

However, there is a difference between exceeding the number of array elements and overflowing the space around the box office: unlike the stadium, there is no parking lot for fans to gather in while waiting to get in the queue to purchase tickets inside a computer.

Programmers work around the size issue by using a linked list instead of an array when creating a queue. As you learned in previous lessons, a linked list can grow and shrink at runtime based on the needs of the application.

2.10.3 Linked representation of Stack

Although we discuss data as being stacked like a stack of dishes, it isn't physically stacked at all. Instead, data is linked together sequentially in a list, where the last data always appears at the front of the list. Data is removed only from the front of the list.

You create this sequential list by using a linked list. In Chapter 2.10, you learned that a linked list contains entries called nodes. A node has three subentries, data and two pointers. The data subentry is the data stored on the stack. Pointers point to the previous node and the next node (Figure 2.10.1). When you enter a new item on a linked list, you allocate the new node and then set the pointers to the previous and next nodes.

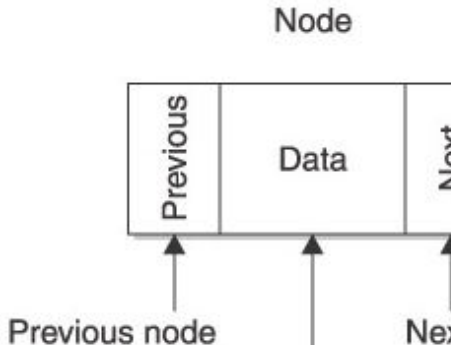


Figure 2.10.1: A node contains references to the previous node and the next node in the linked list and contains data that is associated with the current node.

A node is defined in C++ by using a structure, which is a user-defined data type. The following structure defines a node:

```
typedef struct Node
```

```
{
   struct Node(int data)
   {
      this->data = data;
      previous = NULL;
      next = NULL;
   }
   int data;
   struct Node* previous;
   struct Node* next;
} NODE;
```

The structure is called Node. The name of the structure creates an instance of the structure similar to the way a constructor creates an instance of a class and data type. Let's skip the second definition of the structure and look at the last three statements within the structure because statements at the beginning of the structure actually create an instance of the structure and don't define the structure. The first statement declares an integer that stores the current data of the node. The next two statements declare pointers to the previous and next nodes in the linked list.

The constructor initializes elements of the node when the node is created, which is similar to the way constructors work in a class definition. You provide the current data to the structure when you create a new node. This data is assigned to data in the argument list, which is then assigned to the data element of the instance of the structure. The previous and next nodes are initialized to NULL , which indicates there are no other elements of the linked list. The NULL is replaced with a reference to a node when a new node is added to the linked list.

A LinkedList class is defined to create and manage a linked list. There are two data members and six function members defined in the LinkedList class. Data members are pointers to instances of the Node structure. The first pointer is called front, and it references the first node on the linked list. The second pointer is called back, and it references the last node on the linked list.

Both the front and back pointers are declared in the protected access specifier area of the class definition because the LinkedList class is inherited by the

StackLinkedList class, which you'll learn about in the "The StackLinkedList Class" section of this chapter. The StackLinkedList class uses the front and back pointers.

The six member functions manipulate the linked list. These function members are the constructor, destructor, appendNode(), displayNodes(), displayReverseNodes(), and destroyNodes().

Here is the LinkedList class definition. The front and back pointers are defined in the protected access specifier area because the StackLinkedList class will use them:

```
class LinkedList{
  protected:
    NODE* front;
    NODE* back;
  public:
    LinkedList();
    ~LinkedList();
    void appendNode(int);
    void displayNodes();
    void displayNodesReverse();
    void destroyList();
};
```

2.10.3.1 The StackLinkedList Class

An efficient programmer does not repeat code if possible and instead inherits attributes and behaviors of another class, defining a LinkedList class to create and manipulate a linked list. An efficient programmer might also define a StackLinkedList class to create and manipulate a stack-linked list. The StackLinkedList class inherits attributes and behaviors of the LinkedList class and then defines other behaviors that are necessary to work with a stack-linked list.

In addition to the attributes and behaviors defined in the LinkedList class, the StackLinkedList class requires five behaviors defined as member functions: a constructor and destructor, push(), pop(), and isEmpty(). The StackLinkedList class definition is shown here:

```
class StackLinkedList : public LinkedList{
```

```
    public:
      StackLinkedList();
      virtual ~StackLinkedList();
      void push(int);
      int pop();
      bool isEmpty();
};
```

2.10.3.2 StackLinkedList Constructor and Destructor

The constructor and destructor of the StackLinkedList class may be confusing the first time you look at them because both are empty and there aren't any instructions specified in the body of the constructor and destructor, as shown here:

```
StackLinkedList(){
}
~StackLinkedList(){
}
```

The constructor is empty because the constructor of the LinkedList class is called before the constructor of the StackLinkedList class. You'll recall that the StackLinkedList class inherits the LinkedList class. The LinkedList class constructor initializes the front and back pointers of the linked list to NULL. Therefore, there is nothing else for the StackLinkedList class constructor to do.

Likewise, the destructor of the LinkedList class is called before the destructor of the StackLinkedList class. The LinkedList class constructor deletes all memory that is associated with the nodes of the linked list. Therefore, the destructor of the StackLinkedList class also has nothing to do.

2.10.3.3 Pushing a Node onto a Stack-Linked List

You learned that data is placed at the top of the stack and removed from the top of the stack. Programmers call this pushing data onto the stack and popping data off the stack. The same steps occur when using a linked list for the stack, but instead of placing data at the next available index in an array, it is placed at the back of the linked list.

You'll need to define a push() member function for the StackLinkedList class that is called whenever data is added to the stack. Remember that you are really adding a node to the linked list and not simply data. Data is contained in the node.

To add a node to the stack, you use the same steps you use to add a node to a linked list. This means that the appendNode() member function of the LinkedList class can be used to place a new node on the stack. Therefore, all you need is to call the appendNode() member function from the push() member function. Because appendNode() is public, you could just call appendNode directly to push a node onto the stack, but putting a push() function in the stack class makes this more intuitive to somebody using this class. This also helps hide the underlying implementation so using the class is a little more straightforward.

The appendNode() member function requires one argument, which is the data that is assigned to the new node. You must define the push() member function to accept the same data as its argument in order to pass this data to the appendNode() member function. This is illustrated in the following example. The push() member function requires an integer passed as an argument. The integer is then passed to the appendNode() member function within the body of the push() function definition.

```
void push(int x){
   appendNode(x);
}
```

2.10.3.4 Popping a Node from a Stack-Linked List

You'll also need to define a member function to pop a node from the stack. In this example, we'll call it pop(). Because you're using the linked list as the stack, the pop() member function must remove the node from the back of the linked list.

Unfortunately, you cannot simply call a member function of the LinkedList class to pop the node off the stack because the LinkedList class doesn't define a member function that removes a node from the linked list. If you had a member function in the base class for removeBack(), you could call that to pop a node off the list. In this case, you'll need to define a pop() function in the StackLinkedList class to do this. This will give you a last in, first out access to the stack.

Here is the definition of the pop() member function. Refer to the picture of the stacked linked list in Figure 2.10.2 as you read to help you understand how the pop() member function works.

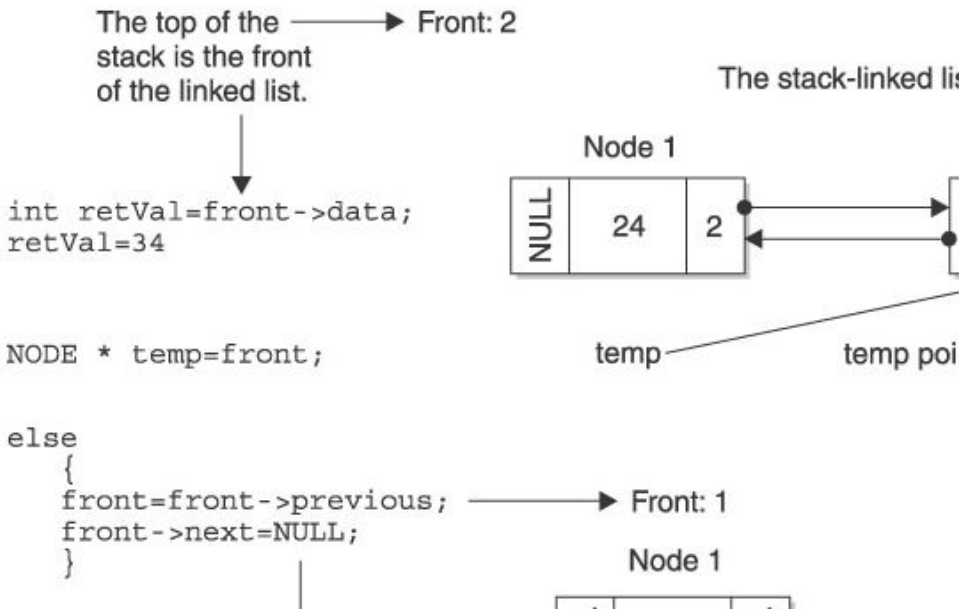


Figure 2.10.2: The pop() member function removes the node at the top of the stack, which is the node at the front of the linked list.

```
int pop()
{
  if (isEmpty())
  {
    return -1;
  }
  int retVal = back->data;
  NODE * temp = back;
  if (back->previous == NULL)
  {
     back = NULL;
     front = NULL;
  }
  else
  {
```

```
        back = back->previous;
        back->next = NULL;
      }
      delete temp;
      return retVal;
    }
```

First, you must determine if there is anything on the stack by calling the isEmpty() member function. We'll show you how this member function works later in this section. For now, understand that the isEmpty() member function returns a Boolean true if the stack is empty, or a Boolean false if it is not. You can see in Figure 2.10.2 that the stack has two nodes on the stack, so it is not empty. Therefore, the return statement in the if statement is not executed.

The pop() member function refers to the back attribute of the LinkedList class. It is important to remember that the back attribute refers to the top of the stack. Nodes will be removed from the back of the linked list to do a pop operation. Therefore, the value of front is Node 2.

The value of Node 2 is assigned to the retVal variable, which is the value returned by the pop() member function if there is a node on the stack. This pops the value from the stack.

Next, the address of the back node, which is Node 2, is assigned to a temporary pointer. The node that the temporary pointer points to is removed from memory with the delete operator at the end of the pop() member function.

Next, you determine if the node at the back of the stack was the only node on the linked list. You make this determination by seeing if the previous attribute of the node is NULL. If the previous pointer on the back of the list is NULL, this indicates that there's only one node in the linked list.

Be careful when analyzing the pop() member function. Remember that the back of the linked list is the top of the stack and that the bottom of the stack is the top of the linked list. If the pop() member function is removing the only node on the stack, then the front and back attributes of the LinkedList class are set to NULL, indicating there are no nodes left on the linked list after the pop() is executed.

However, if there is at least one node on the stack, statements within the else statement are executed, as in Figure 2.10.2. The first statement within the else statement assigns the previous attribute of the back attribute as the new back. In

Figure 2.10.2, the previous attribute is 1. This tells you that Node 1 comes before Node 2. You then assign Node 1 as the new back of the stack. Remember that there isn't a next node on the stack because you are always working with the back of the linked list. Therefore, you need to assign NULL to the next attribute of the back node, which is Node 1. This makes Node 1 at the top of the stack.

The next to last statement removes the node from memory using the delete operator. The temporary pointer points to the memory address of the node that was removed from the stack. The last statement returns the value of the node that was removed from the stack.

2.10.3.5 Determine If the Stack Is Empty

The pop() member function must determine if the stack is empty, or it will attempt to remove a node that isn't on the stack. The pop() member function determines if the stack is empty by calling the isEmpty() member function, which you must define as part of the StackLinkedList class.

The isEmpty() member function is a simple function, as shown next. It determines if the stack is empty by seeing if the value of the front attribute of the LinkedList class is NULL. If so, then a Boolean true is returned; otherwise, a Boolean false is returned. If the stack is empty, both front and back are equal to NULL but you only need to check one of them.

```
bool isEmpty()
    {
      if(front == NULL)
      {
        return true;
      }
      else
      {
        return false;
      }
    }
```

2.10.4 StackLinked List Using C++

Now that you have an understanding of components, you need to create a stack-linked list. In this section, we'll focus on assembling them into a working C++ application. Some programmers organize components of a stack-linked list into five files: LinkedList.h, LinkedList.cpp, StackLinkedList.h, StackLinkedList.cpp, and StackLinkedListDemo.cpp. All these files are joined together at compile time to create the executable.

The LinkedList.h file is the header file that contains the definition of the Node structure and the definition of the LinkedList class. The LinkedList.cpp is a source code file that contains the implementation of member functions of the LinkedList class. The StackLinkedList.h file is the header that contains the definition of the StackLinkedList class. The StackLinkedList.cpp is the source code file that contains the implementation of member functions of the StackLinkedList class.

The StackLinkedListDemo.cpp contains the application. It is here where an instance of the StackLinkedList class is declared and member functions are called.

