1. INTRODUCTION TO DATA STRUCTURE

2. STACK

3. QUEUES

4. LINKED LIST

5. TREES

6. GRAPHS

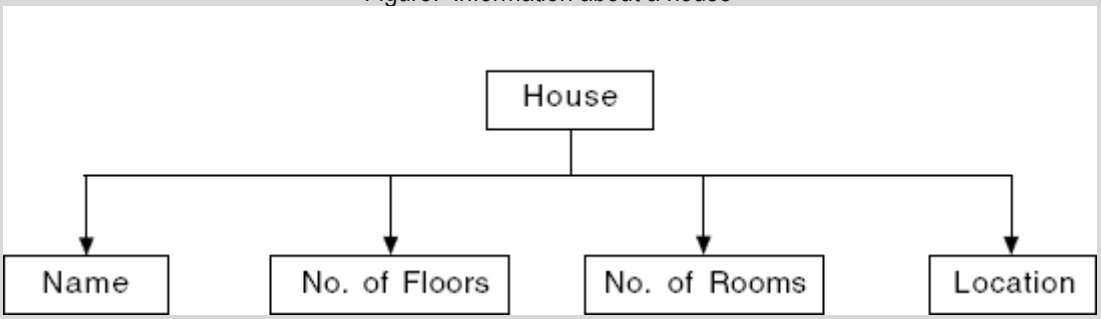
7. SORTING AND SEARCHING

CHAPTER 1
INTRODUCTION TO DATA STRUCTURE

DATA STRUCTURE AND TYPES

Data structures are a method of representing of logical relationships between individual data elements related to the solution of a given problem. Data structures are the most convenient way to handle data of different types including abstract data type for a known problem.
The components of data can be organized and records can be maintained. Further, the record formation leads to the development of abstract data type and database systems.

Figure. Information about a house



In data structures, we also have to decide on the storage, retrieval and operation that should be carried out between logically related items. For example, the data must be stored in memory in computer-understandable format, i.e. 0 and 1 and the data stored must be retrieved in human-understandable format, i.e. ASCII. In order to transform data various operations have to be performed.

TYPES OF DATA STRUCTURES

A data structure is a structured set of variables associated with one another in different ways, cooperatively defining components in the system and capable of being operated upon in the program. As stated earlier, the following operations are done on data structures:

1. Data organisation or clubbing
2. Accessing technique
3. Manipulating selections for information.

Data structures are the basis of programming tools and the choice of data structures should provide the following:

1. The data structures should satisfactorily represent the relationship between data elements.
2. The data structures should be easy so that the programmer can easily process the data.

Data structures have been classified in several ways. Different authors classify it differently. Fig.(a) shows different types of data structures. Besides these data structures some other data structures such as lattice, Petri nets, neural nets, semantic nets, search graphs, etc., can also be used. The reader can see Figs. (a) and (b) for all data structures.

Linear

In linear data structures, values are arranged in linear fashion. Arrays, linked lists, stacks and queues are examples of linear data structures in which values are stored in a sequence.

Non-Linear

This type is opposite to linear. The data values in this structure are not arranged in order. Tree, graph, table and sets are examples of non-linear data structures.

Homogenous

In this type of data structures, values of the same types of data are stored, as in an array.

Non-homogenous

In this type of data structures, data values of different types are grouped, as in structures and classes.

Dynamic

In dynamic data structures such as references and pointers, size and memory locations can be changed during program execution.

Static

Static keyword in C is used to initialize the variable to 0 (NULL). The value of a static variable remains in the memory throughout the program. Value of static variable persists. In C++ member functions are also declared as static and such functions are called as static functions and can be invoked directly.

Figure (a). Types of data structures

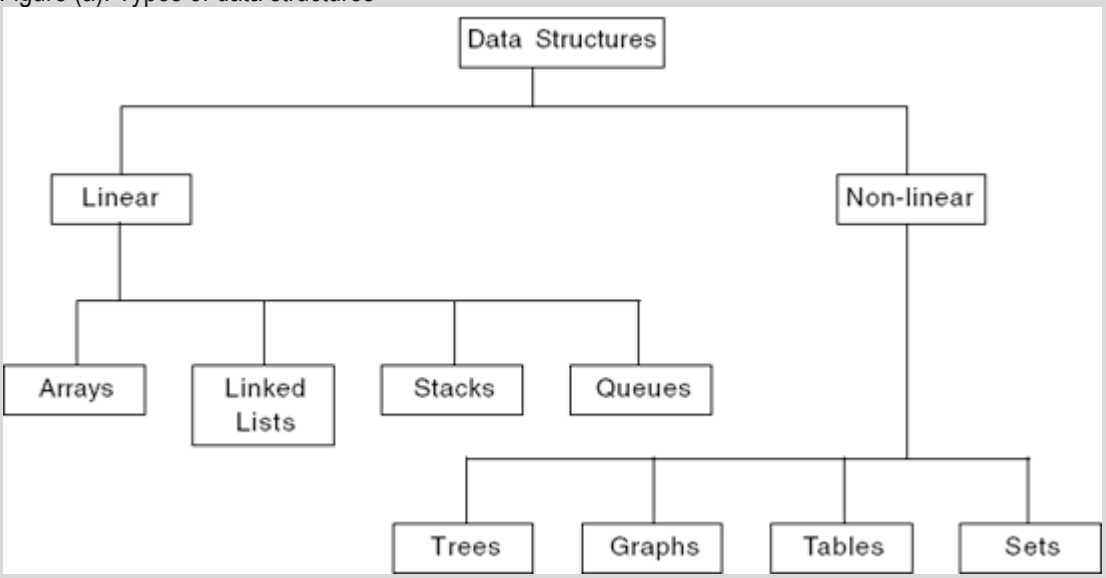
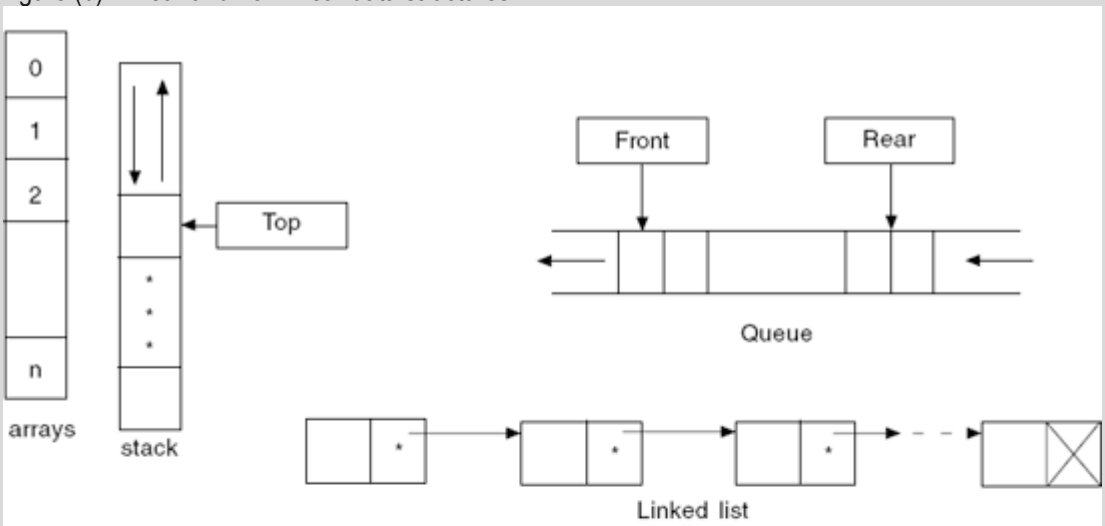
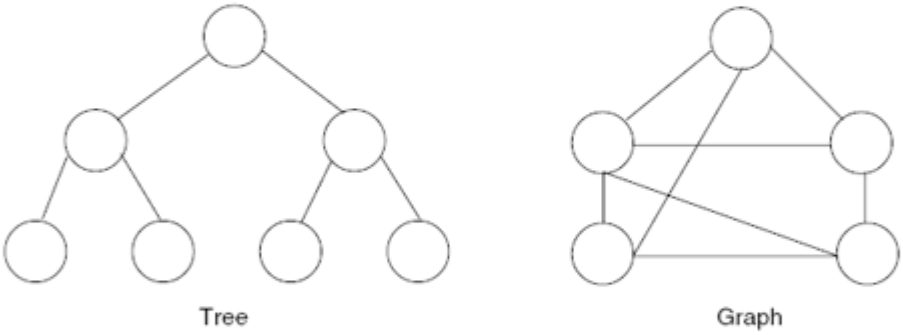
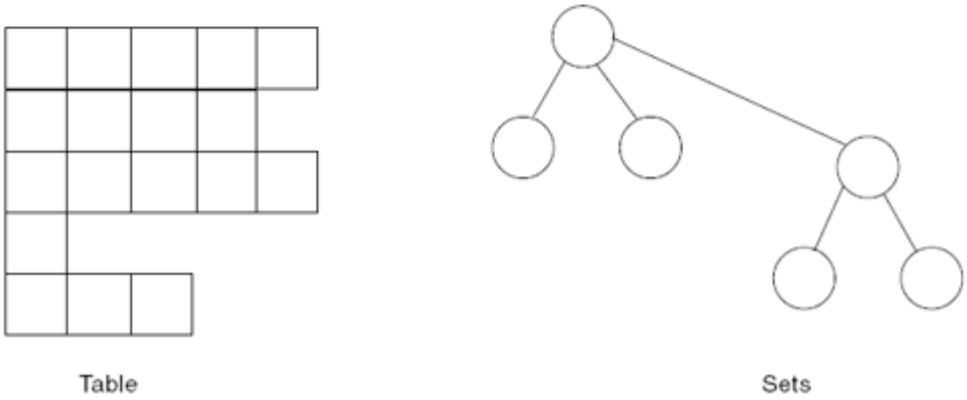


Figure (b). Linear and non-linear data structures



1. Linear data structures





2. Non-linear data structures

Primitive Data Structures

The integers, reals, logical data, character data, pointer and reference are primitive data structures. Data structures that normally are directly operated upon by machine-level instructions are known as primitive data structures.

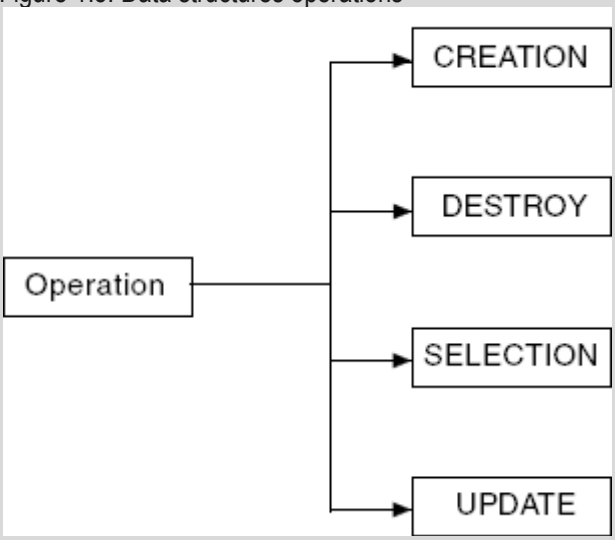
Non-primitive Data Structures

These are more complex data structures. These data structures are derived from the primitive data structures. They stress on formation of sets of homogeneous and heterogeneous data elements.

The different operations that are to be carried out on data are nothing but designing of data structures. The various operations that can be performed on data structures are shown in Fig. 1.5.

1. CREATE
2. DESTROY
3. SELECT
4. UPDATE

Figure 1.5. Data structures operations



An operation typically used in combination with data structures and that creates a data structure is known as creation. This operation reserves memory for the program elements. It can be carried out at compile time and run-time.

For example,

```
int x;
```

Here, variable x is declared and memory is allocated to it.

Another operation giving the balancing effect of a creation operation is destroying operation, which destroys the data structures. The destroy operation is not an essential operation. When the program execution ends, the data structure is automatically destroyed and the memory allocated is eventually de-allocated. C++ allows the destructor member function to destroy the object. In C free () function is used to free the memory. Languages like Java have a built-in mechanism, called garbage collection, to free the memory.

The most commonly used operation linked with data structures is selection, which is used by programmers to access the data within data structures. The selection relationship depends upon yes/no. This operation updates or alters data. The other three operations associated with selections are:

1. Sorting
2. Searching
3. Merging

Searching operations are used to seek a particular element in the data structures. Sorting is used to arrange all the elements of data structures in the given order: either ascending or descending. There is

a separate chapter on sorting and searching in this book. Merging is an operation that joins two sorted lists.
An iteration relationship is nothing but a repetitive execution of statements. For example, if we want to perform any calculation several times then iteration, in which a block of statements is repetitively executed, is useful.
One more operation used in combination with data structures is update operation. This operation changes data of data structures. An assignment operation is a good example of update operation.
For example,

```
int x=2;
```

Here, 2 is assigned to x.

```
x=4;
```

Again, 4 is reassigned to x. The value of x now is 4 because 2 is automatically replaced by 4, i.e. updated.

ABSTRACT DATA TYPES

In programming, a situation occurs when built-in data types are not enough to handle the complex data structures. It is the programmer's responsibility to create this special kind of data type. The programmer needs to define everything related to the data type such as how the data values are stored, the possible operations that can be carried out with the custom data type and that it must behave like a built-in type and not create any confusion while coding the program. Such custom data types are called Abstract data type. In C struct and in C++ struct/class keywords are used to create abstract data type.

For example, if the programmer wants to define date data type which is not available in C/ C++, s/he can create it using struct or class. Only a declaration is not enough; it is also necessary to check whether the date is valid or invalid. This can be achieved by checking using different conditions. The following program explains the creation of date data type:

Write a program to create abstract data type date.

```
# include <stdio.h>
# include <conio.h>

struct date
{
 int dd;
 int mm;
 int yy;
};

main ()
{
struct date d; // date is abstract data type
clrscr();
printf ("Enter date (dd mm yy) :");
scanf ("%d %d %d",&d.dd,&d.mm, &d.yy);
printf ("Date %d-%d-%d",d.dd,d.mm,d.yy);
}
```

OUTPUT

Enter date (dd/mm/yy): 08 12 2003
Date 08-12-2003

RECURSIVE FUNCTION

RULES

1. In recursion, it is essential for a function to call itself, otherwise recursion will not take place.
2. Only user defined function can be involved in the recursion. Library function cannot be involved in recursion because their source code cannot be viewed.
3. A recursive function can be invoked by itself or by other function. It saves return address in order with the intention that to return at proper location when return to a calling statement is made. The last-in-first-out nature of recursion indicates that stack data structure can be used to implement it.
4. Recursion is turning out to be increasingly important in non-numeric applications and symbolic manipulations.
5. To stop the recursive function it is necessary to base the recursion on test condition and proper terminating statement such as exit () or return must be written using if () statement.
6. Any function can be called recursively. An example is illustrated in amar sir's class.
7. When a recursive function is executed, the recursion calls are not implemented instantly. All the recursive calls are pushed onto the stack until the terminating condition is not detected. As soon as the terminating condition is detected, the recursive calls stored in the stack are popped and executed. The last call is executed first then second, then third and so on.

8. During recursion, at each recursive call new memory is allocated to all the local variables of the recursive functions with the same name.
9. At each call the new memory is allocated to all the local variables, their previous values are pushed onto the stack and with its call. All these values can be available to corresponding function call when it is popped from the stack.

RECURSION VS. ITERATIONS

Recursion Vs. Iteration	
Recursion	Iteration
Recursion is the term given to the mechanism of defining a set or procedure in terms of itself.	The block of statement executed repeatedly using loops.
A conditional statement is required in the body of the function for stopping the function execution.	The iteration control statement itself contains statement for stopping the iteration. At every execution, the condition is checked.
At some places, use of recursion generates extra overhead. Hence, better to skip when easy solution is available with iteration.	All problems can be solved with iteration.
Recursion is expensive in terms of speed and memory.	Iteration does not create any overhead. All the programming languages support iteration.

Iterative Processes

Initialization

The variables involved in the iteration process are initialized. These variables are used to decide when to end the loop.

Decision

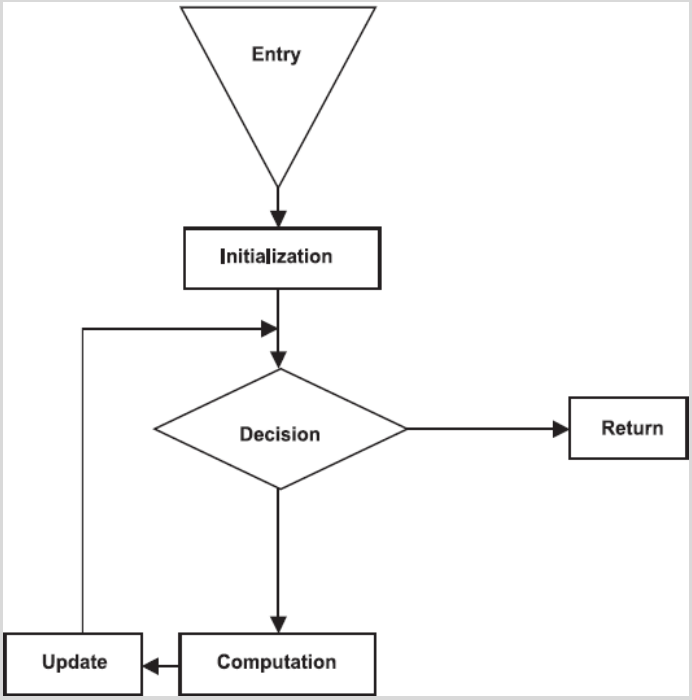
The decision variable is used to decide whether to continue or discontinue the loop. When the condition is satisfied, control goes to return, or else it goes to computation block.

Computation

The required processing or computation is carried out in this block.

Update

The decision argument is changed and shifted to next iteration.



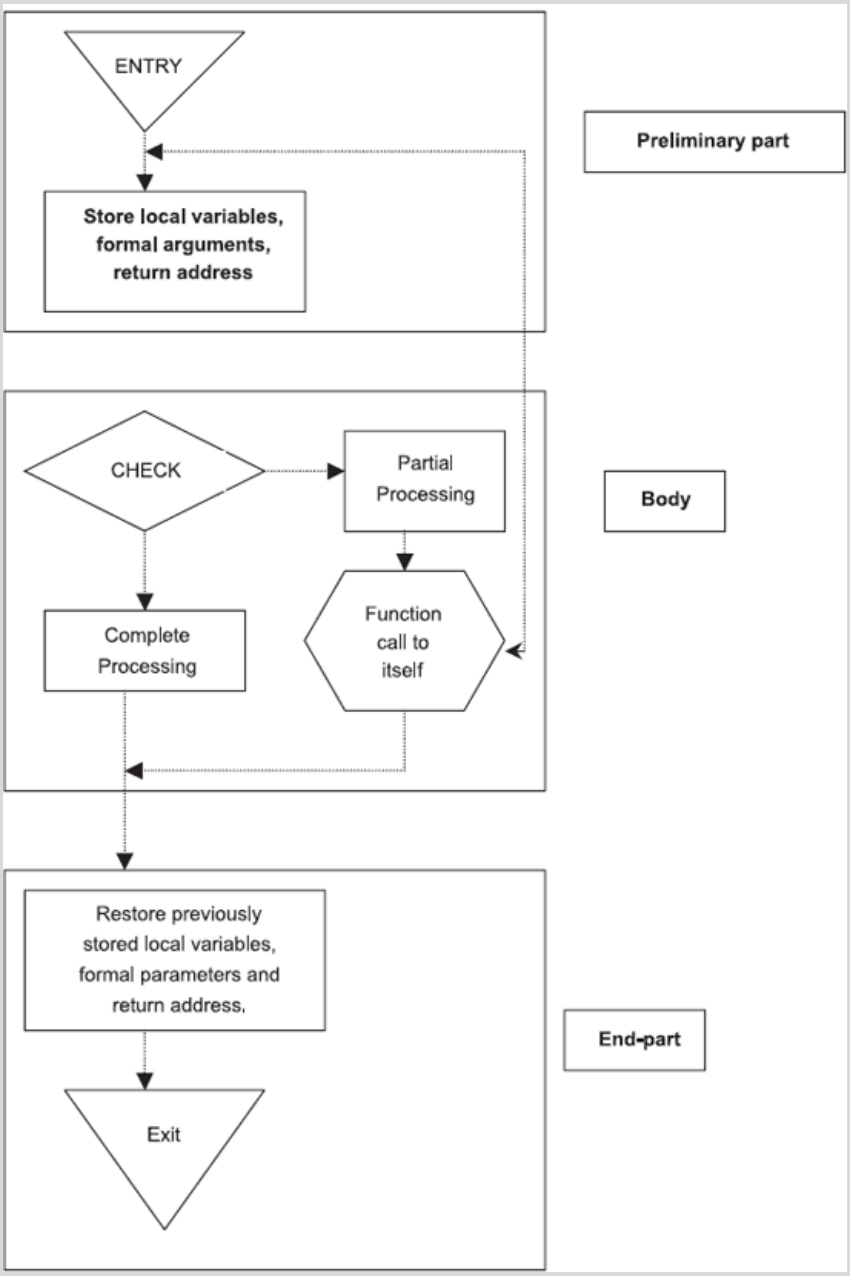
Recursive Function

Preliminary part

The use of this block is to store the local variables, formal arguments and return address. The end-part will restore these data. Only recently saved arguments, local variables and return address are restored. The variables last saved are restored first.

Body

If the test condition is satisfied, it performs the complete processing and control passes to end-block. If not, partial processing is performed and a recursive call is made. The body also contains call to itself and one or more calls can be made. Every time a recursive call is made, the preliminary part of the function saves all the data. The body also contains two processing boxes i.e. partial processing and complete processing. In some programs, the result can be calculated after complete processing. For this, the recursive call may not be required. For example, we want to calculate factorial of one. The factorial of one is one. For this, it is needless to call function recursively. It can be solved by transferring control to complete processing box.



In other case, if five is given for factorial calculation, the factorial of five can be calculated in one step. Hence, the function will be called recursively. Every time one step is solved, i.e. 5*4*3 and so on. Hence, it is called partial processing.

THE TOWERS OF HANOI

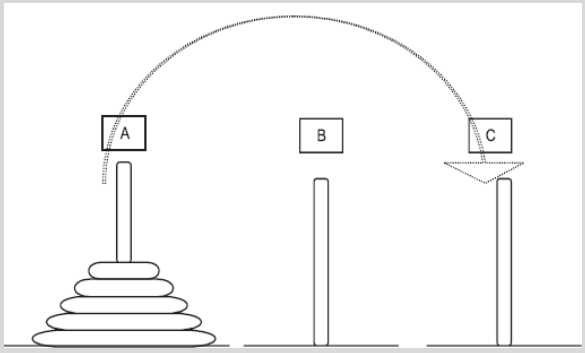
The Tower of Hanoi has historical roots in the ceremony of the ancient tower of Brahma. There are n disks of decreasing sizes mounted on one needle as shown in the Fig. (a). Two more needles are also required to stack the entire disk in the decreasing order. The use of third needle is for impermanent storage. While mounding the disk, following rules should be followed.

1. At a time only one disk may be moved.

2. The disk may be moved from any needle to another needle.
3. No larger disk may be placed on the smaller one.

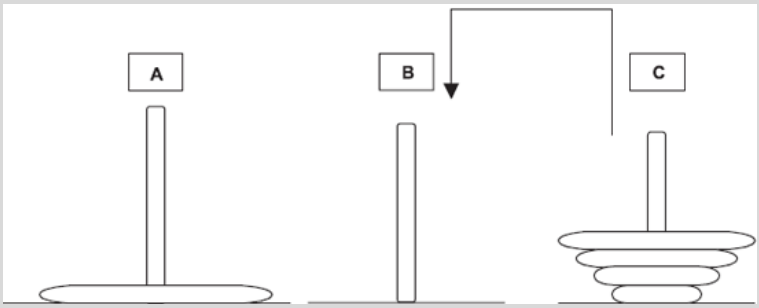
Our aim is to move the disks from A to C using the needle B as an intermediate by obeying the above three conditions. Only top-most disks can be moved to another needle. The following figures and explanation clear the process of Tower of Hanoi stepwise.

Figure (a). Recursive Towers of Hanoi



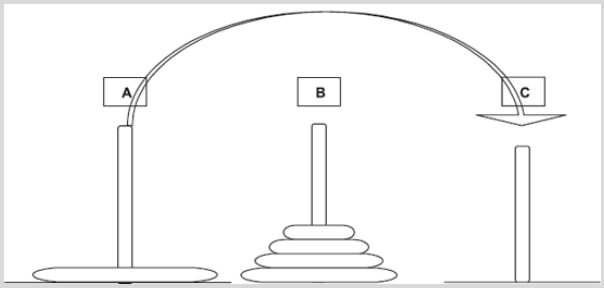
In Fig. (a), the three needles are displayed in their initial states. The needle A contains five disks and there are no disk on needle B and C. The arrow indicates the next operation to be performed, i.e. move first four disk from A to C.

Figure (b). Recursive Towers of Hanoi



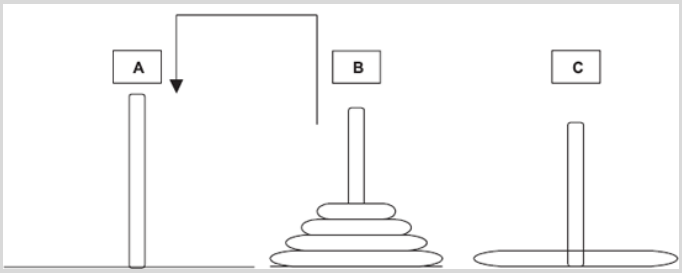
In Fig. (b) the needle, A has only one disk. The needle C contains four disks. The arrow indicates the next operation, i.e. move four disks from C to B.