2.10.4.1 LinkedList Header File and LinkedList Functions

The LinkedList.h file and the LinkedList.cpp file are shown in the following code. The front and back attributes defined in the LinkedList class in the LinkedList.h file are defined within the protected access specifier section of the class definition. The StackLinkedList class needs access to these variables, so you protect them so they're visible to the subclass.

```
//LinkedList.h
typedef struct Node
{
  struct Node(int data)
  {
    this->data = data;
    previous = NULL;
    next = NULL;
  }
  int data;
  struct Node* previous;
```

```
   struct Node* next;
} NODE;
class LinkedList
{
   protected:
      NODE* front;
      NODE* back;
   public:
      LinkedList();
      ~LinkedList();
      void appendNode(int);
      void displayNodes();
      void displayNodesReverse();
      void destroyList();
};
//LinkedList.cpp
#include "LinkedList.h"
LinkedList::LinkedList()
{
   front = NULL;
   back = NULL;
}
LinkedList::~LinkedList()
{
   destroyList();
}
void LinkedList::appendNode(int data)
{
```

```
  NODE* n = new NODE(data);
   if(back == NULL)
   {
      back = n;
      front = n;
   }
   else
   {
      back->next = n;
      n->previous = back;
      back = n;
   }
}
void LinkedList::displayNodes()
{
   cout << "Nodes:";
   NODE* temp = front;
   while(temp != NULL)
   {
      cout << " " << temp->data;
      temp = temp->next;
   }
}
void LinkedList::displayNodesReverse()
{
   cout << "Nodes in reverse order:";
   NODE* temp = back;
   while(temp != NULL)
```

```
    {
      cout << " " <<  temp->data;
      temp = temp->previous;
  }
}
void LinkedList::destroyList()
{
  NODE* temp = back;
  while(temp != NULL)
  {
    NODE* temp2 = temp;
    temp = temp->previous;
    delete temp2;
  }
  back = NULL;
  front = NULL;
}
```

2.10.4.2 StackLinkedList Header File and StackLinkedList Source File

The StackLinkedList.h file contains the definition of the StackLinkedList class, as shown next. Below the StackLinkedList.h file is the StackLinkedList.cpp file that contains the definitions of member functions.

The class definition and each member function were explained in the "The StackLinkedList Class" section of this lesson.

```
//StackLinkedList.h
 class StackLinkedList : public LinkedList
{
  public:
    StackLinkedList();
    virtual ~StackLinkedList();
```

```
        void push(int);
        int pop();
        bool isEmpty();
};
//StackLinkedList.cpp
StackLinkedList.h
StackLinkedList::StackLinkedList()
{
}
StackLinkedList::~StackLinkedList()
{
}
void StackLinkedList::push(int x)
{
  appendNode(x);
}
int StackLinkedList::pop()
{
  if(isEmpty())
  {
    return -1;
  }
  int retVal = back->data;
  NODE* temp = back;
  if(back->previous == NULL)
  {
    back = NULL;
    front = NULL;
```

```
    }
    else
    {
       back = back->previous;
      back->next = NULL;
    }
    delete temp;
    return retVal;
}
bool StackLinkedList::isEmpty()
{
    if(front == NULL)
    {
       return true;
    }
    else
    {
       return false;
    }
}
```

2.10.4.3 StackLinkedList Application

The StackLinkedListDemo.cpp file contains the actual stack application, as shown in the following code listing. The application begins by declaring an instance of the StackLinkedList class. Remember that this statement also indirectly calls the constructor of the LinkedList class, which is inherited by the StackLinkedList class.

The application then calls the push() member function to push the values 10, 20, and 30 onto the stack. The displayNodes() member function is then called to display the values on the stack.

The pop() member function is then called to remove the last node on the stack, which is then displayed on the screen (see Figure 2.10.3). The program then calls the delete operator to remove the stack from memory.

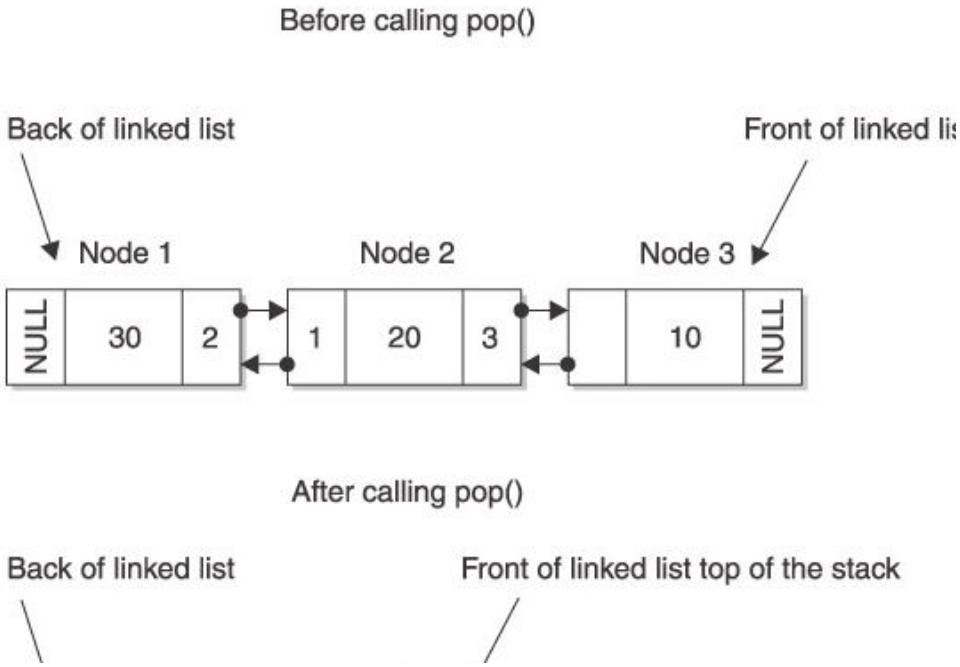


Figure 2.10.3: Before the pop() member function is called, there are three nodes on the stack. Two nodes remain after pop() is called.

Here's the output of this program:

Nodes: 10 20 30 10

```
//StackLinkedListDemo.cpp
#include <iostream>
using namespace std;
void main(){
  StackLinkedList* stack = new StackLinkedList();
  stack->push(10);
  stack->push(20);
  stack->push(30);
  stack->displayNodes();
  cout << stack->pop() << endl;
```

```
    delete stack;
}
```

2.10.5 Linked representation of Queue

Conceptually, a linked list queue is the same as a queue built using an array. Both store data. Both place data at the front of the queue and remove data from the front of the queue. However, in an array queue, data is stored in an array element. In a linked list queue, data is stored in a node of a linked list. The linked list queue consists of three major components: the node, the LinkedList class definition, and the QueueLinkedList class definition. Collectively, they are assembled to organized data into a queue.

Node contains the data and pointers to the previous node and the next node on the linked list (Figure 2.10.4). The next code snippet is the user-defined data type structure node. You'll be using the following user-defined data type structure in this lesson to create the linked list queue.

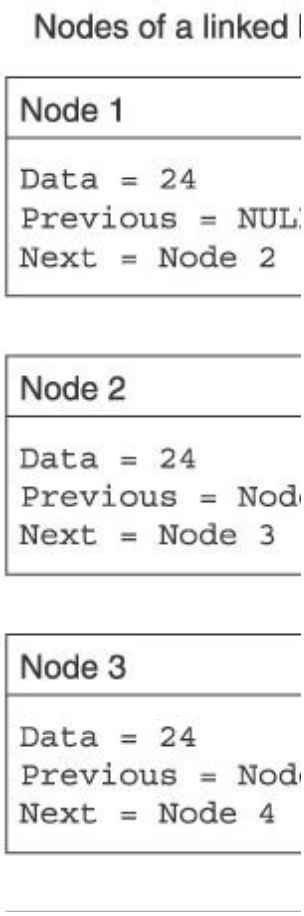


Figure 2.10.4: Each node points to the previous node and the next node.

The name of the user-defined data structure is called Node in this example and is used within the LinkedList class definition to declare instances of the node. The last three statements in the structure declare an integer that stores the current data and declares two pointers to reference the previous node and the next node on the linked list.

Each time a node is created, the user-defined structure is passed data for the node. Pointers to the previous node and to the next node are assigned NULL, which indicates there isn't a previous node or next node. NULL is replaced with reference to a node once the new node is added to the linked list.

```
typedef struct Node
{
   struct Node(int data)
```

```
    {
      this->data = data;
      previous = NULL;
      next = NULL;
    }
    int data;
    struct Node* previous;
    struct Node* next;
} NODE;
```

The LinkedList class creates and manages the linked list. In addition, the LinkedList class defines member functions that manage the linked list. These are the same member functions described earlier in this lesson, a constructor and destructor, appendNode(), displayNodes(), displayNodesReverse(), and destroyList().Here is the LinkedList class definition that you'll use to create the linked list queue:

```
class LinkedList
{
  protected:
    NODE* front;
    NODE* back;
  public:
    LinkedList();
    ~LinkedList();
    void appendNode(int);
    void displayNodes();
    void displayNodesReverse();
    void destroyList();
};
```

Programmers usually place the node structure and the LinkedList class definition in the same header file, LinkedList.h. Placing the code needed to create a linked list

in one file like this helps keep it organized. Programmers then use the preprocessor directive #include to include LinkedList.h in any program that uses a linked list. The last component of the linked list queue is the QueueLinkedList class definition. The QueueLinkedList class inherits the LinkedList class and then defines member functions that are specifically designed to manage a queue.

You might wonder why you don't simply define one class that combines the LinkedList class and the QueueLinkedList class. Intuitively, this seems to be a good idea because everything needed to create a linked list queue is contained in one file. However, doing so repeats code, which is something programmers avoid if possible.

For example, definitions of a node and the LinkedList class would be located in two places. If you needed to upgrade either definition, you'd need to remember all the places where they are defined in your code. A better approach is to place each definition in its own file (for example, LinkedList.h, QueueLinkedList.h) so code won't be repeated.

2.10.5.1 The QueueLinkedList class

Here is the definition of the QueueLinkedList class that you'll use to create a queue. Programmers save this definition in a file called QueueLinkedList.h. The QueueLinkedList class has five member functions: a constructor and destructor, enqueue(), dequeue(), and isEmpty().

```
//QueueLinkedList.h
#include "LinkedList.h"
class QueueLinkedList : public LinkedList
{
  public:
    QueueLinkedList();
    virtual ~QueueLinkedList();
    void enqueue(int);
    int dequeue();
    bool isEmpty();
};
```

The constructor and destructor of the QueueLinkedList class are empty, as shown in the next code snippet. The constructor typically initializes data members of an

instance of the class. In the case of the linked list queue, initialization is performed by the constructor of the LinkedList class, which is called before the constructor of the QueueLinkedList class. This means there isn't anything for the constructor of the QueueLinkedList class to do.

The destructor typically frees memory used by an instance of a class. The linked list used for the queue is removed by the destructor of the LinkedList class, which is also called before the destructor of the QueueLinkedList class. Therefore, there isn't anything for the destructor of the QueueLinkedList to do either.

```
QueueLinkedList::QueueLinkedList(){

}

QueueLinkedList::~QueueLinkedList(){

}
```

2.10.5.2 Enqueue

The enqueue() member function of the QueueLinkedList class is called whenever a new node is placed on the queue. As you see from the function definition in the next code snippet, the enqueue() member function is sparse because it contains only one statement, which calls the appendNode() member function of the LinkedList class.

You don't have to include additional statements in the enqueue() member function because placing a node on the queue is the same process as appending a node to the linked list. Each new node is placed at the back of the linked list. Therefore, the appendNode() member function is all you need.

You may wonder why the new node is being placed on the back of the queue, but it's just because you're reusing the same code in the LinkedList class. The new node will be placed on the back of the queue like a line at the grocery store. Nodes will be pulled off the front.

The enqueue() member function has one argument, which is the data that is being assigned to the new node. In this example, the node is used to store an integer. However, you can store any type of data in a node. In fact, the data can be a pointer to a set of data such as student information. To change this example from integer data to another type of data, you'd need to change the data element in the Node structure to reflect the type of data you want to store in the node.

Data received by the enqueue() member function is passed to the appendNode() member function. Figure 2.10.5 illustrates how the appendNode() member function places a new node at the back of the linked list. At the top of the illustration is a

linked list that contains two nodes. The appendNode() is then called to add a new node to the back of this linked list.

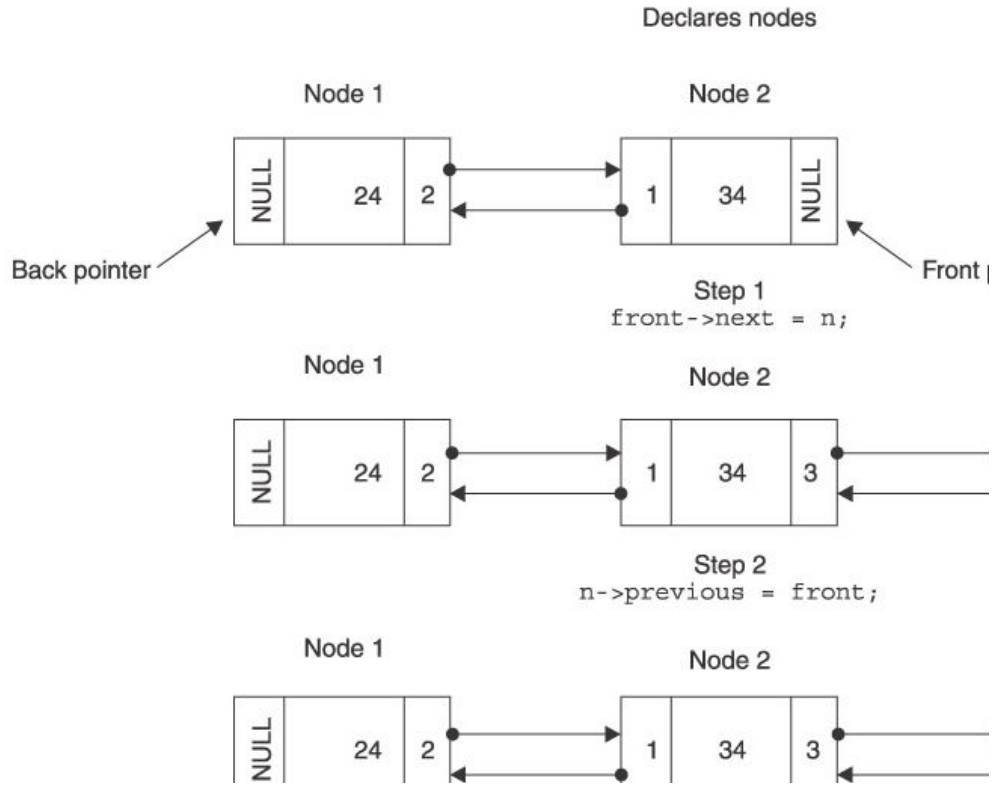


Figure 2.10.5: A new node is added to the queue at the back of the linked list.

The first step in this process assigns a reference to the new node to the next member of the front node. The front node is Node 2 and is assigned the reference Node 3 as the value of the next node in the linked list. This makes Node 3 the back of the linked list.

The second step assigns reference to Node 2 as the value of the previous node in Node 3. This means the program looks at the value of the previous node of Node 3 to know which node comes before Node 3 in the linked list. The last step is to assign Node 3 as the new value of the back data member of the LinkedList class.

```
void enqueue(int x){
  appendNode(x);
}
```

2.10.5.3 Dequeue

The dequeue() member function of the QueueLinkedList class removes a node from the front of the queue. Unfortunately, there aren't any member functions in the LinkedList class that remove a node from the back of the linked list. Therefore, the dequeue() member function must do the job.

The dequeue() function begins by determining if there are any nodes on the queue by calling the isEmpty(). The isEmpty() member function returns a Boolean true if the queue is empty, in which case the dequeue() returns a –1. A Boolean false is returned if there is at least one node on the queue.

Figure 2.10.6 shows how the dequeue() member function works. You'll notice there are three nodes on the queue, so the isEmpty() member function returns a Boolean false, causing the program to remove the front node from the queue.

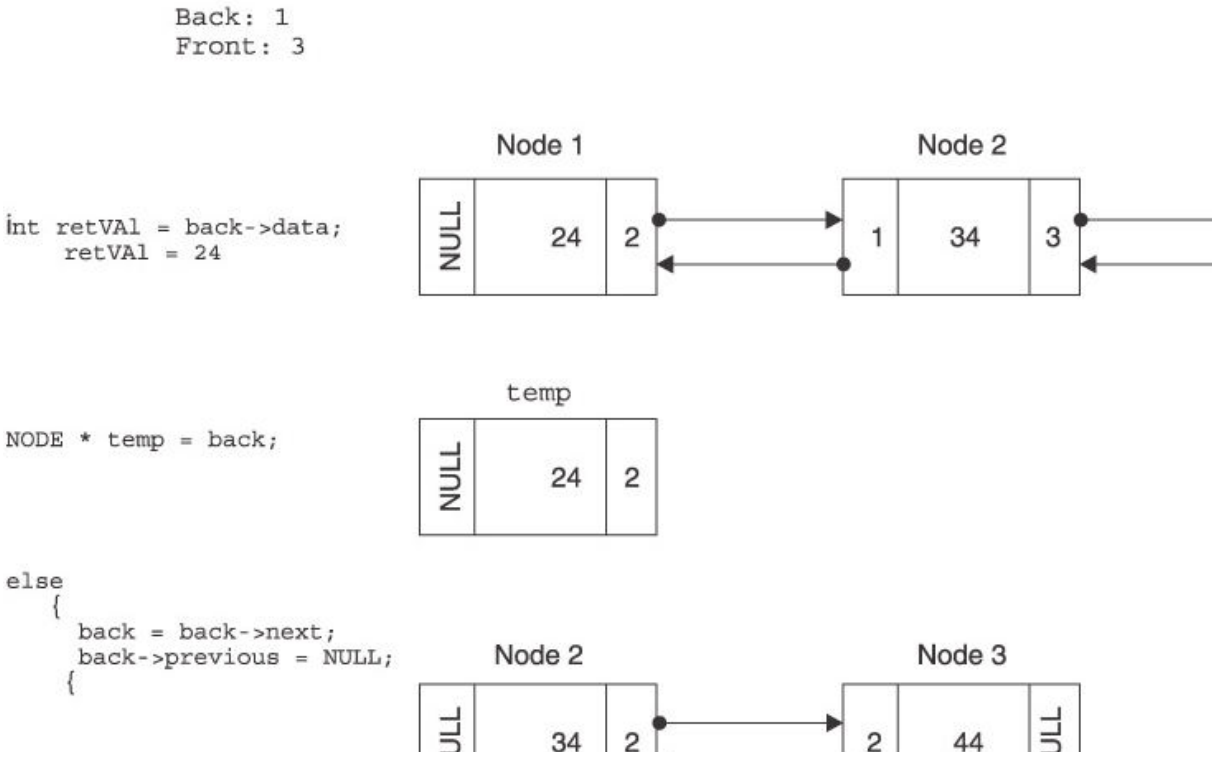


Figure 2.10.6: Node 1 is removed from the back of the queue by the dequeue() member function.

The removal process starts by assigning the data of the node at the front of the queue to a variable called retVal. The value of the retVal is returned by the dequeue() member function in the last statement of the function. Next, reference to the front node is assigned to the temp variable pointer. The delete operator later in the function uses the temp variable to remove the back node from memory.

Next, the function determines if there is another node on the queue by examining the value of the next member of the front node. If the value of the next member is NULL, there aren't any other nodes on the queue. In this case, the front and back members of the LinkedList class are set to NULL, indicating that the queue is empty. However, if the next member of the front node is not NULL, the value of the next member of the front node is assigned to the front member of the LinkedList class. In this example, Node 2 is the next node following Node 1. Node 2 becomes the new front of the queue.

Notice that the previous member of Node 2 is set to Node 1. However, Node 1 no longer exists. Therefore, the previous member must be set to NULL because there is not a previous node. Node 2 is the front of the queue.

The temp node is then deleted from memory. Remember that the temp node is a pointer that points to Node 1, and Node 1 no longer exists in memory. The final statement returns the value of the retVal variable, which is the data that was stored in Node 1.

```
int dequeue(){
  if(isEmpty()){
    return -1;
  }
  int retVal = front->data;
  NODE* temp = front;
  if(front->next == NULL){
    back = NULL;
    front = NULL;
  }
  else{
    front = front->next;
```

```
        front->previous = NULL;
      }
      delete temp;
      return retVal;
    }
```

The isEmpty() member function determines if there are any nodes on the queue, which is called by the dequeue() member function. The isEmpty() member function examines the value of the front data member of the LinkedList class. If the value of front is NULL, then the queue is empty; otherwise, the queue has at least one node.

The isEmpty() member function returns a Boolean true if the value of front is NULL, otherwise a Boolean false is returned as shown in the definition of the isEmpty() here:

```
    bool isEmpty(){
      if(front == NULL){
        return true;
      }
      else{
        return false;
      }
    }
```

2.10.6 Linked List Queue Using C++

Now that you understand how to create a queue using a linked list, let's assemble all the pieces and build a working queue in C++. Programmers organize an application into several files, each containing a distinct component of the application.

In the case of the demo queue application illustrated next, there are five distinct components: the driver file (QueueLinkedListDemo.cpp), the header file that contains the definition of the node and the LinkedList class (LinkedList.h), the file that contains the implementation of member functions of the LinkedList class (LinkedList.cpp), the header file that contains the definition of the QueueLinkedList

class (QueueLinkedList.h), and the file that contains the implementation of member functions of the QueueLinkedList class (QueueLinkedList.cpp).

The application is called QueueLinkedListDemo, and it uses a linked list to create a queue, as shown in the next code. The application begins by declaring an instance of the QueueLinkedList class using the new operator. It then declares a pointer to an instance of the QueueLinkedList. The pointer is called queue, which is assigned a reference to the instance created by the new operator.

The enqueue() member function is then called three times, each time another node is placed on the queue. The queue shown in Figure 2.10.7 depicts the queue after the last time the enqueue() method is called.

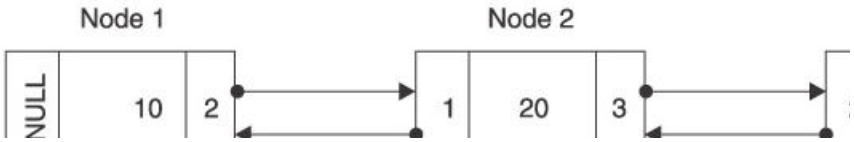


Figure 2.10.7: The queue after all three values are placed on the queue.

The dequeue() member function is then called to remove the first node from the queue and display its data member on the screen. Figure 2.10.8 shows the queue after the dequeue() member function is called.

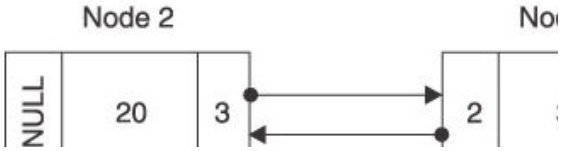


Figure 2.10.8: The queue after the dequeue() member function is called

The last statement in the program removes the queue from memory.

Each of the remaining components of the application was discussed in the previous section.

```
//QueueLinkedListDemo.cpp
#include <iostream>
using namespace std;
void main(){
```

```
    QueueLinkedList* queue = new QueueLinkedList();
    queue->enqueue(10);
    queue->enqueue(20);
    queue->enqueue(30);
    cout << queue->dequeue() << endl;
    delete queue;
}
//LinkedList.h
typedef struct Node
{
    struct Node(int data)
    {
        this->data = data;
        previous = NULL;
        next = NULL;
    }
    int data;
    struct Node* previous;
    struct Node* next;
} NODE;
class LinkedList
{
    protected:
        NODE* front;
        NODE* back;
    public:
        LinkedList();
        ~LinkedList();
```

```
        void appendNode(int);
        void displayNodes();
        void displayNodesReverse();
        void destroyList();
};
//LinkedList.cpp
#include "LinkedList.h"
LinkedList::LinkedList()
{
  front = NULL;
  back = NULL;
}
LinkedList::~LinkedList()
{
  destroyList();
}
void LinkedList::appendNode(int data)
{
  NODE* n = new NODE(data);
  if(front == NULL)
  {
    back = n;
    front = n;
  }
  else
  {
    back->next = n;
    n->previous = back;
```

```
        back = n;
    }
}
void LinkedList::displayNodes()
{
    cout << "Nodes:";
    NODE* temp = front;
    while(temp != NULL)
    {
        cout << " " << temp->data;
        temp = temp->next;
    }
}
void LinkedList::displayNodesReverse()
{
    cout << "Nodes in reverse order:";
    NODE* temp = back;
    while(temp != NULL)
    {
        cout << " " <<  temp->data;
        temp = temp->previous;
    }
}
void LinkedList::destroyList()
{
    NODE* temp = back;
    while(temp != NULL)
    {
```

```
        NODE* temp2 = temp;
        temp = temp->previous;
        delete temp2;
      }
      back = NULL;
      front = NULL;
    }
    //QueueLinkedList.h
    #include "LinkedList.h"
    class QueueLinkedList : public LinkedList
    {
      public:
        QueueLinkedList();
        virtual ~QueueLinkedList();
        void enqueue(int);
        int dequeue();
        bool isEmpty();
    };
    //QueueLinkedList.cpp
    #include "QueueLinkedList.h"
    QueueLinkedList::CQueueLinkedList()
    {
    }
    QueueLinkedList::~CQueueLinkedList()
    {
    }
    void QueueLinkedList::enqueue(int x)
    {
```

```
    appendNode(x);
}
int QueueLinkedList::dequeue()
{
    if(isEmpty())
    {
        return -1;
    }
    int retVal = front->data;
    NODE* temp = front;
    if(front->next == NULL)
    {
        back = NULL;
        front = NULL;
    }
    else
    {
        front = front->next;
        front->previous = NULL;
    }
    delete temp;
    return retVal;
}
bool QueueLinkedList::isEmpty()
{
    if(front == NULL)
    {
        return true;
```

```
    }
    else
    {
      return false;
    }
  }
```

2.10.7 Summary

There is a serious problem with using arrays to store data for queues: you must know the size of the array when you write the program. An array can store only a specific maximum number of elements at any point in time. Programmers work around the size issue by using a linked list instead of an array when creating a stack or a queue. A linked list can grow and shrink at runtime based on the needs of the application.