Figure (c). Recursive Towers of Hanoi



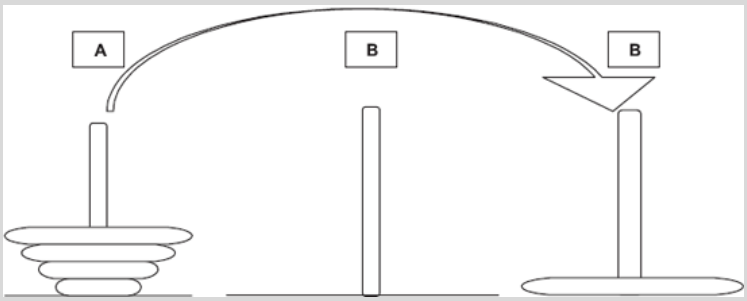
In Fig. (c), the needle B has four disks. The needle A has only one disk, which will be moved to needle C.

Figure (d). Recursive Towers of Hanoi



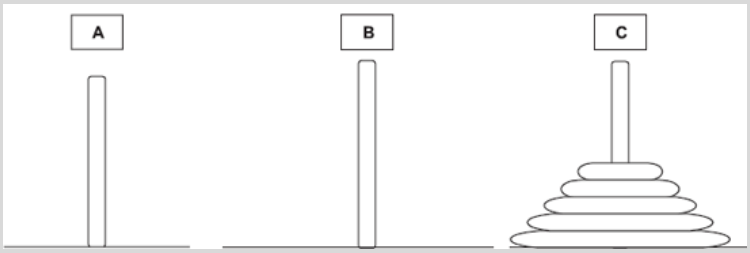
In Fig. (d), the needle A has no disk. The needle B contains four disks and C contains one disk. The four disks move from B to A.

Figure (e). Recursive Towers of Hanoi



In Fig. (e), the four disks from needle A are moved to needle C. Finally, the final position of needles A, B and C will be as shown in Fig. (f). Here, solution of Tower of Hanoi is complete.

Figure (f). Recursive Towers of Hanoi



The above process can be summarized as:

1. Move first four disks from needle A to C
2. Move four disks from C to B
3. Move the disk one from A to C
4. Move four disks from B to A
5. Move four disks from A to C.

Program to illustrate Towers of Hanoi.

```
# include <conio.h>
# include <stdio.h>

void hanoi (int,char,char,char);

main ()
{
  int num;
  clrscr();
  printf ("\n Enter a Number: ");
```


```
  scanf ("%d",&num);
  clrscr();
  hanoi(num,'A','C','B');
}

void hanoi (int num, char f_peg,char t_peg, char a_peg)
{


  if (num==1)
  {
      printf ("\nMove disk 1 from Needle %c to Needle %c",f_peg,t_peg);
      return;
  }
  hanoi(num-1,f_peg,a_peg,t_peg);
  printf ("\nMove disk %d from Needle %c to Needle %c",num,f_peg,t_peg);
  hanoi(num-1,a_peg,t_peg,f_peg);
}
```

OUTPUT

```
Move disk 1 from Needle A to Needle C
Move disk 2 from Needle A to Needle B
Move disk 1 from Needle C to Needle B
Move disk 3 from Needle A to Needle C
Move disk 1 from Needle B to Needle A
Move disk 2 from Needle B to Needle C
Move disk 1 from Needle A to Needle C
```


ADVANTAGES & DISADVANTAGES OF RECURSION

Advantages of recursion,

1. Sometimes, in programming a problem can be solved without recursion, but at some situations in programming it is must to use recursion. For example, a program to display the list of all files of the system cannot be solved without recursion.
2. The recursion is very flexible in data structure like stacks, queues, linked list and quick sort.
3. Using recursion, the length of the program can be reduced.

Disadvantages of recursion,

1. It requires extra storage space. The recursive calls and automatic variables are stored on the stack. For every recursive call separate memory is allocated to automatic variables with the same name.
2. If the programmer forgot to specify the exit condition in the recursive function, the program will execute out of memory. In such a situation user has to press ctrl+ break to pause or stop the function.
3. The recursion function is not efficient in execution speed and time.
4. If possible, try to solve problem with iteration instead of recursion.

CHAPTER 2:
STACKS

Stack is an important tool in programming languages. Stack is one of the most essential linear data structures. Implementation of most of the system programs is based on stack data structure.

We can insert or delete an element from a list, which takes place from one end. The insertion of element onto the stack is called as "push" and deletion operation is called "pop", i.e. when an item is added to the stack the operation is called "push" and when it is removed the operation is called "pop". Due to the push operation from one end, elements are added to the stack, the stack is also known as pushdown list.

The most and least reachable elements in the stack are respectively known as the "top" and "bottom" of the stack. A stack is an arranged collection of elements into which new elements can be inserted or from which existing new elements can be deleted at one end. Stack is a set of elements in a last-in-first-out technique. As per Fig. 2.1 the last item pushed onto the stack is always the first to be removed from the stack. The end of the stack from where the insertion or deletion operation is carried out is called top. In Fig. 2.1(a) a stack of numbers is shown.

Figure 2.1(a). Stack

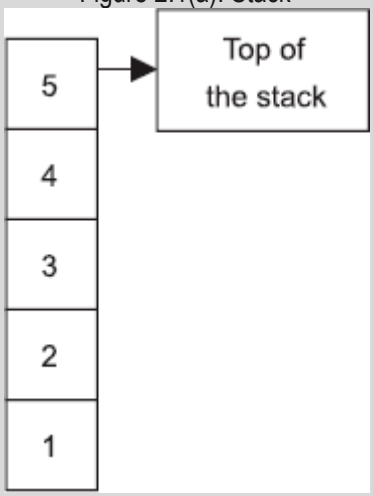
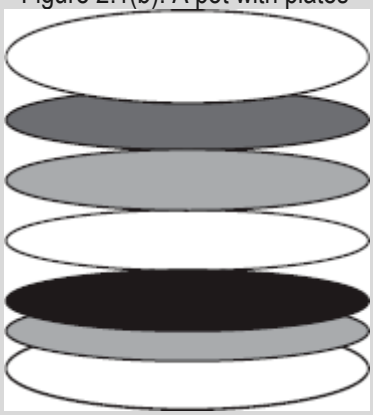


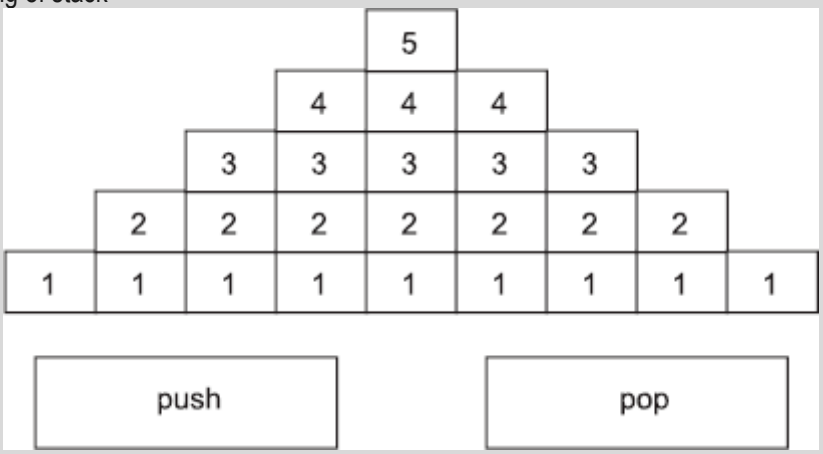
Figure 2.1(b). A pot with plates



As shown in Fig. 2.1(a) the stack is based on the rule last-in-first-out. The element appended lastly is deleted first. If we want to delete element 3 it is necessary to delete the top elements 5 and 4 first.

In Fig. 2.1(b), you can see a pot containing plates kept one above the other. Plates can be inserted or removed from the top. The top plate can be removed first in case pop operation is carried out, otherwise plates are to be added on the top. In other words, the removal operation has to be carried out from the top. Thus, placing or removing takes place from top, i.e. from the same end.

Figure 2.2. Working of stack



As shown in the first half portion of Fig. 2.2, first integer 1 is inserted and then 2,3,4 and 5 are pushed on to the stack. In the second half portion of the same figure, the elements are deleted (pop) one by one from the top. The elements are deleted in the order from 5 to 1. The insertion and deletion operation is carried out at one end (top). Hence, the recently inserted element is deleted first. If we want to delete a particular element of the stack, it is necessary to delete all the elements present above that element. It is not possible to delete element 2 without deleting elements 5,4 and 3. The stack expands or shrinks with the passage of time. In the above example, the stack initially expands until element 5 is inserted and then shrinks after removal of elements. There is no higher limit on the number of elements to be inserted in the stack. The total capacity of stack depends on memory of the computer.

In practical life, we come across many examples based on the principle of stack.

Figure 2.3. Stack of books

Fig. 2.3 illustrates the stack of books that we keep in the order. Whenever we want to remove a book the removal operation is made from the top or new books can be added at the top.

STACK-RELATED TERMS

Stack

Stack is a memory portion, which is used for storing the elements. The elements are stored based on the principle of last-in-first-out. In the stack the elements are kept one above the other and its size is based on the memory.

Top

The top of the pointer points to the top element in the stack. The top of the stack indicates its door from where elements are entered or deleted. The stack top is used to verify stack's current position, i.e. underflow, overflow, etc. The top has value 0 when the stack is empty. Some programmers assign -1 to the top as initial value. This is because when the element is added, the top is incremented and it would become zero. The stack is generally implemented with the help of an array. In an array, counting of elements begins from 0 onwards. Hence, on the similar grounds stack top also begins from 0 and it is convenient to assign -1 to top as initial value.

Stack Underflow

When there is no element in the stack or stack holds elements less than its capacity, the status of stack is known as stack underflow. In this situation, the top is present at the bottom of the stack. When an element is pushed, it will be the first element of the stack and top will be moved one step up.

Stack Overflow

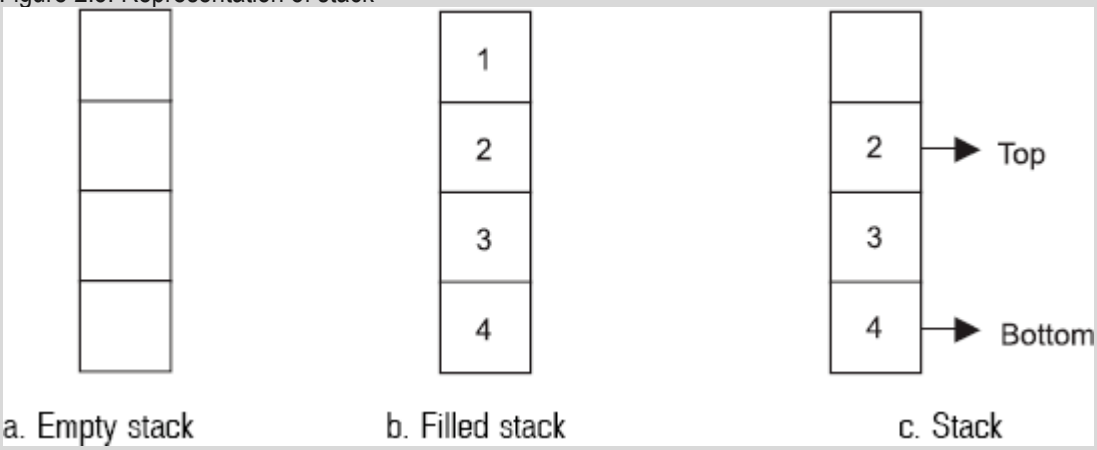
When the stack contains equal number of elements as per its capacity and no more elements can be added, the status of stack is known as stack overflow. In such a position, the top rests at the highest position.

2.2.1 Representation of Stack

The stack is represented in various ways. The stack can be shown as completely empty or fully filled or filled with few elements. When stack has no element, it is called empty stack which is shown in Fig. 2.5(a). The stack with completely filled elements is shown in the Fig. 2.5 (b) and no further elements can be inserted.

A top pointer maintains track of the top elements as shown in Fig. 2.5(c). For empty stack, the top value is zero. When stack has one element the value of top is one. Whenever an element is inserted (push operation) in the stack the value of pointer top is incremented by one. In the same manner, the value of pointer is reduced when an element is deleted from the stack (pop operation). The push operation can be applied to any stack, but before applying pop operation on the stack, it is necessary to make sure that the stack is not empty. If a pop operation is performed on empty stack, it results in underflow.

Figure 2.5. Representation of stack



Stack is very helpful in every ordered and chronological processing of functions. The most useful application of stack is in recursion (explained in previous chapter). It saves memory space. The mechanism of stack last-in-first-out (LIFO) is commonly useful in applications such as manufacturing and accounting calculations. It is a well-known accounting concept.

OPERATION ON STACK

Stack Methods

Create Stack: The fields of the stack comprise a variable to hold its maximum size (the size of the array), the array itself, and a variable top, which stores the index of the item on the top of the stack. (Note that we need to specify a stack size only because the stack is implemented using an array. If it had been implemented using a linked list, for example, the size specification would be unnecessary.)

The push() method increments top so it points to the space just above the previous top, and stores a data item there. Notice that top is incremented before the item is inserted.

The pop() method returns the value at top and then decrements top. This effectively removes the item from the stack; it's inaccessible, although the value remains in the array (until another item is pushed into the cell).

The peek() method simply returns the value at top, without changing the stack.

The isEmpty() and isFull() methods return true if the stack is empty or full, respectively. The top variable is at –1 if the stack is empty and maxSize-1 if the stack is full.

Figure 2.6(a). Push operation with stack

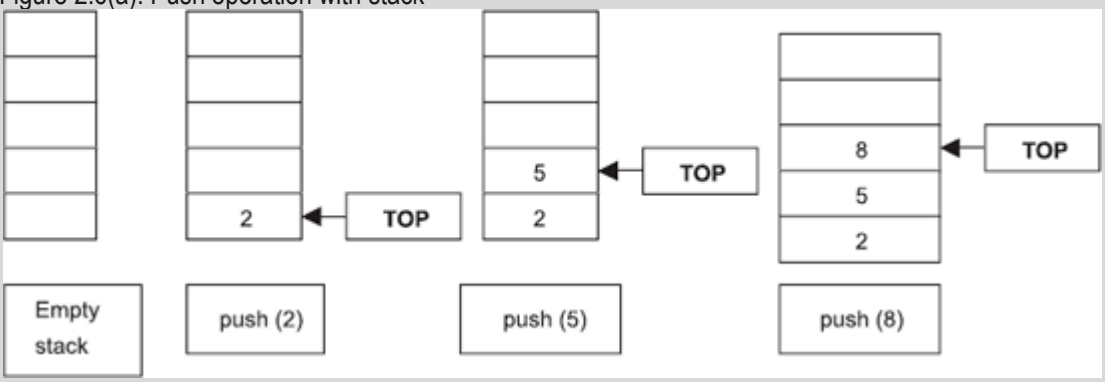
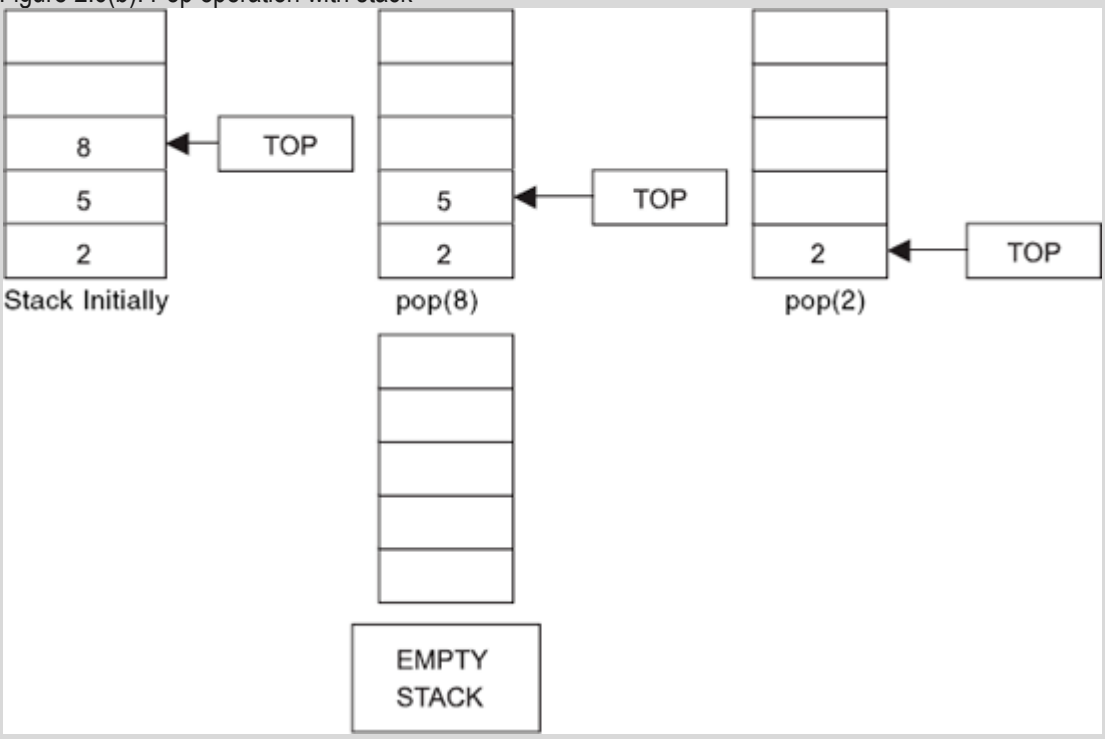


Figure 2.6(b). Pop operation with stack



STACK IMPLEMENTATION

The stack implementation can be done in the following two ways:

1 Static Implementation

Static implementation can be achieved using arrays. Though, it is a very simple method, it has few limitations. Once a size of an array is declared, its size cannot be modified during program execution. It is also inefficient for utilization of memory. While declaration of an array, memory is allocated which is equal to array size. The vacant space of stack (array) also occupies memory space. In both cases, if we store less argument than declared, memory is wasted and if we want to store more elements than declared, array cannot be expanded. It is suitable only when we exactly know the number of elements to be stored.

2 Dynamic Implementation

Pointers can also be used for implementation of stack. The linked list is an example of this implementation. The limitations noticed in static implementation can be removed using dynamic implementation. The dynamic implementation is achieved using pointers. Using pointer implementation at run time there is no restriction on the number of elements. The stack may be expandable. The memory is efficiently utilized with pointers. Memory is allocated only after element is pushed to the stack. Both the above implementations are illustrated with suitable examples in the next section.

REPRESENTATION OF ARITHMETIC EXPRESSIONS

An arithmetic expression contains operators and operands. Variables and constants are operands. The operators are of various kinds such as arithmetic binary, unary for example, + (addition), - (subtraction), *(multiplication), / (division) and % (modular division).

The precedence of operator also plays an important role in expression solving. The precedence of operators is given in Table 2.2.

We have already studied the different stack operations. Now, we will study how stack can be useful in problem solving. Consider a mathematical expression.

5 - *((A* * ((*B* + *K*)/ (*U*-4))+*K*) / 8 (8-2.3))

The common mistake, which can be made frequently by the programmer, is unbalance of parenthesis. For correct representation of mathematical expression, the following precautions must be taken:

Table 2.2. Precedence of operators

Operators	Precedence	Associativity
+ (unary), - (unary), NOT	6	—
^ (Exponentiation)	6	Right to left
* (multiplication), / (division)	5	Left to right
+ (addition), - (subtraction)	4	Left to right
<,<=,+,<,>,>=	3	Left to right

1. There must be equal number of left and right parenthesis.
2. Each left parenthesis must be balanced by right parenthesis.

INFIX, PREFIX AND POSTFIX NOTATIONS

. Stack has various real life applications. Stack is very useful to keep sequence of processing. In solving arithmetic expressions, stacks are used. Stacks are used to convert bases of numbers. In a large program, various functions are invoked by the main () function; all these functions are stacked in the memory. Most of the calculators work on stack mechanism.

Arithmetic expressions can be defined in three kinds of notation: infix, prefix and postfix. The prefixes in, pre and post indicate the relative location of the operator with two operands.

Infix Notation

Expressions are generally expressed in infix notation with binary operator between the operands. The binary operator means the operator requires two operands such as +, *, / and %. In this type, the operator precedes the operands. Following are examples of infix notation:

1. *x+y*
2. *x+y*z*
3. *(x+y)*z*
4. *(x+y)*(p+q)*

Every single letter (A-Z) and an unsigned integer is a legal infix expression. If X and Y are two valid expressions then (X+Y), then (X-Y) and (X/Y) are legal infix expressions.

Prefix Notation

The prefix notation is also called polish notation. The polish mathematician Lukasiewicx invented it. In this type also the operator precedes the operands. Following are the examples of prefix notation:

1. *+xy*
2. *+x*yz*
3. **+xyz*
4. **+xy+pq*

The postfix notation is also called as reverse polish notation. The operator trails the operand. Following are the examples of postfix notation:

1. *xy+*
2. *xyz*+*
3. *xy+z**
4. *xy+pq*+*

EVALUATION OF POSTFIX AND PREFIX EXPRESSION

An algorithm to evaluate postfix expression.

1. Scan the input from left to right.
2. If the number is an operand, push it on the stack.
3. If the number is an operator, pop the two operands from the stack, perform the operation and push the result back on the stack.

4. Continue from step 2 onwards till the end of the input.

An algorithm to evaluate prefix expression.

(Reverrse prreffiix and evalluatte it.)

1. reverse given prefix expression;
2. scan the reversed prefix expression;
3. for each symbol in reversed prefix
4. if operand then push its value onto stack.
5. If the number is an operator, pop the two operands from the stack, perform the operation and push the result back on the stack
6. Continue from step 2 onwards till the end of the input

CONVERSION OF EXPRESSION FROM INFIX TO PREFIX AND POSTFIX

Algorithm to convert Infix Expression to Postfix Expression

Step 1. Push Left Parenthesis "(" onto stack and add right parenthesis ")" to end of the A
Step 2. Scan A from left to right and repeat steps 3 to 6 for each element of A until the stack is empty
Step 3. If an operand is encountered, add it to B
Step 4. If a left parenthesis is encountered push it onto the stack
Step 5. If an operator is encountered then
    a. Repeatedly pop from the STACK and add to B each operator (on the top of the stack) which has the same precedence as or higher precedence than operator
    b. Add operator to STACK
Step 6. If a right parenthesis is encountered, then
    a. Repeatedly pop from the STACK and add to B each operator (on the top of STACK) until a left parenthesis is encountered
    b. Remove the left parenthesis. (Do not add left parenthesis to B)
Step 7. Exit

Algorithm to Convert Infix to Prefix Form

Suppose A is an arithmetic expression written in infix form. The algorithm finds equivalent prefix expression B.
Step 1. Push ")" onto STACK, and add "(" to end of the A
Step 2. Scan A from right to left and repeat step 3 to 6 for each element of A until the STACK is empty
Step 3. If an operand is encountered add it to B
Step 4. If a right parenthesis is encountered push it onto STACK
Step 5. If an operator is encountered then:
    a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
    b. Add operator to STACK
Step 6. If left parenthesis is encontered then
    a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encounterd)
    b. Remove the left parenthesis
Step 7. Exit

APPLICATIONS OF STACKS

- Expression conversion
- Expression evaluation
- Reversing a string
- Parsing
- Well formed parentheses
- Decimal to Binary Conversion
- Recursive procedure call
- We will be studying only two applications:
- Evaluating a postfix expression
- Converting an expression from infix to postfix.

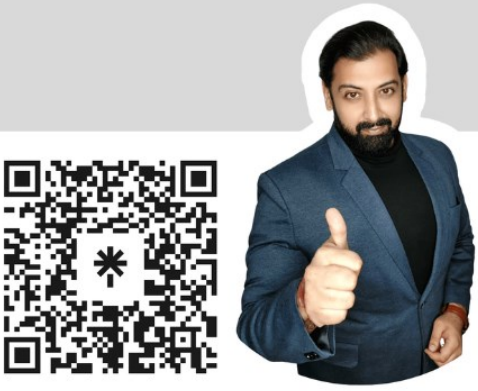
Few applications of stack are narrated together with examples.

Reverse String

We know the stack is based on last-in-first-out rule. It can be achieved simply by pushing each character of the string onto the stack. The same can be popped in reverse fashion. Thus, reverse string can be done. Consider the following program.