Although we discuss data as being stacked like a stack of dishes, it isn't physically stacked at all. Instead, data is linked together sequentially in a list, where the last data always appears at the front of the list. Data is removed only from the front of the list. You create this sequential list by using a linked list.

Conceptually, a linked list queue is the same as a queue built using an array. Both store data. Both place data at the front of the queue and remove data from the front of the queue. However, in an array queue, data is stored in an array element. In a linked list queue, data is stored in a node of a linked list.

2.10.8 Questions

1. What is a stack-linked list?
2. How does a stack-linked list differ from a linked list?
3. What is the benefit of using a stack-linked list?
4. Where is the front of the stack in a stack-linked list?
5. What is the maximum number of nodes that you can have on a stack-linked list?
6. Can a node on a stack-linked list have more than one data element?
7. Why does the StackLinkedList class inherit the LinkedList class?
8. What happens when you push a new node onto a stack?

9. What is a queue linked list?
10. How does a queue linked list differ from an array queue?
11. What is the benefit of using a queue linked list?
12. Where are new nodes added to the queue?
13. Which node is removed from the queue when the dequeue() member method is called?
14. Can a node on a queue linked list have more than one data element?
15. What form of access is used to add and remove nodes from a queue?
16. Why does the QueueLinkedList class inherit the LinkedList class?
17. What happens when dequeue() is called?

2.10.9 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.
2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.
3. R. Sedgewick, “Algorithms in C++”, Pearson Education.
4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.
5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley.

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.11
AUTHOR : VISHAL SINGH

SORTING-I

(Linear Search, Binary Search, Bubble Sort, Insertion Sort, Selection Sort and Radix Sort)

2.11.1 Objective of the lesson

The primary purpose of this lesson is to provide an extensive set of examples illustrating the use of the data structures introduced in the preceding lesson and to show how the choice of structure for the underlying data profoundly influences the algorithms that perform a given task. Sorting is also a good example to show that such a task may be performed according to many different algorithms, each one

having certain advantages and disadvantages that have to be weighed against each other in the light of the particular application.

2.11.2 Introduction

Sorting is generally understood to be the process of rearranging a given set of objects in a specific order. The purpose of sorting is to facilitate the later search for members of the sorted set. As such it is an almost universally performed, fundamental activity. Objects are sorted in telephone books, in income tax files, in tables of contents, in libraries, in dictionaries, in warehouses, and almost everywhere that stored objects have to be searched and retrieved. Even small children are taught to put their things "in order", and they are confronted with some sort of sorting long before they learn anything about arithmetic. Hence, sorting is a relevant and essential activity, particularly in data processing. What else would be easier to sort than data! Nevertheless, our primary interest in sorting is devoted to the even more fundamental techniques used in the construction of algorithms. There are not many techniques that do not occur somewhere in connection with sorting algorithms. In particular, sorting is an ideal subject to demonstrate a great diversity of algorithms, all having the same purpose, many of them being optimal in some sense and most of them having advantages over others. It is therefore an ideal subject to demonstrate the necessity of performance analysis of algorithms. The example of sorting is moreover well suited for showing how a very significant gain in performance may be obtained by the development of sophisticated algorithms when obvious methods are readily available.

The dependence of the choice of an algorithm on the structure of the data to be processed -- an ubiquitous phenomenon -- is so profound in the case of sorting that sorting methods are generally classified into two categories, namely, sorting of arrays and sorting of (sequential) files. The two classes are often called internal and external sorting because arrays are stored in the fast, high-speed, random-access "internal" store of computers and files are appropriate on the slower, but more spacious "external" stores based on mechanically moving devices (disks and tapes). The importance of this distinction is obvious from the example of sorting numbered cards. Structuring the cards as an array corresponds to laying them out in front of the sorter so that each card is visible and individually accessible.

A search algorithm is an algorithm that evaluates a number of possible solutions to a given problem and returns the best one.

2.11.3 Linear Search

This method of searching for data in an array is straightforward easy to understand. To find a given item, begin your search at the start of the data collection and continue to look until you have either found the target of exhausted the search space. Clearly to employ this method you must first know where the data collection begins and the size of the area to search. Alternatively, a unique value could be used to signify the end of the search space. This method of searching is most often used on an array data structure whose upper and lower bounds are known.

The complexity of this type of search is O(N) because in the worst case all items in the search space will be examined. This type of search is (n/2) as, in the average case, one-half of the items in the search space will be examined before a match is found.

A linear search algorithm, also called **Sequential Search**, is checking every value of a data list, one at a time, in sequence until a match is found. Let's take for example, an algorithm that searches through an array of numbers and compares every value within it with a given match number.

C++ Implementation

```
#include<iostream>
using namespace std;
int linearsearch(int v[], int n, int value){
for(int I = 0; i<n;i++)
if (v[i]==value)
return 1;
return -1;
}
int main(){
int v[] ={1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
cout<<linearsearch(v, 9, 4)<<end1;
}
```

It's pretty straight forward. We have the num array made up of 10 elements and we check each of them in the for-loop to see if they match our given number (match). A linear search algorithm has a complexity of O(n).

2.11.4 Binary Search

A binary search algorithm is an algorithm used for finding a specific value in a sorted list. The implementation is simple : first we get the middle element of the array. If the middle element is equal with the value to be found, the algorithm stops. If not, we have the following :

- If the value to be found is less than the middle element, then perform the above operations for the part before the middle element (or the left side)
- If the value to be found is greater than the middle element, then perform the above operations for the part after than the middle element (or the right side).

Here's an example to help you understand better : let's say we have the following array {1, 2, 3, 4, 5, 6, 7, 8, 9, 10} and we want find number 7. First the middle it's taken that means 5 and compared with our given number, 7. Obviously 7>5, so now the array look like this : {6, 7, 8, 9, 10}. Now 7<8 and the next part like this {6, 7}. 7 =7=> element found.

C++ Iterative Implementation

```
#include<iostream>
Using namespace std;
        int binarysearch(int v[], int n, int value){
        int low = 0, high = n, mid;
                while(low <high)
{
        Mid = low + (high – low)/2;
                If(v[mid] = =value)
        Return 1;
                else if(v[mid]<value)
        low=mid+1;
                else
```

```
        high =mid-1;
}
If(low<n && v [low}==value)
        return 1;
else
        return-1;
}
int main(){
        int v[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
cout <<binarysearch(v, 10, 5)<<end1;
}
```

2.11.5 Bubble Sort

The bubble sort is a sort that uses a different scheme for finding the minimum (or maximum) value. Each iteration puts the smallest unsorted element into its correct place, but it also makes changes in the locations of the other elements in the array. The first iteration puts the smallest element in the array into the first array position. Starting with the last array element, we compare successive pairs of elements, swapping whenever the bottom element of the pair is smaller than the one above it. In this way the smallest element "bubbles up" to the top of the array. The next iteration puts the smallest element in the unsorted part of the array into the second array position, using the same technique.

As you look at the example in Figure 2.11.1, note that in addition to putting one element into its proper place, each iteration causes some intermediate changes in the array.

	values		values		values		values		values
[0]	36	[0]	6	[0]	6	[0]	6	[0]	6
[1]	24	[1]	36	[1]	10	[1]	10	[1]	10
[2]	10	[2]	24	[2]	36	[2]	12	[2]	12
[3]	6	[3]	10	[3]	24	[3]	36	[3]	24
[4]	12	[4]	12	[4]	12	[4]	24	[4]	36

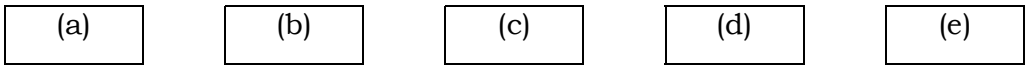


Figure 2.11.1 Example of bubble sort (sorted elements are shaded)

The basic algorithm for the bubble sort is

BubbleSort

Set current to the index of first item in the array

while more items in unsorted part of array

"Bubble up" the smallest item in the unsorted part,

causing intermediate swaps as needed

Shrink the unsorted part of the array by incrementing current

The structure of the loop is much like that of the selectionSort. The unsorted part of the array is the area from values[current] to values[SIZE - 1]. The value of current begins at 0, and we loop until current reaches SIZE - 1, with current incremented in each iteration. On entrance to each iteration of the loop body, the first current values are already sorted, and all the elements in the unsorted part of the array are greater than or equal to the sorted elements.

The inside of the loop body is different, however. Each iteration of the loop "bubbles up" the smallest value in the unsorted part of the array to the current position.

The algorithm for the bubbling task is

bubbleUp(startIndex, endIndex)

for index going from endIndex DOWNTO startIndex +1

if values[index] < values[index - 1]

Swap the value at index with the value at index -1

A snapshot of this algorithm is shown in Figure 2.11.2. We use the swap method as before. The code for methods bubbleUp and bubbleSort follows. The code can be tested using our test harness.

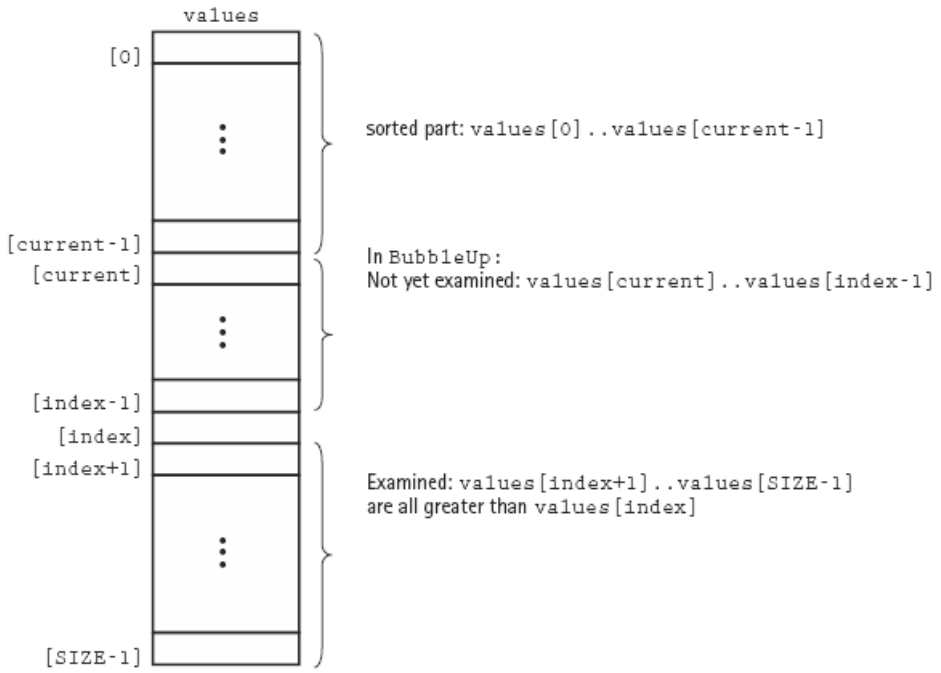


Figure 2.11.2 Snapshot of a bubble sort

```
static void bubbleUp(int startIndex, int endIndex) {
// Post: Adjacent pairs that are out of order have been switched
// between values[startIndex]..values[endIndex] beginning at
// values[endIndex]
        for (int index = endIndex; index > startIndex; index--)
                if (values[index] < values[index - 1])
                        swap(index, index - 1);
}
static void bubbleSort(){
// Post: The elements in the array values are sorted
        int current = 0;
        while (current < SIZE - 1){
                bubbleUp(current, SIZE -1);
                current++;
        }
```

```
}
```

Analyzing Bubble Sort

To analyze the work required by bubbleSort is easy. It is the same as for the straight selection sort algorithm. The comparisons are in bubbleUp, which is called N-1 times.

There are N-1 comparisons the first time, N-2 comparisons the second time, and so on. Therefore, bubbleSort and selectionSort (explain later in this lesson) require the same amount of work in terms of the number of comparisons. bubbleSort does more than just make comparisons though; selectionSort has only one data swap per iteration, but bubbleSort may do many additional data swaps. By reversing out-of-order pairs of data as they are noticed, the method might get the array in order before N-1 calls to bubbleUp. However, this version of the bubble sort makes no provision for stopping when the array is completely sorted. Even if the array is already in sorted order when bubbleSort is called, this method continues to call bubbleUp (which changes nothing) N-1 times.