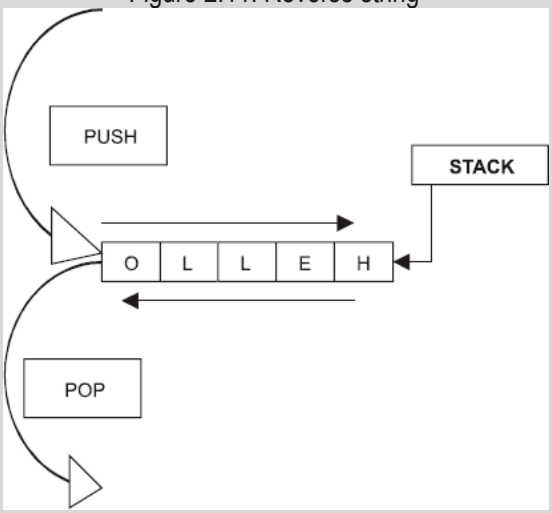
Write a program to reverse the string using stack.

```
# include <stdio.h>
# include <conio.h>

char text[40];
void main ()
{
 char ch;
 void push (char,int);
 char pop(int);
 int j=39,k;
 clrscr();
 puts ("\n Enter a string (* to end): ");

while (ch!='*' && j>=0)
{
 ch=getche();
 push (ch,j);
 j--;
}
k=j;
j=0;

printf ("\n Reverse string is: ");

while (k!=40)
{
 ch=pop(k);
 printf ("%c",ch);
 k++;
}
}
void push (char c, int j)
{
 if (c!='*')
 text[j]=c;
}
```

```
char pop (int j)
{
 char c;
 c=text[j];
 text[j]=text[j+1];
 return c;
}

OUTPUT
Enter a string (* to end):
HELLO*
Reverse string is: OLLEH
```

Figure 2.11. Reverse string



2.10.3 Conversion of Number System

Suppose, one wants to calculate binary number of a given decimal number, the given number is repeatedly divided by 2 until 0 is obtained. The binary number can be displayed in reverse order using stack rule last-in-first-out (LIFO). Consider the following program:

Example 2.13.

Write a program to convert a given decimal number to binary. Explain the role of stack mechanism.

Solution

```
# include <stdio.h>
# include <conio.h>

int num[7];

main ()
{
  int n,k,T=7;
  void show();
  void push(int,int);
  clrscr();
  printf ("\n Enter a number: ");
  scanf("%d",&n);
  printf ("\n The binary number is: ");
  while (n)
  {
```


```
   k=n%2;
   push(--T,k);
   n=n/2;
   }

show();
}

void push (int j, int b)
{
 num[j]=b;
}
void show ()
{
 int j;
 for (j=0;j<7;j++)
 printf (" %d ",num[j]);
}

OUTPUT
Enter a number: 9
The binary number is: 0 0 0 1 0 0 1
```

Explanation In this program the binary equivalent is obtained by repeatedly dividing by two. Here, the first binary digit obtained is pushed on the stack. This process is continued. The show () function displays the binary digits stored in the stack, i.e. an array.

2.10.4 Recursion

The recursion is the fundamental concept of the mathematical logic. Recursion is one of the important facilities provided in many languages. C also supports recursion. There are many problems, which can be solved recursively. The loop which performs repeated actions on a set of instructions can be classified as either iterative or recursive. The recursive loop is different from the iterative loop. In the recursion, the procedure can call itself directly or indirectly. In the directly called recursion the procedure calls itself repeatedly. On the other hand, if the procedure calls another procedure then it is an indirect recursion. In recursion, some statements, which are specified in the function, are executed repeatedly. Every time a new value is passed to the recursive function till the condition is satisfied. A simple programming example is given below.

Example 2.14.

Write a program and find the greatest common divisor of the given two numbers.

```
# include <stdio.h>
# include <conio.h>
int stack[40],top=-1;
main ()
{
 void gcd(int,int);
 int n1,n2;
 clrscr();
 printf("\nEnter number:- ");
 scanf("%d",&n1);
 printf("\nEnter number:- ");
 scanf("%d",&n2);
 gcd(n1,n2);
 printf("\nThe gcd of %d & %d is:- %d",n1,n2,stack[top]);
```


```
}

void gcd(int a,int b)
{
if(a!=b)
  {
  if(a>b)
  {
  top++;
  stack[top]=a-b;
  printf("\nTop value is:- %d",stack[top]);
  gcd(a-b,b);
  }
else
  {
  top++;
  stack[top]=b-a;
  printf("\nTop value is:- %d",stack[top]);
  gcd(a,b-a);
  }
 }
}

OUTPUT:
Enter number:- 5
Enter number:- 25
Top value is:- 20
Top value is:- 15
Top value is:- 10
Top value is:- 5
The gcd of 5 & 25 is:- 5
```


Example 2.15.

Write a program to convert decimal to binary by using the concept of recursion.

```
# include <stdio.h>
# include <conio.h>
int stack[40],top=-1;
main ()
{
  void binary(int);
  int no;
  clrscr();
  printf("\nEnter number:- ");
  scanf("%d",&no);
  binary(no);

  printf("\nThe binary of the given number is:-");

  while(top>=0)
  {
  printf(" %d ",stack[top]);
  top--;
  }
```

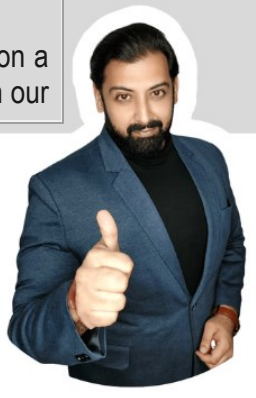

```
}

void binary(int b)
{
   if(b>0)
   { top++;
   stack[top]=b%2;
   binary(b/2);
 }
}
```

OUTPUT:
Enter number:- 255
The binary of the given number is:- 1 1 1 1 1 1 1 1

QUEUES

INTRODUCTION

A queue is one of the simplest data structures and can be implemented with different methods on a computer. Queue of tasks to be executed in a computer is analogous to the queue that we see in our



daily life at various places. The theory of a queue is common to all. This data structure is very useful in solving various real life problems.

A queue is a non-primitive, linear data structure, and a sub-class of list data structure. It is an ordered, homogenous collection of elements in which elements are appended at one end called rear end and elements are deleted at other end called front end. The meaning of front is face side and rear means back side. The first entry in a queue to which the service is offered is to the element that is on front. After servicing, it is removed from the queue. The information is manipulated in the same sequence as it was collected. Queue follows the rule first-in-first-out (FIFO). Fig. 3.1 illustrates a queue.

Figure 3.1. Queue

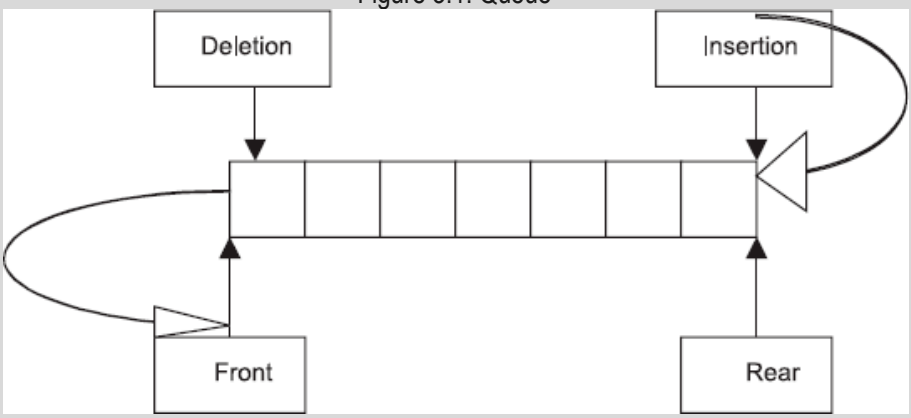


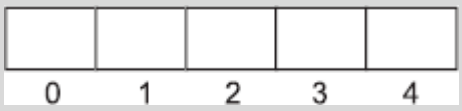
Figure 3.1 shows that insertion of elements is done at the rear end and deletion at the front end. In other words, the service is provided to the element, which is at front and this element is to be removed first. We call this entry front of the queue. The entry of element that is recently added is done from rear of the queue or it can be called as tail end of the queue.

Two operations are frequently carried on queue. They are insertion and deletion of elements. These two operations depend upon the status of the queue. The insertion operation is possible, only when the queue contains elements less than the capacity it can hold. The deletion operation is only possible when the queue contains at least one element.

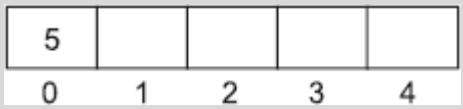
DISADVANTAGES OF SIMPLE QUEUES

In the previous sections we have studied implementation of queues using arrays and pointers. There are some disadvantages in simple queue implementation using arrays. In the following example, it is considered that the elements after deletion are not shifted to beginning of an array. Consider the following example:

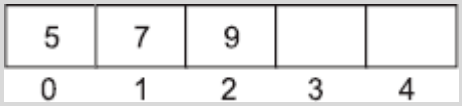
Queue [5] is a simple queue declared. The queue is initially empty. In the following figures insertion and deletion operations will be performed.



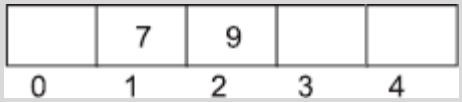
The above figure is an empty queue. Initially the queue is always empty. Here, rear = -1 and front = -1. The user can also assign 0 instead of -1. However, the -1 is suitable because we are implementing this problem with arrays and an array element counting begins always with zero. Thus, it is suitable for problem logic.



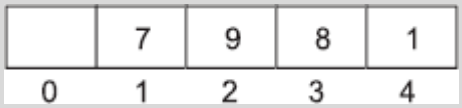
In the above figure, one element is inserted. So, the new values of front and rear are increased and reaches to 0 (zero). This is the only stage where front and rear have same values.



In the above figure, two elements are appended in the queue. At this moment, the value of rear = 2 and front = 0.



In the above figure one element is deleted and the value of rear = 2 and f = 1. The value of front is increased due deletion of one element.



In the above figure, two more elements are appended to queue. The value of rear = 4 and front = 1.
If more elements are deleted, the value of front will be increased and as a result, the beginning portion of queue will be empty. In such a situation if we try to insert new elements to queue, no elements can be inserted in the queue. This is because, new elements are always inserted from the rear and if the last location of queue (rear) holds an element, even though the space is available at the beginning of queue, no elements will be inserted. The queue will be treated as overflow.
To overcome this problem, we have to update the queue. The solution to this problem is circular queue.

Operations :

Three operations can be performed on a queue.

1. insert(x)- Inserts item x at the rear of the queue q.
2. x=remove(); Deletes the front element from the queue and sets x to its contents.
3. empty()- Returns false or true depending on whether or not the queue contains any elements.
4. Full()-Returns false or true depending on whether or not the queue contains maximum elements.
5. Element_at_rear()-print element at rear.
6. Element_at_front()-print element at front.

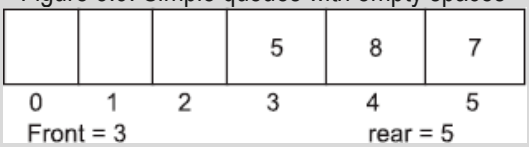
TYPES OF QUEUES

In the last few topics, we have studied simple queue and already seen the disadvantages. When rear pointer reaches to the end of the queue (array), no more elements can be added in the queue, even if beginning memory locations of array are empty. To overcome the above disadvantage, different types of queues can be applied. Different types of queues are:

3.8.1 Circular Queue

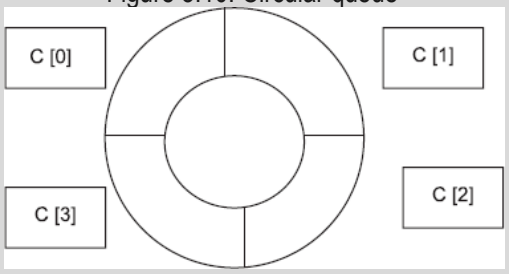
The simple or in a straight line queue there are several limitations we know, even if memory locations at the beginning of queue are available, elements cannot be inserted as shown in the Fig. 3.9. We can efficiently utilize the space available of straight-line queue by using circular queue.

Figure 3.9. Simple queues with empty spaces



In the above discussion, it is supposed that front is not shifted to the beginning of the queue. As shown in the Fig. 3.10 a circular queue is like a wheel or a perfect ring. Suppose, c [4] is an array declared for implementation of circular queue. However, logically elements appear in circular fashion but physically in computer memory, they are stored in successive memory locations.

Figure 3.10. Circular queue



In the circular queue, the first element is stored immediately after last element. If last location of the queue is occupied, the new element is inserted at the first memory location of queue implementing an array (queue). For example, C[n], C is queue of n elements. After inserting, storing an element in the last memory location [(n-1th) element], the next element will be stored at the first location, if space is available. It is like a chain of where starting and ending points are joined and a circle is formed. When an element is stored in the last location wrapping takes place and element inserting routine of the program pointed to beginning of the queue.

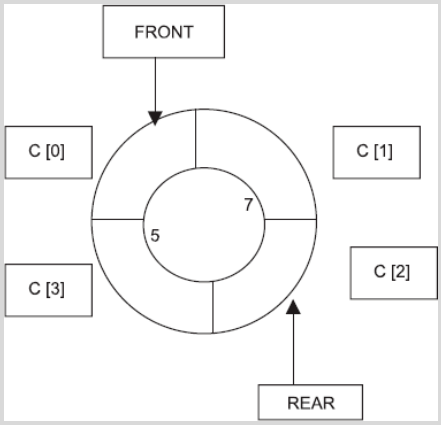
Recall that a pointer (stack and queue pointer) plays an important role to know the position of the elements in the stack and queue. Here, as soon as last element is filled, the pointer is wrapped to the beginning of the queue by shifting the pointer value to one.

A circular queue overcomes the limitation of the simple queue by utilizing the entire space of the queue. Like a simple queue, the circular queue also have front and rear ends. The rear and front ends help us to know the status of the queue after the insertion and deletion operations. In the circular queue implementation, the element shifting operation of queue that we apply in the simple queue is not required. The pointer itself takes care to move at vacant location of the queue and element is placed at that location. In order to simulate the circular queue a programmer should follow the following points:

1. The front end of the queue always points to the first element of the queue.
2. If the values of front and queue are equal, the queue is empty. The values of front and rear pointers are only increased / decreased after insertion/deletion operation.
3. Like simple queue, rear pointer is incremented by one when a new element is inserted and front pointer is incremented by one when an element is deleted.

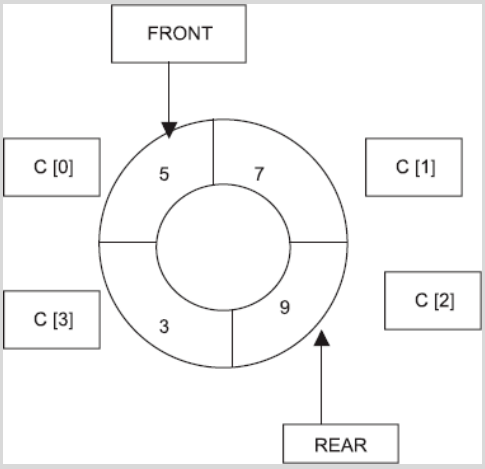
Insertion

Figure 3.11(a). Insertion in a queue



As per Fig. 3.11(a), the insertion of an element in a queue will be same as in a linear queue. The programmer must have to trace the values of front and rear variables.
In Fig. 3.11 (a), only two elements are there in the queue. If we continue to add elements, the queue would be as shown in Fig. 3.11 (b). The new element is always inserted from the rear end. The position where the next element is to be placed can be calculated by the following formula.
REAR = (1+*REAR*) % *MAXSIZE*
C[REAR] = *NUMBER*

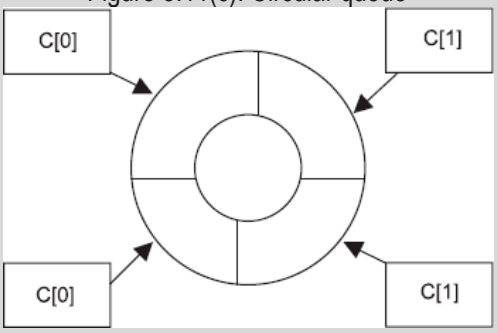
Figure 3.11(b). Full queue



From the Fig. 3.11(b), the value of rear is three and the capacity to store maximum element of queue is four. Therefore,
REAR = (1+*REAR*) % *MAXSIZE*
REAR = (1+3)%4
REAR = 4%4 = 0.

The operator % is modular divisor operator and returns remainder. Thus, in the above equation the remainder obtained is zero. The value of front as well as rear is zero, it means stack is full. Any attempt to insert new element will display the "queue full" message.
Consider, the following Fig. 3.11 (c),

Figure 3.11(c). Circular queue

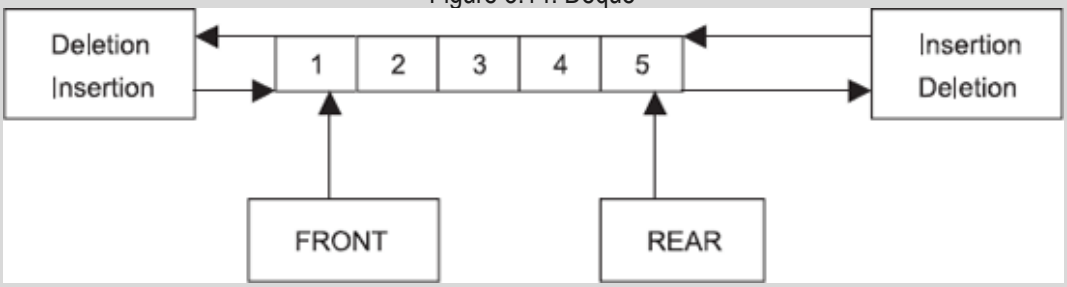


Double Ended Queues

Till now, we have studied stack and queue. The stack has only one end and can be used alternately to insert and delete elements. On the other hand, the double-ended queue has two ends, front and rear. The front end is used to remove element and rear end is used to insert elements. Here, in the double-ended queues, insertion and deletion can be performed from both the ends and therefore, it is called as double-ended queue and in short deque. It is a homogenous list of elements. The deque is a general representation of both stack and queue and can be used as stack and queue. There are various methods to implement deque. Linked list and array can be used to perform the deque. The array implementation is easy and straightforward.

Consider the following Fig. 3.14:

Figure 3.14. Deque



In deque, it is necessary to fix the type of operation to be performed on front and rear ends. The deque can be classified into two types:

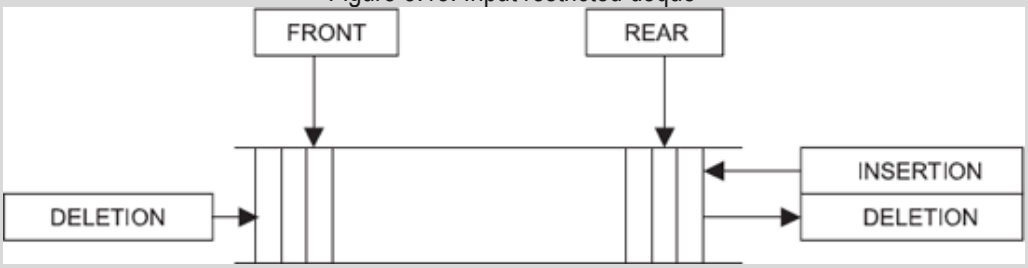
Input Restricted Deque

In the input restricted deque, insertion and deletion operation are performed at rear end whereas only deletion operation is performed at front end. The Fig. 3.15 represents the input restricted deque.

The following operations are possible in the input restricted deque:

1. Insertion of an element at the rear end.
2. Deletion of element at both front and rear ends.

Figure 3.15. Input restricted deque

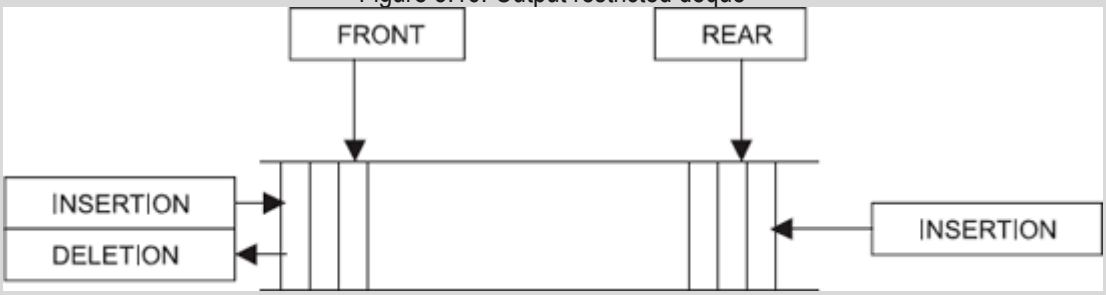


Thus, input restricted deque allows insertion at one and deletion at both ends.

Output Restricted Deque

In the output restricted deque insertion of an element can be done at both front and rear end and deletion operation can be done only at front end. The output restricted deque is shown in Fig. 3.16.

Figure 3.16. Output restricted deque



The following operations are possible in the output restricted deque:

1. Insertion at both front and rear end.
2. Deletion at only front end.

Priority Queues

We know that queue is based on the technique first come first out (FIFO). The element inserted first will be deleted first. A priority queue is another type of queue. Here, the priority is determined according to the position of the queue, which is to be entered. Various real life problems are based on priority queues. For example, in the buses few seats are reserved for ladies and handicapped; if a company is providing any scheme, the employees of the same company are given second priority.

Priority queue is a data structure in which prioritized insertion and deletion operations on elements can be performed according to their priority values.

There are two types of priority queues:

Ascending Priority Queue

Descending Priority Queue

In this queue also, elements can be inserted randomly but the largest element is deleted first.

In both the above types, if elements with equal priority are present, the FIFO technique is applied. In case the queue elements are sorted, the deletion of an element will be quick and easy because elements will appear either in ascending or descending order. However, the insertion operation will be easier said than done because empty locations will have to be searched and only then elements can be placed at that location.

In case the queue elements are not in order, i.e. not sorted, insertion operation will be quick and easy. However, the deletion operation will take place after the priority value set.

Therefore, we can say that in both the above types insertion operation can be carried out easily. However, the deletion operation, which is, based on certain priority, i.e. the smallest or largest is first searched out and later it is removed from the queue. The locations of that element do not matter. In stack and queues deletion operation is performed at the ends. Conceptually, there is no provision for deleting an element, which is neither first nor last element of the list.

In the above program, we have not arranged the elements in an ascending or descending order. Just smallest or largest element is searched and erased.

In deletion operation the element may not be physically removed from the queue and it can be kept inaccessible in the program. Alternatively, special character called empty indicator such as #, $ can be placed at that place. Both the insertion and deletion operation perform scanning of the queue elements. While performing deletion operation the element must be deleted physically. It is possible to make its access denied. However, for other computation operation, the element logically deleted but physically

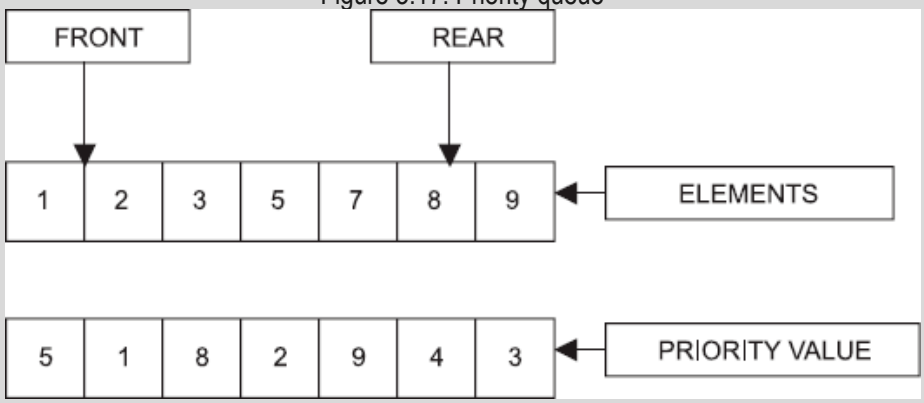
existing will be taken into account and will change the result. So, for sure result, the element must be removed.

The elements of the queue can be number, character or any complex object which is a composition of one or more basic data types.

In priority queue every element has been assigned with a priority value called priority. The elements can be inserted or deleted randomly anywhere in the queue. Consider the following points of queue:

1. An element of upper priority is processed prior to an element of lower priority.
2. If two elements have the same priority, they are processed depending on the order they are inserted in the queue, i.e. FIFO.

Figure 3.17. Priority queue



As shown in Fig. 3.17, the element 7 is deleted first. However, the element 1 is nearer to the front as compared to 7, but 7's priority value is nine, which is higher, and hence it is deleted first. Though the queue is based on FIFO technique, the priority queue is not based on FIFO operation firmly. A program on priority queue is provided below for detailed understanding.

APPLICATIONS OF QUEUES

There are various applications of computer science, which are performed using data structure queue. This data structure is usually used in simulation, various features of operating system, multiprogramming platform systems and different type of scheduling algorithm are implemented using queues. Round robin technique is implemented using queues. Printer server routines, various applications software are also based on queue data structure.

3.9.1 Round Robin Algorithm

Round Robin (RR) algorithm is an important scheduling algorithm. It is used especially for the time-sharing system. The circular queue is used to implement such algorithms.

For example, there are N procedures or tasks such as P1, P2, P3 ... P_N. All these tasks are to be executed by the central processing unit (CPU) of the computer system. The execution times of the tasks or processes are different. The tasks are executed in sequence P1, P_2, P_3 and P_N.

In time, sharing mode the tasks are executed one by one. The algorithm forms a small unit of time, say, from 10 to 100 milliseconds for each task. This time is called time slice or time quantum of a task. The CPU executes tasks from P_1 to P_N allocating fixed amount of time for each process. On allocating time to all tasks, CPU resumes the first task. That is, when all tasks are completed, it returns to P_1. In the time-sharing system, if any task is completed before the estimated time, the next task is taken up for execution immediately. Consider the following table:

Tasks	Expiry Time (Units)
P1	10
P2	19
P3	8

P4	5

There are total four tasks and the total time to complete all the tasks would be (10+19+8+5) 42 units. Suppose, the time slice is of 7 minutes. The RR scheduling for the above case would be
In the first pass each task takes seven units of time-task P4 is executed in the first pass whereas task P1, P2, P3 requires more than 7 units of time. Hence, in the second round task P1 and P3 are executed. At last, P2 is executed.