We could quit before the maximum number of iterations if bubbleUp returns a boolean flag, to tell us when the array is sorted. Within bubbleUp, we initially set a variable sorted to true; then in the loop, if any swaps are made, we reset sorted to false. If no elements have been swapped, we know that the array is already in order. Now the bubble sort only needs to make one extra call to bubbleUp when the array is in order. This version of the bubble sort is as follows:

```
boolean bubbleUp2(int startIndex, int endIndex) {
// Post: Adjacent pairs that are out of order have been switched
// between values[startIndex]..values[endIndex] beginning at
// values[endIndex]
// Returns false if a swap was made; otherwise, true
        boolean sorted = true;
        for (int index = endIndex; index > startIndex; index--)
        if (values[index] < values[index - 1]){
                swap(index, index - 1);
                sorted = false;
        }
```

```
        return sorted;
    }
    void shortBubble(){
    // Post: The elements in the array values are sorted by key
    // The process stops as soon as values is sorted
        int current = 0;
        boolean sorted = false;
        while (current < SIZE - 1 && !sorted){
            sorted = bubbleUp2(current, SIZE - 1);
            current++;
        }
    }
```

The analysis of shortBubble is more difficult. Clearly, if the array is already sorted to begin with, one call to bubbleUp tells us so. In this best-case scenario, shortBubble is O(N); only N- 1 comparisons are required for the sort. What if the original array was actually sorted in descending order before the call to shortBubble? This is the worst possible case: shortBubble requires as many comparisons as bubbleSort and selectionSort, not to mention the "overhead"—all the extra swaps and setting and resetting the sorted flag. Can we calculate an average case? In the first call to bubbleUp, when current is 0, there are SIZE-1 comparisons; on the second call, when current is 1, there are SIZE-2 comparisons. The number of comparisons in any call to bubbleUp is SIZE-current-1. If we let N indicate SIZE and K indicate the number of calls to bubbleUp executed before shortBubble finishes its work, the total number of comparisons required is

$$(N-1) + (N-2) + (N-3) + \ldots + (N-K)$$

$$1^{st} \text{ call } 2^{nd} \text{ call } 3^{rd} \text{ call } \ldots K^{th} \text{ call}$$

A little algebra changes this to

$$(2KN-2K^2-K)/2$$

In Big-O notation, the term that is increasing the fastest relative to N is 2KN. We know that K is between 1 and N-1. On average, over all possible input orders, K is proportional to N. Therefore, 2KN is proportional to N2; that is, the shortBubble algorithm is also $O(N^2)$.

Due to the extra intermediate swaps performed by bubble sort, it can quickly sort an array that is "almost" sorted. If the shortBubble variation is used, bubble sort can be very efficient for this situation.

2.11.6 Insertion Sort

The principle of the insertion sort is quite simple: Each successive element in the array to be sorted is inserted into its proper place with respect to the other, already sorted elements. As with the previous sorts, we divide our array into a sorted part and an unsorted part. (Unlike the previous sorts, there may be values in the unsorted part that are less than values in the sorted part.)

Initially, the sorted portion contains only one element: the first element in the array. Now we take the second element in the array and put it into its correct place in the sorted part; that is, values[0] and values[1] are in order with respect to each other. Now the value in values[2] is put into its proper place, so values[0]..values[2] are in order with respect to each other. This process continues until all the elements have been sorted. Figure 2.11.3 illustrates this process, which we describe in the following algorithm, and Figure 2.11.4 shows a snapshot of the algorithm.

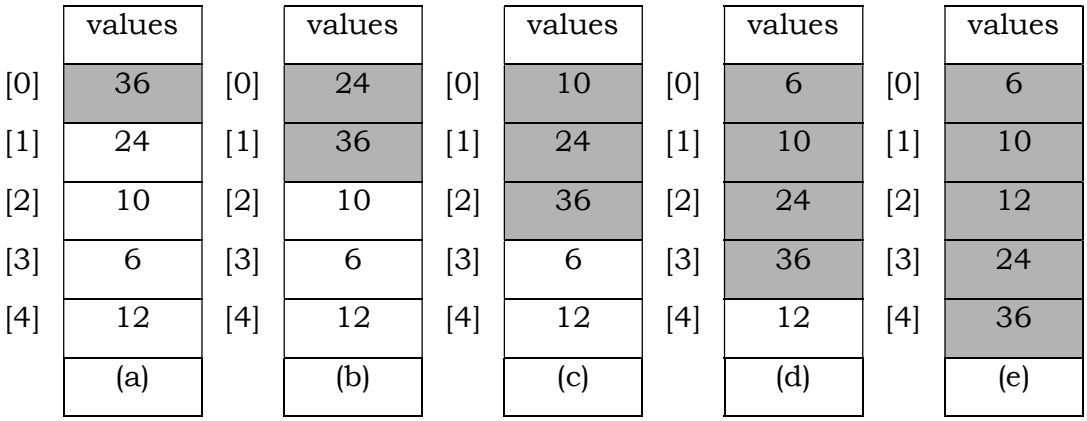


Figure 2.11.3 Example of the insertion sort algorithm

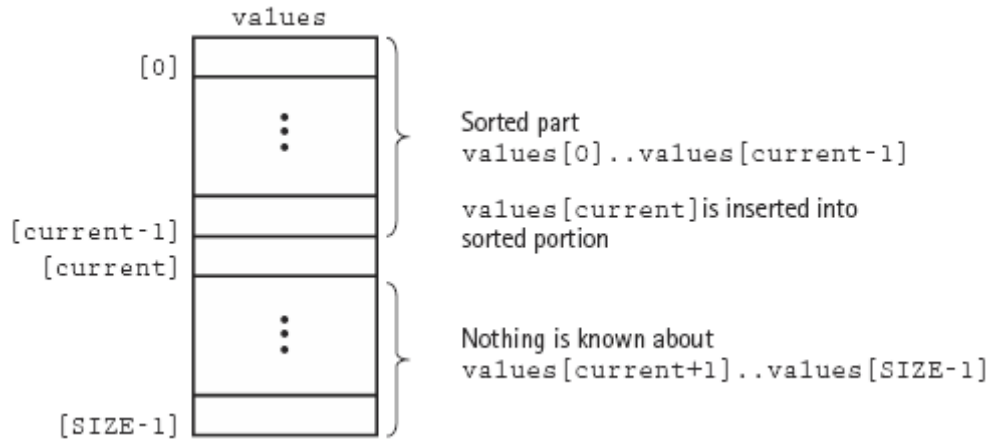


Figure 2.11.4 A snapshot of the insertion sort algorithm

Our strategy is to search for the insertion point from the beginning of the array and shift the elements from the insertion point down one slot to make room for the new element. We can combine the searching and shifting by beginning at the end of the sorted part of the array. We compare the item at values[current] to the one before it. If it is less, we swap the two items. We then compare the item at values[current-1] to the one before it, and swap if necessary. The process stops when the comparison shows that the values are in order or we have swapped into the first place in the array.

insertionSort

```
for count going from 1 through SIZE - 1
        insertItem(0, count)
```

InsertItem(startIndex, endIndex)

```
Set finished to false
Set current to endIndex
Set moreToSearch to true
while moreToSearch AND NOT finished
        if values[current] < values[current - 1]
                swap(values[current], values[current - 1])
                Decrement current
                Set moreToSearch to (current does not equal startIndex)
```

```
        else
                Set finished to true
```

Here are the coded versions of insertItem and insertionSort.

```
        void insertItem(int startIndex, int endIndex) {
        // Post: values[0]..values[endIndex] are now sorted
                boolean finished = false;
                int current = endIndex;
                boolean moreToSearch = true;
                while (moreToSearch && !finished){
                        if (values[current] < values[current - 1]){
                                swap(current, current - 1);
                                current--;
                                moreToSearch = (current != startIndex);
                        }
                        else
                                finished = true;
                }
        }
        static void insertionSort(){
        // Post: The elements in the array values are sorted by key
                for (int count = 1; count < SIZE; count++)
                        insertItem(0, count);
        }
```

Analyzing Insertion Sort

The general case for this algorithm mirrors the selectionSort and the bubbleSort, so the general case is O(N2). But like shortBubble, insertionSort has a best case: The data are already sorted in ascending order. When the data are in ascending order, insertItem is called N times, but only one comparison is made each time and no

swaps are necessary. The maximum number of comparisons is made only when the elements in the array are in reverse order.

If we know nothing about the original order of the data to be sorted, selectionSort, shortBubble and insertionSort are all O(N2) sorts and are very time consuming for sorting large arrays. Thus, we need sorting methods that work better when N is large.

2.11.7 Selection Sort If you were handed a list of names and asked to put them in alphabetical order, you might use this general approach:

1. Find the name that comes first in alphabetical order and write it on a second sheet of paper.
2. Cross the name out on the original list.
3. Continue this cycle until all the names on the original list have been crossed out and written onto the second list, at which point the second list is sorted.

This algorithm is simple to translate into a computer program, but it has one drawback: It requires space in memory to store two complete lists. Although we have not talked a great deal about memory-space considerations, this duplication is clearly wasteful. A slight adjustment to this manual approach does away with the need to duplicate space, however. As you cross a name off the original list, a free space opens up. Instead of writing the minimum value on a second list, you can exchange it with the value currently in the position where the crossed-off item should go. Our "by-hand list" is represented in an array. Let's look at an example—sorting the five-element array shown in Figure 2.11.6.5(a). Because of this algorithm's simplicity, it is usually the first sorting method that students learn.

SelectionSort

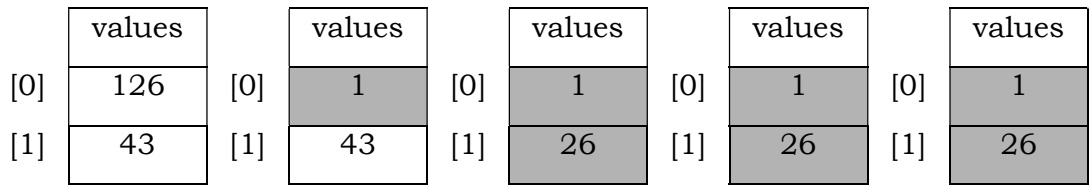
Set current to the index of first item in the array

while more items in unsorted part of array

Find the index of the smallest unsorted item

Swap the current item with the smallest unsorted one

Shrink the unsorted part of the array by incrementing current



	(a)		(b)		(c)		(d)		(e)
[2]	26	[2]	26	[2]	43	[2]	43	[2]	43
[3]	1	[3]	126	[3]	126	[3]	126	[3]	113
[4]	113	[4]	113	[4]	113	[4]	113	[4]	126

Figure 2.11.5 Example of a straight selection sort (sorted elements are shaded)

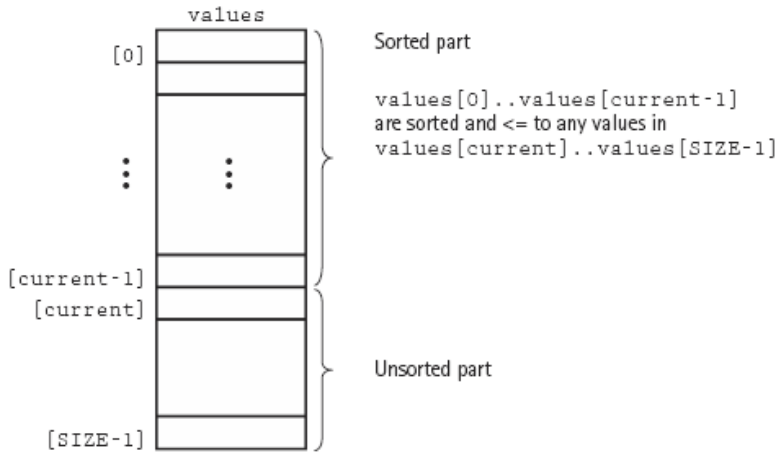


Figure 2.11.6 A snapshot of the selection sort algorithm

We implement the algorithm with a method selectionSort that is part of our Sorts class. This method sorts the values array, declared in that class and therefore has access to the SIZE constant that indicates the number of elements in the array. Within the selectionSort method we use a variable, current, to mark the beginning of the unsorted part of the array. This means that the unsorted part of the array goes from index current to index SIZE - 1. We start out by setting current to the index of the first position (0).

The main sort processing is in a loop. In each iteration of the loop body, the smallest value in the unsorted part of the array is swapped with the value in the current location. After the swap, current is in the sorted part of the array, so we shrink the size of the unsorted part by incrementing current. The loop body is now complete. Back at the top of the loop body, the unsorted part of the array goes from the (now incremented) current index to position SIZE - 1. We know that every value in the unsorted part is greater than or equal to any value in the sorted part of the array.

As long as current <= SIZE - 1, the unsorted part of the array (values[current] .. values[SIZE - 1]) contains values. In each iteration of the loop body, current is incremented, shrinking the unsorted part of the array. When current = SIZE - 1, the

"unsorted" part contains only one element, and we know that this value is greater than or equal to any value in the sorted part. So the value in values[SIZE - 1] is in its correct place, and we are done. The condition for the while loop is current < SIZE-1. A snapshot of the selection sort algorithm is illustrated in Figure 2.11.2.