3.9.2 Simulation

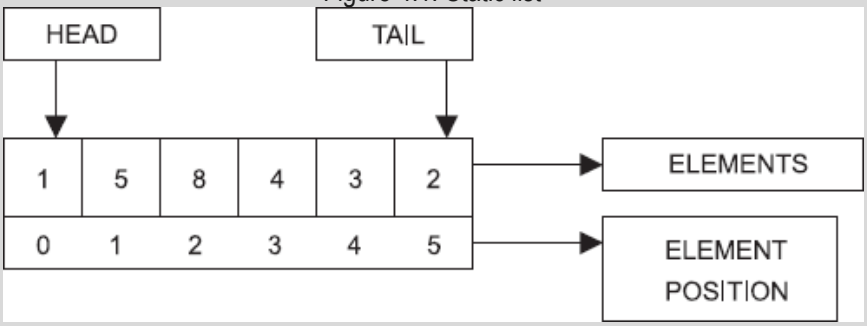
It is an extremely powerful tool used for experimentation purpose. Without performing real experiments simulation permits to take the results and if necessary modification can be done as per expected results. One of the standard applications of queue can be implemented with simulation. Simulation is a method of managing a theoretical illustration of a real life problem with the purpose of understanding the effect of modification, concern factors and implementing some approaches to get reliable solution. The main goal of simulation is to help the user or to guess for obtaining the output of the program after implementing some approaches. It permits the user to make several trials for getting the actual results and planned situations without disturbing the real situation.

There are few disadvantages in the simulation. Lengthy simulation creates execution overhead on computer. For solving a real life problem, various assumptions are made to find out the key solution. If the assumption is straightforward, the outcome will be less reliable. In contrast, if the assumption is having more particulars, the outcome will be reliable and better. If the primary assumptions are wrong, even though, the program source code is given in detail, the guess obtained will be incorrect. In this situation, the program developed becomes futile.

Chapter 4
Linked list

A list is a series of linearly arranged finite elements (numbers) of same type. The data elements are called nodes. The list can be of two types, i.e. basic data type or custom data type. The elements are positioned one after the other and their position numbers appear in sequence. The first element of the list is known as head/root and the last element is known as tail.

Figure 4.1. Static list



As shown in the above Fig. 4.1, the element 1 is at head position (0th) and element 2 is at tail position (5th). The element 5 is predecessor of element 8 and 4 is successor. Every element can act as predecessor excluding the first element because it does not have predecessor in the list. The list has following properties:

a. The list can be enlarged or reduced from both the ends.
b. The tail (ending) position of the list depends on how long the list is extended by the user.
c. Various operations such as transverse, insertion and deletion can be performed on the list.
d. The list can be implemented by applying static (array) or dynamic (pointer) implementation.

IMPLEMENTATION OF LIST

There are two methods of implementation of the list: they are static and dynamic.

4.2.1 Static Implementation

Static implementation can be implemented using arrays. It is a very simple method but it has few limitations. Once a size is declared, it cannot be changed during the program. It is also not efficient for memory. When array is declared, memory allocated is equal to the size of the array. The vacant space of array also occupies the memory space. In both the cases, if we store less arguments than declared, the memory is wasted and if more elements are stored than declared, array cannot be expanded. It is suitable only when exact number of elements are to be stored.

4.2.2 Dynamic Implementation

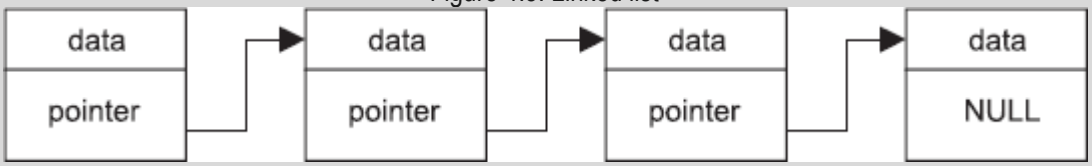
The linked list is a major application of the dynamic implementation and the pointers are used for the implementation. The limitations noticed in static implementation can be removed by using dynamic implementation. The memory is utilized efficiently in this method. Hence, it is superior than the static implementation. With pointers, link does not have any restriction on number of elements at run time. The list can be stretched like elastic. Memory is allocated only after node is added or element is accepted in the list. Memory is de-allocated whenever node or element is removed. Dynamic memory management

policy is used in the implementation of linked list and because of this memory is used resourcefully. In addition to that insertion and deletion of a node can be done easily.

LINKED LIST

A linked list is a dynamic data structure. It is an ideal technique to store data when the user is not aware of the number of elements to be stored. The dynamic implementation of list using pointers is also known as linked list. Each element of the list is called as node. Each element points to the next element. In the linked list a node can be inserted or deleted at any position. Each node of linked list has two components. The first component contains the information or any data field and second part the address of the next node. In other words, the second part holds address of the next element. This pointer points to the next data item. The pointer variable member of the last record of the list is generally assigned a NULL value to indicate the end of the list. Fig. 4.5 indicates the linked list.

Figure 4.5. Linked list



The basic data type in the linked list can be int, float and user defined data types can be created by struct call. The link structure as well as a pointer of structure type to the next object in the link is observed in it.

IMPORTANT TERMS

We have already discussed in previous sections that a linked list is a non-sequential collection of elements called nodes. These nodes are nothing but objects. These nodes are declared using structure or classes. Every node has two fields and they are:

1. Data field: In this field, the data or values are stored and processed.
2. Link field: This field holds address of the next data element of the list. This address is used to access the successive elements of the list.

In the linked list the ordering of elements is not done by their physical location in the memory but by logical links, which are stored in the link field.

Node

The components or objects which form the list are called nodes.

Null Pointer

The link field of the last record is assigned a NULL value instead of any address. It means NULL pointer not pointing to any element.

External Pointer

It is a pointer to the starting node. The external pointer contains the base, i.e. address of first node. Once a base address is available, its next successive nodes can be accessed.

Em pty List

When there is no node in the list, it is called as empty list. If the external pointer were assigned a value NULL, the list would be empty.

Types of Linked List

The linked lists are classified in the following types:

SINGLY LINKED LIST

Recall that linked list is a dynamic data structure with ability to expand and shrink as per the program requirement. The singly linked list is easy and straightforward data structure as compared to other structures. By changing the link position other type of linked list such as circular, doubly linked list can be formed. For creating linked list the structure is defined as follows,

```
struct node
{
 int number;
 struct node *p;
};
```

The above structure is used to implement the linked list. In the number, variable entered numbers are stored. The second member is pointer to the same structure. The pointer *p points to the same structure. Here, though the declaration of struct node has not been completed, the pointer declaration of the same structure type is permitted by the compiler. However, the variable declaration is not allowed. This is because, the pointers are dynamic in nature whereas variables are formed by early binding. The declaration of objects inside the struct leads to preparation of very complex data structure. This concept is called object composition and its detailed discussion is out of the scope of this book.

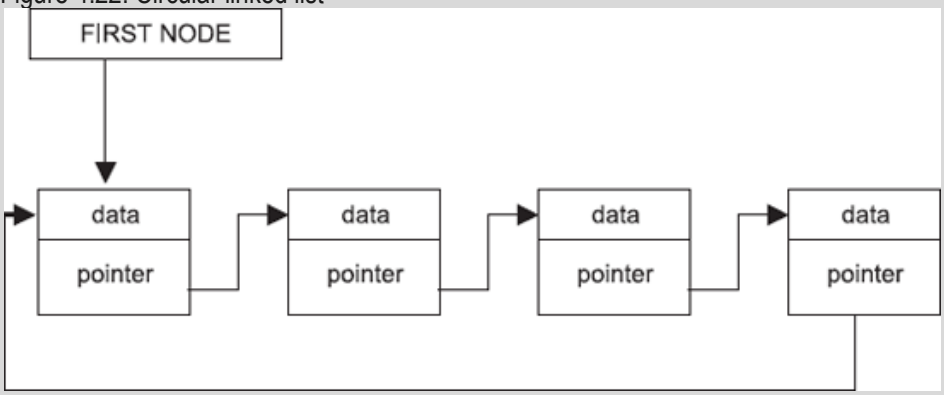
We are familiar with the array and we know the importance of base address. Once a base address is obtained, successive elements can also be accessed. In the linked list, list can be created with or without header node. The head holds the starting address.

Figure 4.8. Singly linked list

CIRCULAR LINKED LIST

In circular linked list the last node points to the header node. The linear link list can be converted to circular linked list by linking the last node to the first node. The last node of the linear linked list holds NULL pointer, which indicates the end of linked list but performance of linked list can be advanced with minor adjustment in the linear linked list. Instead of placing the NULL pointer, the address of the first node can be given to the last node, such a list is called circular linked list (as shown in Fig. 4.22).

Figure 4.22. Circular linked list



The circular linked list is more helpful as compared to singly linked list. In the circular linked list, all the nodes of the list are accessible from the given node. Once a node is accessed, by traversing all the nodes can be accessed in succession.

In this type of list, the deletion operation is very easy. To delete an element from the singly linked list, it is essential to obtain the address of the first node of the list. For example, we want to delete the element, say 5, which exists in the middle of the list. To remove the element five, we need to find predecessor of five. Obviously, a particular element can be searched using searching process in which all elements are visited and compared. In the circular linked list, no such process is needed. The address of predecessor

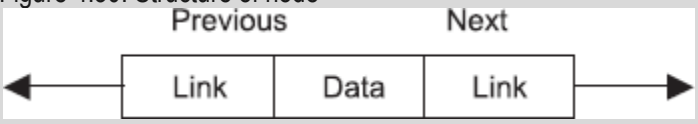
can be found from the given element itself. In addition, the operation splitting and concatenation of the list (discussed later) are easier.

Though the circular linked list has advantages over linear linked list, it also has some limitations. This list does not have first and last node. While traversing the linked list, due to lack of NULL pointer, there may be a possibility to get into an infinite loop. Thus, in the circular linked list it is necessary to identify the end of the list. This can be done by setting up the first and last node by convention. We detect the first node by creating the list head, which holds the address of the first node. The list head is also called as external pointer. We can also keep a counter, which is incremented when nodes are created and end of the node can be detected. The Fig. 4.23 gives an example of circular linked list with header.

DOUBLY LINKED LIST

The singly linked list and circular linked list contain only one pointer field. Every node holds an address of next node. Thus, the singly linked list can traverse only in one direction, i.e. forward. This limitation can be overcome by doubly linked list. Each node of the doubly linked list has two pointer fields and holds the address of predecessor and successor elements. These pointers enable bi-directional traversing, i.e. traversing the list in backward and forward direction. In several applications, it is very essential to traverse the list in backward direction. The pointer pointing to the predecessor node is called left link and pointer pointing to successor is called right link. A list having such type of node is called doubly linked list. The pointer field of the first and last node holds NULL value, i.e. the beginning and end of the list can be identified by NULL value. The structure of the node is as shown in Fig. 4.30.

Figure 4.30. Structure of node

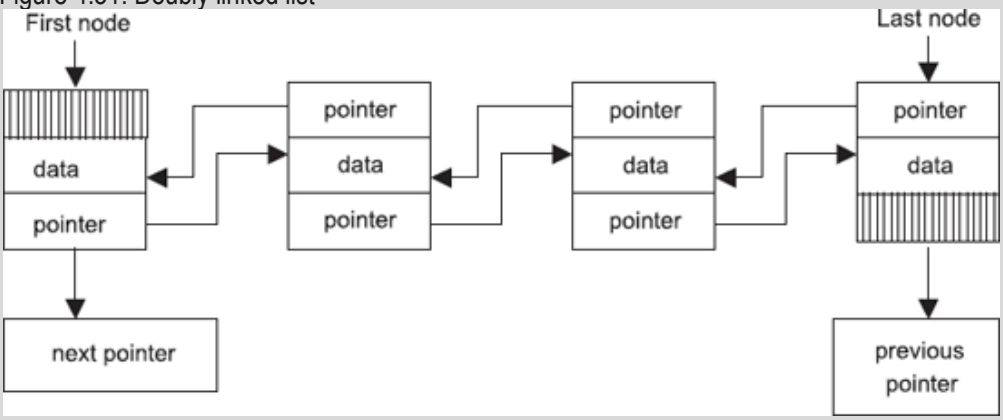


The structure of node would be as follows:

```
struct node
{
  int number;
  struct node *llink
  struct node *rlink;
}
```

the above structure can be represented by using the Fig. 4.31.