Now all we have to do is locate the smallest value in the unsorted part of the array. Let's write a method to do this task. The minIndex method receives first and last indexes of the unsorted part and returns the index of the smallest value in this part of the array.

```
minIndex(startIndex, endIndex): return int
        Set indexOfMin to startIndex
        for index going from startIndex + 1 to endIndex
                if values[index] < values[indexOfMin]
                        Set indexOfMin to index
return indexOfMin
```

Now that we know where the smallest unsorted element is, we swap it with the element at index current. We use the swap method that is part of our test harness. Here is the code for the minIndex and selectionSort methods. Since they are placed directly in our test harness class, a class with a main method, they are declared as static methods.

```
int minIndex(int startIndex, int endIndex)
// Post: Returns the index of the smallest value in
// values[startIndex]..values[endIndex]
{
        int indexOfMin = startIndex;
        for (int index = startIndex + 1; index <= endIndex; index++)
                if (values[index] < values[indexOfMin])
                        indexOfMin = index;
        return indexOfMin;
}
void selectionSort()
// Post: The elements in the array values are sorted
```

```
{
        int endIndex = SIZE - 1;
        for (int current = 0; current < endIndex; current++)
                swap(current, minIndex(current, endIndex));
}
```

Analyzing Selection Sort

Now let's try measuring the amount of "work" required by this algorithm. We describe the number of comparisons as a function of the number of items in the array, i.e., SIZE. To be concise, in this discussion we refer to SIZE as N.

The comparison operation is in the minIndex method. We know from the loop condition in the selectionSort method that minIndex is called N-1 times. Within minIndex, the number of comparisons varies, depending on the values of startIndex and endIndex:

```
for (int index = startIndex + 1; index <= endIndex; index++)
        if (values[index] < values[indexOfMin])
                indexOfMin = index;
```

In the first call to minIndex, startIndex is 0 and endIndex is SIZE - 1, so there are N - 1 comparisons; in the next call there are N - 2 comparisons and so on, until in the last call, when there is only one comparison. The total number of comparisons is (N - 1) + (N - 2) + (N - 3) + . . . + 1 = N(N - 1)/2

Table 2.11.1 Number of Comparisons Required to Sort Arrays of Different Sizes Using Selection Sort

Number of Items	Number of Comparisons
10	45
20	190
100	4950
1000	499500
10000	49995000

To accomplish our goal of sorting an array of N elements, the straight selection sort requires N(N - 1)/2 comparisons. Note that the particular arrangement of values in the array does not affect the amount of work done at all. Even if the array is in sorted order before the call to selectionSort, the method still makes N(N-1)/2 comparisons.

Table 2.11.1 shows the number of comparisons required for arrays of various sizes. Note that doubling the array size roughly quadruples the number of comparisons. How do we describe this algorithm in terms of Big-O? If we express N(N -1)/2 as 1/2N2 - 1/2N, it is easy to see. In Big-O notation we only consider the term $1/2N^2$, because it increases fastest relative to N. Further, we ignore the constant, 1/2, making this algorithm $O(N^2)$. This means that, for large values of N, the computation time is approximately proportional to N^2. Looking back at the previous table, we see that multiplying the number of elements by 10 increases the number of comparisons by a factor of more than 100; that is, the number of comparisons is multiplied by approximately the square of the increase in the number of elements.

Looking at this chart makes us appreciate why sorting algorithms are the subject of so much attention: Using selectionSort to sort an array of 1,000 elements requires almost a half million comparisons!

The identifying feature of a selection sort is that, on each pass through the loop, one element is put into its proper place. In the straight selection sort, each iteration finds the smallest unsorted element and puts it into its correct place. If we had made the helper method find the largest value instead of the smallest, the algorithm would have sorted in descending order. We could also have made the loop go down from SIZE - 1 to 1, putting the elements into the end of the array first. All these are variations on the straight selection sort. The variations do not change the basic way that the minimum (or maximum) element is found.

2.11.8 Radix Sort (Bucket sort)

Unlike the sorting algorithms described previously radix sort uses buckets to sort items, each bucket holds items with a particular property called a key. Normally a bucket is a queue, each time radix sort is performed these buckets are emptied starting the smallest key bucket to the largest. When looking at items within a list to sort we do so by isolating a specific key, e.g. in the example we are about to show we have a maximum of three keys for all items, that is the highest key we need to look at is hundreds. Because we are dealing with, in this example base 10 numbers we have at any one point 10 possible key values 0 - 9 each of which has their own bucket. Before we show you this first simple version of radix sort let us clarify what

we mean by isolating keys. Given the number 102 if we look at the first key, the ones then we can see we have two of them, progressing to the next key - tens we can see that the number has zero of them, finally we can see that the number has a single hundred. The number used as an example has in total three keys:

1. Ones (2)
2. Tens (0)
3. Hundreds (1)

For further clarification what if we wanted to determine how many thousands the number 102 has? Clearly there are none, but often looking at a number as final like we often do it is not so obvious so when asked the question how many thousands does 102 have you should simply pad the number with a zero in that location, e.g. 0102 here it is more obvious that the key value at the thousands location is zero.

The last thing to identify before we actually show you a simple implementation of radix sort that works on only positive integers, and requires you to specify the maximum key size in the list is that we need a way to isolate a specific key at any one time. The solution is actually very simple, but its not often you want to isolate a key in a number so we will spell it out clearly here. A key can be accessed from any integer with the following expression:

key = (number / keyToAccess) % 10.

As a simple example lets say that we want to access the tens key of the number 1290, the tens column is key 10 and so after substitution yields key = (1290 / 10) % 10 = 9. The next key to look at for a number can be attained by multiplying the last key by ten working left to right in a sequential manner. The value of key is used in the following algorithm to work out the index of an array of queues to enqueue the item into.

```
1)  algorithm Radix(list, maxKeySize)
2)  //List represents the list of numbers to be sorted
3)  //maxKeySize represents the largest key size in the list
4)  //Declare 10 queues
5)  Queue queues[10]
6)  indexOfKey = 1
7)  for I = 0 to maxKeySize - 1
8)  foreach item in list
```

```
9)      queues[GetQueueIndex(item, indexOfKey)].Enqueue(item)
10)     end foreach
11)     list = CollapseQueues(queues)
12)     ClearQueues(queues)
13)     indexOfKey = indexOfKey * 10
14)     end for
15)     return list
16)     end Radix
```

Following is the implementation of the algorithm described above operating on the list whose members are 90; 12; 8; 791; 123; and 61.

Iteration 1: Key is ones

Bucket positions

	791								
90	61	12	123					8	
0	1	2	3	4	5	6	7	8	9

After bringing out all values from bucket the resulting arrangement of values will be

90; 61; 791; 12; 123; 8

Iteration 2: Key is tens

Bucket positions

									90
8	12	123				61			791

0	1	2	3	4	5	6	7	8	9

After bringing out all values from bucket the resulting arrangement of values will be

8; 12; 123; 61; 90; 791

Iteration 3: Key is hundreds

Bucket positions

8									
12									
61									
90	123						791		
0	1	2	3	4	5	6	7	8	9

Finally, after bringing out all values from bucket the resulting arrangement of values will be 8; 12; 61; 90; 123; 791

2.11.9 Summary

Throughout this lesson we have seen many different algorithms for sorting lists, some are very efficient some are not. Selecting the correct sorting algorithm is usually denoted purely by efficiency, e.g. you would always choose merge sort over shell sort and so on. There are also other factors to look at though and these are based on the actual implementation. Some algorithms are very nicely expressed in a recursive fashion, however these algorithms ought to be pretty efficient, e.g. implementing a linear, quadratic, or slower algorithm using recursion would be a very bad idea.

2.11.10 Questions

1. Write and explain the bubble sort algorithm.
2. Write and explain insertion sort algorithm.
3. Write and explain selection sort algorithm.
4. Write the complexity of the sorting algorithms discussed in this lesson for average and worst case.

5. Write the radix sort algorithm for English words considering 26 buckets representing each alphabet.

2.11.11 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, “Data Structures Using C++”, PHI.

2. M. A. Weiss, “Data Structures and Algorithm Analysis in C++”, Pearson Education.

3. R. Sedgewick, “Algorithms in C++”, Pearson Education.

4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.

5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley.

B.A. PART-II **PAPER : BAP-201**

SEMESTER-III **C PROGRAMMING AND DATA STRUCTURES**

LESSON NO. 2.12 **AUTHOR : VISHAL SINGH**

SORTING-II
(Merge Sort, Quick Sort, Heap Sort)

2.12.1 Objective of the lesson

In this lesson we shall learn more complex but efficient sorting algorithms based on 'Divide and conquer' technique.

2.12.2 Introduction

Considering how rapidly N^2 grows as the size of the array increases, can't we do better? We note that N^2 is a lot larger than (1/2N)2 + (1/2N)2. If we could cut the array into two pieces, sort each segment and then merge the two back together, we should end up sorting the entire array with a lot less work. An example of this approach is shown in Figure 2.12.1.

The idea of "divide and conquer" has been applied to the sorting problem in different ways, resulting in a number of algorithms that can do the job much more

efficiently than $O(N^2)$. In fact, there is a category of sorting algorithms that are $O(N\log_2 N)$. We examine three of these algorithms here: mergeSort, quickSort and heapSort. As you might guess, the efficiency of these algorithms is achieved at the expense of the simplicity seen in the straight selection, bubble, and insertion sorts.

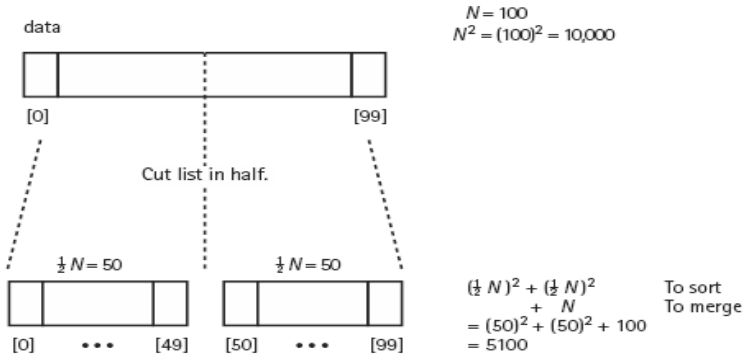


Figure 2.12.1 Rationale for divide-and-conquer sorts

24.3 Merge Sort

The merge sort algorithm is taken directly from the idea presented above.

mergeSort

Cut the array in half

Sort the left half

Sort the right half

Merge the two sorted halves into one sorted array

Merging the two halves together is a O(N) task: We merely go through the sorted halves, comparing successive pairs of values (one in each half) and putting the smaller value into the next slot in the final solution. Even if the sorting algorithm used for each half is O(N2), we should see some improvement over sorting the whole array at once.

Actually, because mergeSort is itself a sorting algorithm, we might as well use it to sort the two halves. That's right—we can make mergeSort a recursive method and let it call itself to sort each of the two subarrays:

mergeSort—Recursive

Cut the array in half

mergeSort the left half

mergeSort the right half

Merge the two sorted halves into one sorted array

This is the general case, of course. What is the base case, the case that does not involve any recursive calls to mergeSort? If the "half " to be sorted doesn't have more than one element, we can consider it already sorted and just return. Let's summarize mergeSort in the format we used for other recursive algorithms. The initial method call would be mergeSort(0, SIZE – 1).

Cutting the array in half is simply a matter of finding the midpoint between the first and last indexes:

```
middle = (first + last) / 2;
```

Then, in the smaller-caller tradition, we can make the recursive calls to mergeSort:

```
mergeSort(first, middle);

mergeSort(middle + 1, last);
```

So far this is simple enough. Now we only have to merge the two halves and we're done.

Merging the Sorted Halves

Obviously, all the serious work is in the merge step. Let's first look at the general algorithm for merging two sorted arrays and then we can look at the specific problem of our subarrays.

To merge two sorted arrays, we compare successive pairs of elements, one from each array, moving the smaller of each pair to the "final" array. We can stop when one array runs out of elements and then move all the remaining elements from the other array to the final array. Figure 2.12.2 illustrates the general algorithm. We use a similar approach in our specific problem, in which the two "arrays" to be merged are actually subarrays of the original array (Figure 2.12.3). Just as in Figure 2.12.2, where we merged array1 and array2 into a third array, we need to merge our two subarrays into some auxiliary structure. We only need this structure, another array, temporarily. After themerge step, we can copy the now-sorted elements back into the original array. The whole process is shown in Figure 2.12.4.

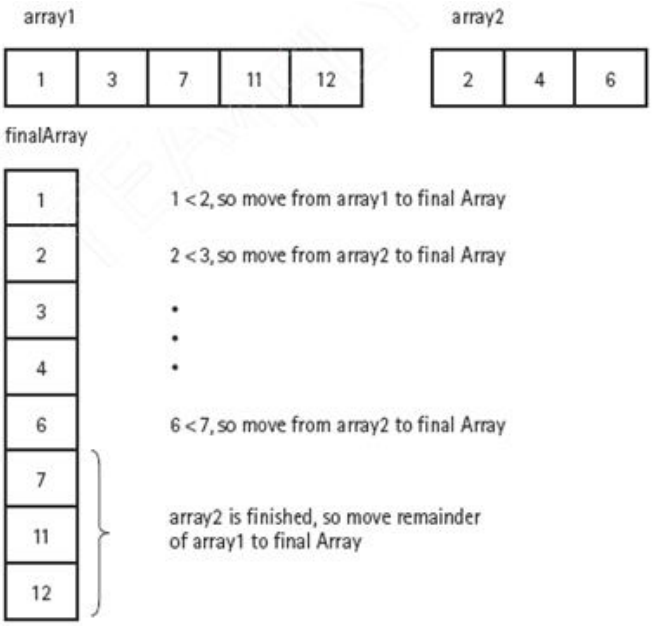


Figure 2.12.2 Strategy for merging two sorted arrays

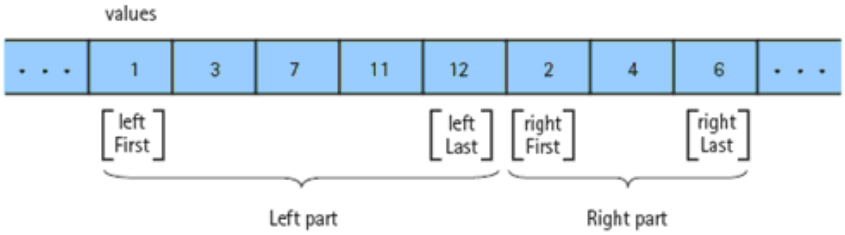


Figure 2.12.3 Two subarrays

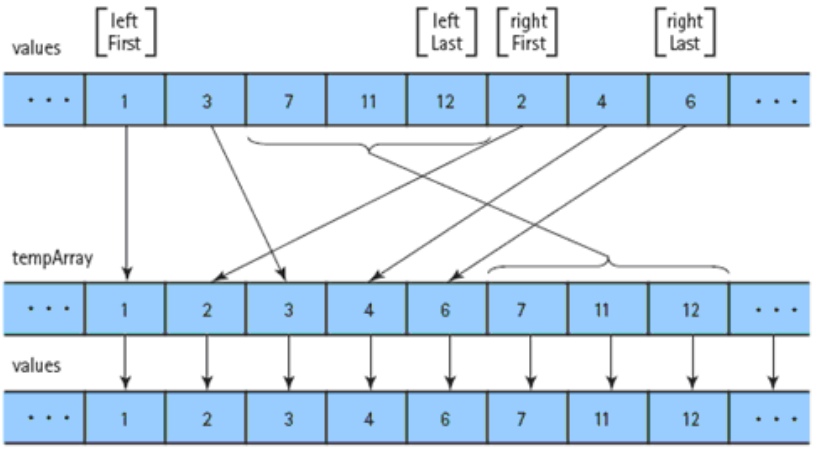


Figure 2.12.4 Merging sorted halves

Let's specify a method, merge, to do this task:

```
merge(int leftFirst, int leftLast, int rightFirst, int rightLast)
```

Here is the algorithm for Merge:

```
merge (leftFirst, leftLast, rightFirst, rightLast)
(uses a local array, tempArray)
Set saveFirst to leftFirst // To know where to copy back
Set index to leftFirst
while more items in left half AND more items in right half
        if values[leftFirst] < values[rightFirst]
                Set tempArray[index] to values[leftFirst]
                Increment leftFirst
        else
                Set tempArray[index] to values[rightFirst]
                Increment rightFirst
Increment index
Copy any remaining items from left half to tempArray
Copy any remaining items from right half to tempArray
Copy the sorted elements from tempArray back into values
```

In the coding of method merge, we use leftFirst and rightFirst to indicate the "current" position in the left and right halves, respectively. Because these are values of the primitive type int and not objects, copies of these parameters are passed to method merge, rather than references to the parameters. These copies are changed in the method; but changing the copies does not affect the original values. Note that both of the "copy any remaining items" loops are included. During the execution of this method, one of these loops never executes. Can you explain why?

```
void merge (int leftFirst, int leftLast, int rightFirst, int rightLast)
// Post: values[leftFirst]..values[leftLast] and
// values[rightFirst]..values[rightLast] have been merged.
// values[leftFirst]..values[rightLast] are now sorted
```

```
{
        int tempArray[SIZE];
        int index = leftFirst;
        int saveFirst = leftFirst;
        while ((leftFirst <= leftLast) && (rightFirst <= rightLast)){
                if (values[leftFirst] < values[rightFirst]){
                        tempArray[index] = values[leftFirst];
                        leftFirst++;
                }
                else{
                        tempArray[index] = values[rightFirst];
                        rightFirst++;
                }
                index++;
        }
        while (leftFirst <= leftLast) {// Copy remaining items from left half
                tempArray[index] = values[leftFirst];
                leftFirst++;
                index++;
        }
        while (rightFirst <= rightLast) {// Copy remaining items from right half
                tempArray[index] = values[rightFirst];
                rightFirst++;
                index++;
        }
        for (index = saveFirst; index <= rightLast; index++)
                values[index] = tempArray[index];
}
```

As we said, most of the work is in the merge task. The actual mergeSort method is short and simple:

```
void mergeSort(int first, int last)
// Post: The elements in values are sorted by key
{
    if (first < last){
        int middle = (first + last) / 2;
        mergeSort(first, middle);
        mergeSort(middle + 1, last);
        merge(first, middle, middle + 1, last);
    }
}
```

Analyzing mergeSort

The mergeSort method splits the original array into two halves. It first sorts the first half of the array, using the divide and conquer approach; then it sorts the second half of the array using the same approach; then it merges the two halves. To sort the first half of the array it follows the same approach, splitting and merging. During the sorting process the splitting and merging operations are all intermingled. However, analysis is simplified if we imagine that all of the splitting occurs first—we can view the process this way without affecting the correctness of the algorithm.

We view the mergeSort algorithm as continually dividing the original array (of size N) in two, until it has created N one element subarrays. Figure 2.12.5 shows this point of view for an array with an original size of 16. The total work needed to divide the array in half, over and over again until we reach subarrays of size 1 is O(N). After all, we end up with N subarrays of size 1.

Each subarray of size 1 is obviously a sorted subarray. The real work of the algorithm involves merging the smaller sorted subarrays back into the larger sorted subarrays.

To merge two sorted subarrays of size X and size Y into a single sorted subarray using the merge operation requires O(X + Y) steps. We can see this because each time through the while loops of the merge method we either advance the leftFirst

index or the rightFirst index by 1. Since we stop processing when these indexes become greater than their "last" counterparts, we know that we take a total of (leftLast – leftFirst + 1) + (rightLast – rightFirst + 1) steps. This expression represents the sum of the lengths of the two subarrays being processed.

How many times must we perform the merge operation? And what are the sizes of the subarrays involved? Let's work from the bottom up. The original array of size N is eventually split into N subarrays of size 1. Merging two of those subarrays, into a subarray of size 2, requires O(1 + 1) = O(2) steps based on the analysis of the preceding paragraph. That is, it requires a small constant number of steps in each case. But, we must perform this merge operation a total of 1⁄2N times (we have N one-element subarrays and we are merging them two at a time). So the total number of steps to create all the sorted two-element subarrays is O(N). Now we repeat this process to create four-element subarrays. It takes four steps to merge two two-element subarrays.

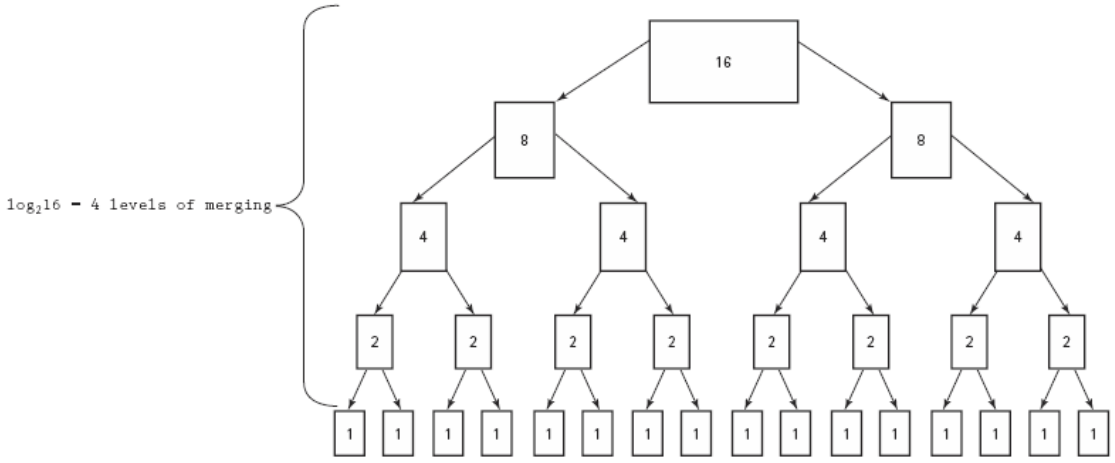


Figure 2.12.5 Analysis of Merge Sort algorithm with N = 16

We must perform this merge operation a total of 1⁄4N times (we have 1⁄2N two-element subarrays and we are merging them two at a time). So the total number of steps to create all the sorted four-element subarrays is also O(N) (4 * 1⁄4N = N). The same reasoning leads us to conclude that each of the other levels of merging also requires O(N) steps—at each level the sizes of the subarrays double, but the number of subarrays is cut in half, balancing out.

We now know that it takes O(N) total steps to perform merging at each "level" of merging. How many levels are there? The number of levels of merging is equal to the

number of times we can split the original array in half. If the original array is size N, we have Log_2N levels. (This is just like the analysis of the binary search algorithm.) For example, in Figure 2.12.5 the size of the original array is 16 and the number of levels of merging is 4. Since we have Log_2N levels, and we require O(N) steps at each level, the total cost of the merge operation is: $O(NLog_2N)$. And since the splitting phase was only O(N), we conclude that Merge Sort algorithm is $O(NLog_2N)$. Table 2.12.1 illustrates that, for large values of N, $O(NLog_2N)$ is a big improvement over $O(N^2)$.

The disadvantage of mergeSort is that it requires an auxiliary array that is as large as the original array to be sorted. If the array is large and space is a critical factor, this sort may not be an appropriate choice. Next we discuss two sorts that move elements around in the original array and do not need an auxiliary array.

Table 2.12.1 Comparing N^2 and N Log_2 N

N	Log_2N	N^2	$NLog_2N$
32	5	1,024	160
64	6	4,096	384
128	7	16,384	896
256	8	65,536	2,048
512	9	262,144	4,608
1024	10	1,048,576	10,240
2048	11	4,194,304	22,528
4096	12	16,777,216	49,152

2.12.4 Quick Sort

Like Merge Sort, Quick Sort is a divide-and-conquer algorithm, which is inherently recursive. If you were given a large stack of final exams to sort by name, you might use the following approach: pick a splitting value, say, L and divide the stack of tests into two piles, A–L and M–Z. (Note that the two piles do not necessarily contain the same number of tests.) Then take the first pile and subdivide it into two piles, A–F and G–L. The A–F pile can be further broken down into A–C and D–F. This division process goes on until the piles are small enough to be easily sorted. The same process is applied to the M–Z pile.

Eventually all the small sorted piles can be collected one on top of the other to produce a sorted set of tests. (See Figure 2.12.6)

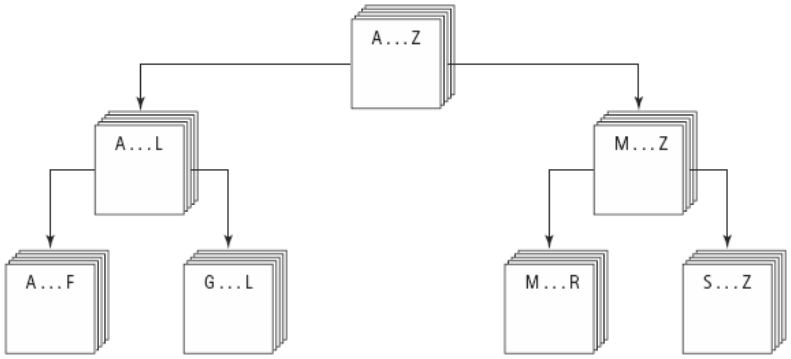


Figure 2.12.6 Ordering a list using the Quick Sort algorithm

This strategy is recursive—on each attempt to sort the pile of tests, the pile is divided, and then the same approach is used to sort each of the smaller piles (a smaller case). This process goes on until the small piles do not need to be further divided (the base case). The parameter list of the quickSort method reflects the part of the list that is currently being processed; we pass the first and last indexes that define the part of the array to be processed on this call. The initial call to quickSort is :

quickSort(0, SIZE – 1);

quickSort

if there is more than one element in values[first]..values[last]

Select splitVal

Split the array so that

values[first]..values[splitPoint – 1] <= splitVal

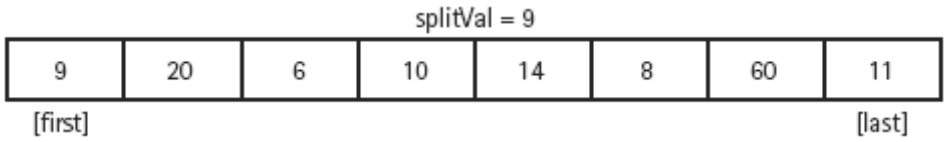
values[splitPoint] = splitVal

values[splitPoint + 1]..values[last] > splitVal

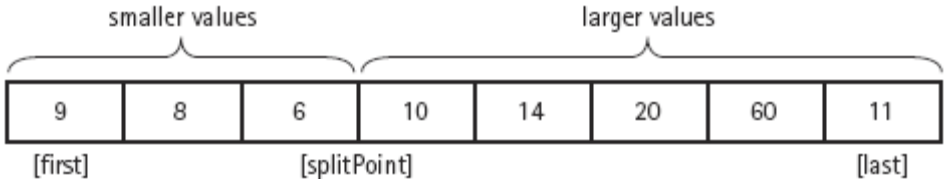
quickSort the left half

quickSort the right half

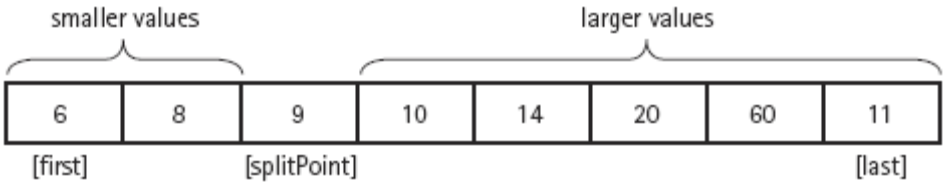
How do we select splitVal? One simple solution is to use the value in values[first] as the splitting value.



We create a helper method split, to rearrange the array elements as planned. After the call to split, all the items less than or equal to splitVal are on the left side of the array and all of those greater than splitVal are on the right side of the array.



The two "halves" meet at splitPoint, the index of the last item that is less than or equal to splitVal. Note that we don't know the value of splitPoint until the splitting process is complete. Its value is returned by split. We can then swap splitVal with the value at splitPoint.



Our recursive calls to quickSort use this index (splitPoint) to reduce the size of the problem in the general case.

quickSort(first, splitPoint – 1) sorts the left "half " of the array. quick-Sort(splitPoint + 1, last) sorts the right "half " of the array. (The "halves" are not necessarily the same size.) splitVal is already in its correct position in values[splitPoint].

What is the base case? When the segment being examined has less than two items, we do not need to go on. So "there is more than one item in values[first]..values[last]" can be translated into "if (first < last)". We can now code our quickSort method.

```
void quickSort(int first, int last){
    if (first < last){
        int splitPoint;
```

```
            splitPoint = split(first, last);
            // values[first]..values[splitPoint – 1] <= splitVal
            // values[splitPoint] = splitVal
            // values[splitPoint+1]..values[last] > splitVal
            quickSort(first, splitPoint – 1);
            quickSort(splitPoint + 1, last);
        }
    }
```

Now we must develop our splitting algorithm. We must find a way to get all of the elements equal to or less than splitVal on one side of splitVal and the elements greater than splitVal on the other side.

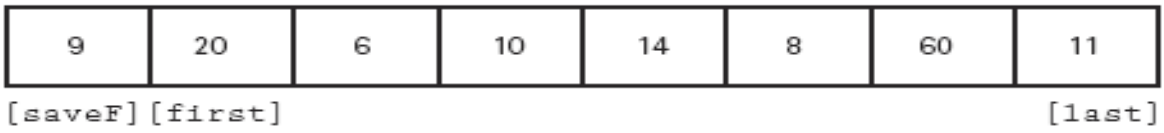
We do this by moving the indexes, first and last, toward the middle of the array, looking for items that are on the wrong side of the split point and swapping them (Figure 2.12.7). While this is proceeding, the splitVal remains in the first position of the subarray being processed. As a final step, we swap it with the value at the split Point; therefore, we save the original value of first in a local variable, saveF. (See Figure 2.12.7a.)

We start out by moving first to the right, toward the middle, comparing values[first] to splitVal. If values[first] is less than or equal to splitVal, we keep incrementing first; otherwise we leave first where it is and begin moving last toward the middle. (See Figure 2.12.7b.)

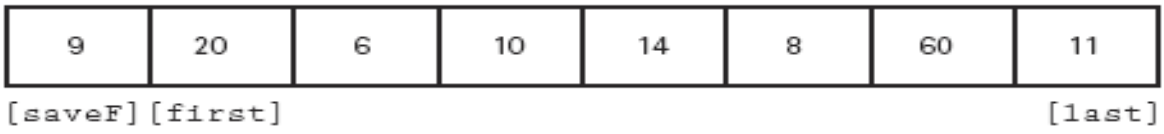
Now values[last] is compared to splitVal. If it is greater, we continue decrementing last; otherwise, we leave last in place. (See Figure 2.12.7c.) At this point, it is clear that values[last] and values[first] are each on the wrong side of the array.

Note that the elements to the left of values[first] and to the right of values[last] are not necessarily sorted; they are just on the correct side of the array with respect to splitVal. To put values[first] and values[last] into their correct sides, we merely swap them, then increment first and decrement last. (See Figure 2.12.7d.) Now we repeat the whole cycle, incrementing first until we encounter a value that is greater than splitVal, then decrementing last until we encounter a value that is less than or equal to splitVal. (See Figure 2.12.7e.)

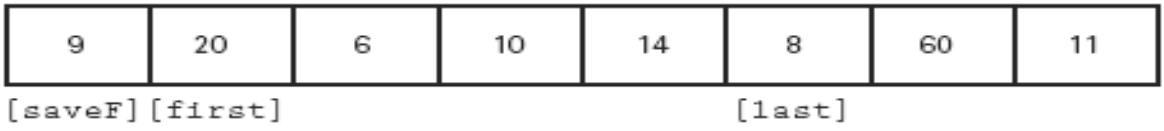
(a) Initialization. Note that `splitVal = values[first] = 9`.



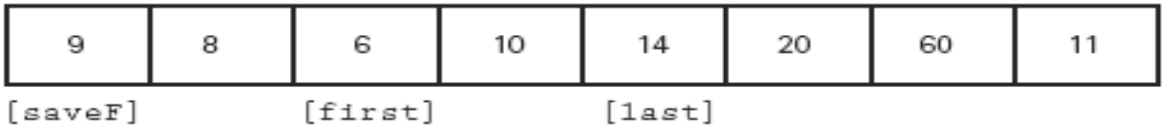
(b) Increment `first` until `values[first] > splitVal`



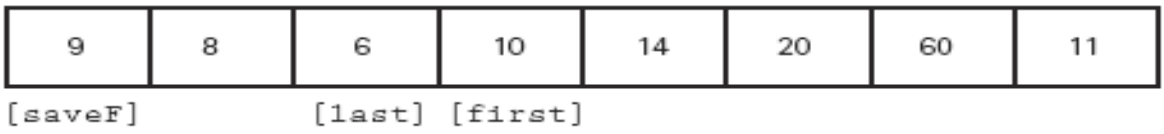
(c) Decrement `last` until `values[last] <= splitVal`



(d) Swap `values[first]` and `values[last]`; move `first` and `last` toward each other



(e) Increment `first` until `values[first] > splitVal` or `first > last`.
Decrement `last` until `values[last] <= splitVal` or `first > last`



(f) `first > last` so no swap occurs within the loop.
swap `values[saveF]` and `values[last]`

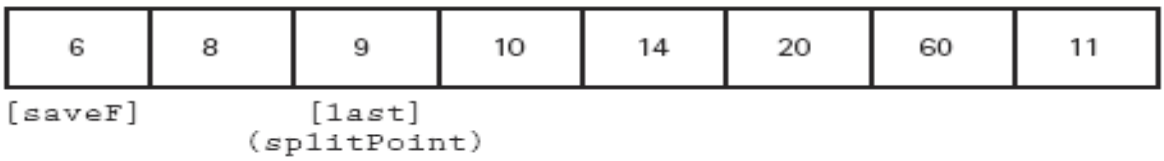


Figure 2.12.7 The split operation

When does the process stop? When first and last meet each other, no further swaps are necessary. Where they meet determines the splitPoint. This is the location where splitVal belongs, so we swap values[saveF], which contains splitVal, with the

element at values[splitPoint] (Figure 2.12.7f). The index splitPoint is returned from the method, to be used by quickSort to set up the next pair of recursive calls.

```
int split(int first, int last){
        int splitVal = values[first];
        int saveF = first;
        boolean onCorrectSide;
        first++;
        do{
                onCorrectSide = true;
                while (onCorrectSide) // Move first toward last
                        if (values[first] > splitVal)
                                onCorrectSide = false;
                        else{
                                first++;
                                onCorrectSide = (first <= last);
                        }
                onCorrectSide = (first <= last);
                while (onCorrectSide) // Move last toward first
                        if (values[last] <= splitVal)
                                onCorrectSide = false;
                        else{
                                last--;
                                onCorrectSide = (first <= last);
                        }
                if (first < last){
                        swap(first, last);
                        first++;
                        last--;
```

```
        } while (first <= last);
        swap(saveF, last);
        return last;
    }
```

What happens if our splitting value is the largest or the smallest value in the segment? The algorithm still works correctly, but because the split is lopsided, it is not so quick. Occurrence of this situation depends on how we choose our splitting value and on the original order of the data in the array. If we use values[first] as the splitting value and the array is already sorted, then every split is lopsided. One side contains one element, while the other side contains all but one of the elements. Thus, our quickSort is not a quick sort. Our splitting algorithm works best for an array in random order.

It is not unusual, however, to want to sort an array that is already in nearly sorted order. If this is the case, a better splitting value would be the middle value, values[(first + last) / 2] This value could be swapped with values[first] at the beginning of the method.

Analyzing quickSort

The analysis of quickSort is very similar to that of mergeSort. On the first call, every element in the array is compared to the dividing value (the "split value"), so the work done is O(N). The array is divided into two parts (not necessarily halves), which are then examined.

Each of these pieces is then divided in two, and so on. If each piece is split approximately in half, there are $O(\log_2 N)$ levels of splits. At each level, we make O(N) comparisons. So Quick Sort is also an $O(N\log_2 N)$ algorithm, which is quicker than the $O(N^2)$ sorts we discussed in the last lesson. But Quick Sort isn't always quicker. Note that there are $\log_2 N$ levels of splits if each split divides the segment of the array approximately in half. As we've seen, the array division of Quick Sort is sensitive to the order of the data, that is, to the choice of the splitting value.

What happens if the array is already sorted when our version of quickSort is called? The splits are very lopsided and the subsequent recursive calls to quickSort break into a segment of one element and a segment containing all the rest of the array. This situation produces a sort that is not at all quick. In fact, there are N - 1 levels; in this case Quick Sort is $O(N^2)$.

Such a situation is very unlikely to occur by chance. By way of analogy, consider the odds of shuffling a deck of cards and coming up with a sorted deck. On the other hand, in some applications you may know that the original array is likely to be sorted or nearly sorted. In such cases you would want to use either a different splitting algorithm or a different sort—maybe even shortBubble! Quick Sort does not require an extra array, as Merge Sort does. Since Quick Sort uses a recursive approach., there can be many levels of recursion "saved" on the system stack at any time. On average, the algorithm requires $O(\log_2 N)$ extra space to hold this information and in the worst case requires O(N) extra space, the same as Merge Sort.

2.12.6 Summary Basic sorting algorithms like bubble, insertion and selection sort are not efficient. Advanced sorting algorithms based on divide and conquer technique are far superior. They are based on recursively dividing the list and rearranging the element in sublists. In some case, when list is already sorted, shortBubble sort algorithm gives best performance. Merge sort is based on dividing array into two subarrays and then merging the sorted subarrays. Quicksort is based on selecting a pivot element and dividing the array by shifting elements smaller then its pivot element to the left of the pivot element and larger elements to the right and then applying the same login to the left and right half of the subarrays so obtained. Heap sort algorithm is based on creating a binary tree and then successively deleting the top element.

2.12.7 Questions

1. What do you mean by divide and conquer technique?
2. Explain the Merge sort algorithm.
3. Write the recursive Quick sort algorithm.
4. What is a pivot element? How is it chosen? What is the effect of choice of the pivot element? Explain giving example.
5. Define Heap. Explain the heap sort algorithm.
6. Write reheadDown algorithm.

2.12.8 Suggested readings

1. A. Tanenbaum, Y. Lanhgsam and A. J. Augenstein, "Data Structures Using C++", PHI.
2. M. A. Weiss, "Data Structures and Algorithm Analysis in C++", Pearson Education.

3. R. Sedgewick, “Algorithms in C++”, Pearson Education.

4. S. Lipschutz, “Data Structures”, Tata McGraw Hill.

5. Donald E. Knuth, “The Art Of Computer Programming”, Vol 1-4. 3rd ed., Addison Wesley

Type Setting :

Department of Distance Education,Punjabi University, Patiala