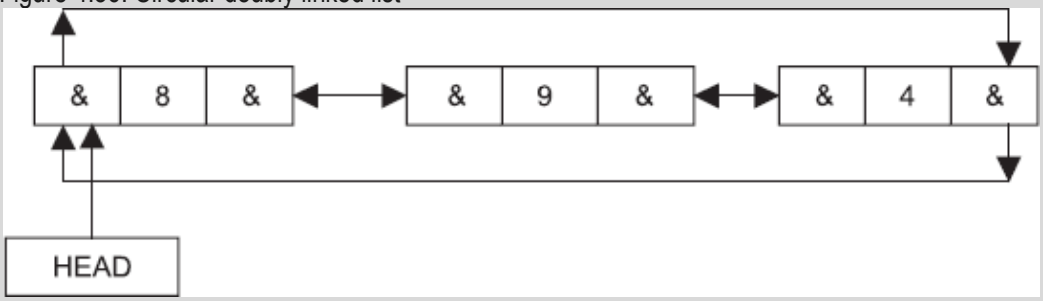
Figure 4.31. Doubly linked list



CIRCULAR DOUBLY LINKED LIST

A circular doubly linked list has both successor and predecessor pointers. Using the circular fashioned doubly linked list the insertion and deletion operation, which are little complicated in the previous types of linked list are easily performed. Consider the following Fig. 4.36:

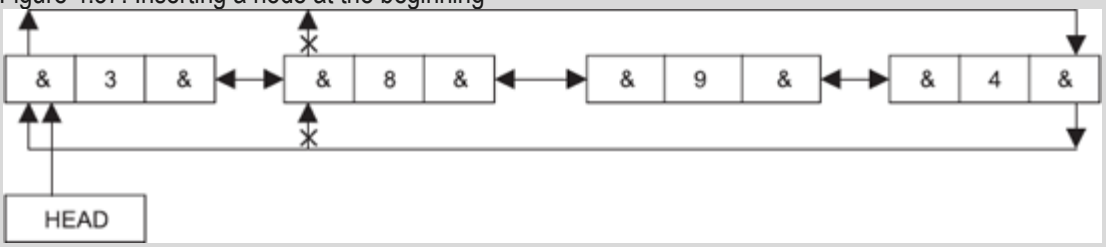
Figure 4.36. Circular doubly linked list



4.27.1 Insertion and Deletion Operation

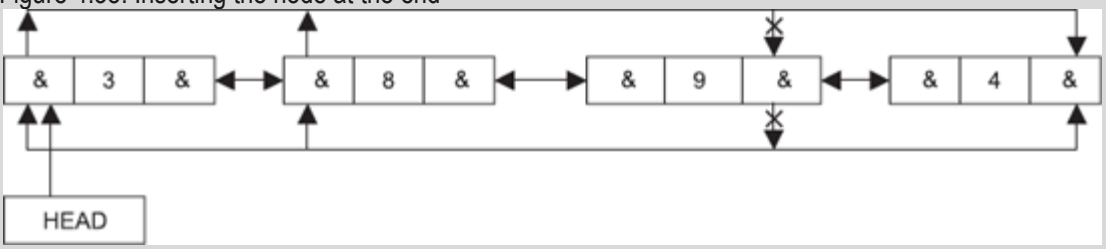
The insertion operation is similar to what we have already learnt in previous types. The Only difference is the way we link the pointer fields. Consider Fig. 637.

Figure 4.37. Inserting a node at the beginning



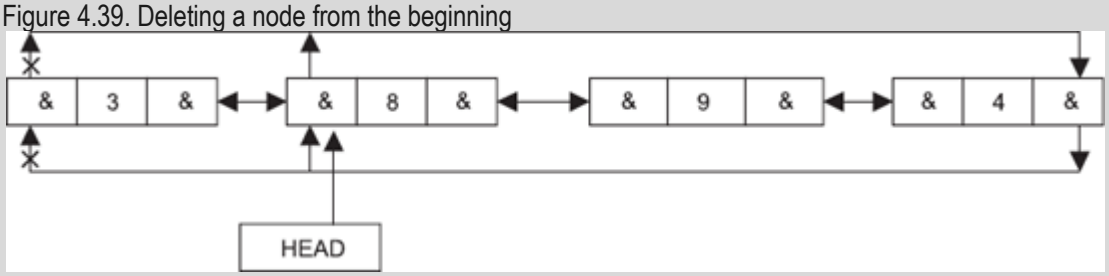
The × indicates that the previous links are destroyed. The pointer links from ex-first node are removed and linked to new inserted node at the beginning. The element 8 was previous first node and 3 is the new node inserted and becomes first node now after inserting it at the beginning.

Figure 4.38. Inserting the node at the end



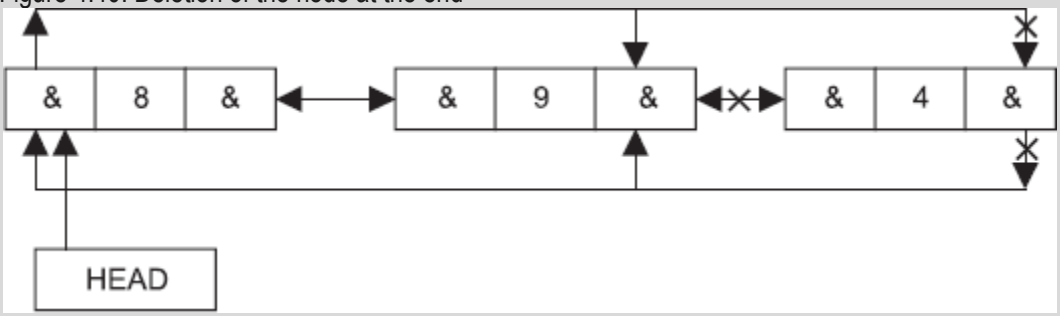
The × indicates, that the previous links are destroyed. The pointer links from ex-last node are removed and linked to new inserted node at the end. The address of last node is given to first node to form circular list. The node 9 was previously the last node but after insertion of the node at the end, newly inserted node is the last node. Fig. 4.39 shows the deletion of the node at beginning.

Figure 4.39. Deleting a node from the beginning



The × indicates that the previous links are destroyed. After deletion of the first node, the second node becomes first node. The pointer head also points to the newly appeared first node. Thus, when a node from the beginning is removed, the node followed by it will become the head node (first node). Accordingly, pointer adjustment is performed in the real application.

Figure 4.40. Deletion of the node at the end



The × as usual is the symbol destroying previous links. When the last node is removed, one node will be the last node and its links are established with the first node. The removal operation is shown in Fig. 4.40.

MEMORY ALLOCATION AND DE-ALLOCATION

Malloc (): This library function is used to allocate memory space in bytes to the variable. The function reserves bytes of requested size and returns the base address to pointer variable. The prototype of this function is declared in the header file alloc.h and stdlib.h. One of the header files must be included in the header. The syntax of the malloc() function is as follows:

Syntax: pnt=(data type*) malloc(size);

Here, pnt is a pointer.

Example: pnt =(int *) malloc(20);

In the above statement, 20 bytes are allocated to integer pointer pnt. In addition, there are other various memory allocation functions and its complete description is out of the scope of this book.

free(): This function is used to release the memory allocated by the malloc () function.

Syntax: free(pointer variable)

Example: free(pnt)

The above statement releases the memory allocated to pointer pnt.

OPERATIONS ON LINKED LISTS

The following primitive operations can be performed with linked list:

1. Creation
2. Display
    a. Ascending
    b. Descending
3. Traversing
4. Insertion
    a. At beginning
    b. Before or after specified position
5. Searching
6. Concatenation
7. Merging

Creation

The linked list creation operation involves allocation of structure size memory to pointer of the same structure. The structure must have a member which points recursively to the same structure. In this operation, constituent node is created and it is linked to the link field of preceding node.

Traversing

It is the procedure of passing through (visiting) all the nodes of the linked list from starting to end. When any given record is to be searched in the linked list, traversing is applied. When the given element is found, the operation can be terminated or continued for next search. The traversing is a common procedure and it is necessary because operations like insertion, deletion, listing cannot be carried out without traversing linked list.

Display

The operation in which data field of every node is accessed and displayed on the screen. In the display operation from beginning to end, link field of every node is accessed which contains address of next node. The data of that field is displayed. When NULL is detected the operation ends. Each node points to the next node and this recursion fashion enables the pointer to reach the successive elements.

The above three operations are common and every program involves these operations. Hence, separate program is not given.

Searching

The searching is a process, in which a given element is compared with all the linked list elements. The if statement is placed and it checks entire list elements with the given element. When an element is found it is displayed.

Besides the above operations additional operations such as concatenation and merging of list can be done.

APPLICATIONS OF LINKED LIST

The most useful linear data structure is linked list. This section introduces you to a few applications of linked lists which are useful in computer science.

1 Polynomial Manipulation

A polynomial can be represented and manipulated using linear link list. Various applications on polynomials can be implemented with linked lists. We perform various operations such as addition, multiplication with polynomials. To get better proficiency in processing, each polynomial is stored in decreasing order. These arrangements of polynomial in series allow easy operation on them. Actually, two polynomials can be added by checking each term. The prior comparison can be easily done to add corresponding terms of two polynomials.

A polynomial is represented with various terms containing coefficients and exponents. In other words, a polynomial can be expressed with different terms, each of which comprises of coefficients and exponents. The structure of a node of linked list for polynomial implementation will be as follows. Its pictorial representation is shown in Fig. 4.41.

Figure 4.41. A term of a polynomial

The coefficient field contains the value of coefficient of the term. Similarly, the exponent field contains the value of exponent. As usual, the link field points to the term (next node).

The structure for the above node would be as follows:

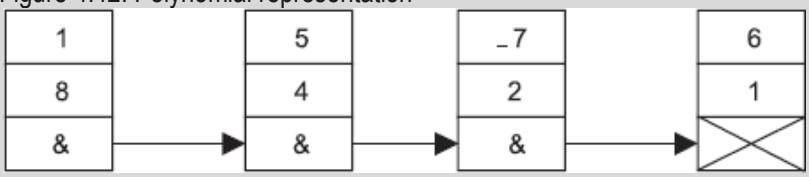
```
struct poly
{
  double coeff;
  int exp;
  struct poly *next;
};
```

Consider a polynomial

$P = P^8 + 5P^4 - 7P^2 + 6P$

In this equation 1,5,7 and 6 are coefficients and exponents are 8,4,2 and 1. The number of nodes required would be the same as the number of terms in the polynomial. There are four terms in this polynomial hence it is represented with four nodes.

Figure 4.42. Polynomial representation



The top of every node represents coefficients of the polynomial, exponents are at the centre and the pointers are (next) at the bottom. The terms are stored in order of descending exponent in the linked list. It is assumed that no two terms have the similar exponents. Fig. 4.42 shows the polynomial.

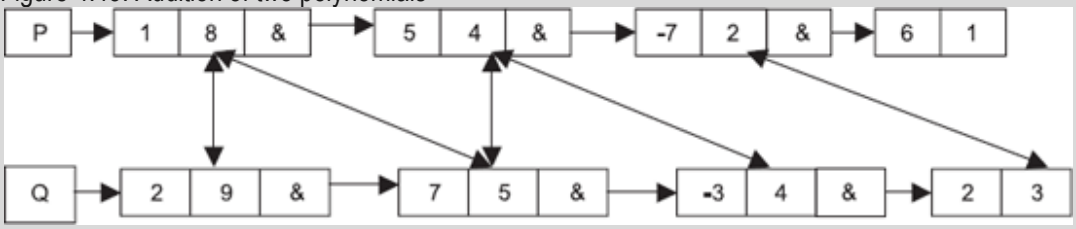
Consider the following equations,

$P = P^{P8}P + 5P^{P4}P - 7P^{P2}P + 6P$

$Q = 2P^{P9}P + 7P^{P5}P - 3P^{P4} + 2P^{3}$

The representation of the above two polynomials can be shown in Fig. 4.43.

Figure 4.43. Addition of two polynomials



If the term is shown without coefficient then the coefficient is assumed to be 1. In case the term is without variable, the coefficient is zero. Such terms are not required to be stored in the memory. The arrow in the figure indicates that the two exponents are compared. Where the link of arrow is disconnected, exponent of both the terms is same. The term, which has no touch of any arrow, means it is the last node and inserted in the resulting list at the end. The arrow indicates two terms, which are compared systematically from left to right.

Table 4.1. Exponent

Exponent comparison from list p and Q	List R (Inserted exponent)	Smaller exponent
8<9	9	8 is carried forward
8>5	8	5 is carried forward
5>4	5	4 is carried forward
4= =4	4	No term is carried
	(Sum of coefficient is taken)	
2<3	3	2 is carried forward
1 is taken (last node of P)	1	End of linked list

Traverse the list p and Q. Compare the corresponding terms of list p and Q. In case one node has larger exponent value than the other then insert the larger exponent node in the third list and forward the pointer to next node of the list whose current term is inserted in the third list. The pointer of the list whose exponent is smaller will not be forwarded. The pointer of the lists forwarded only when the current nodes from the lists are inserted into the third list. Table 4.1 shows these operations.

If exponents are equal, add the coefficients and insert their addition in the third list. In this step, exponents from both the expressions are same, move the pointer to next node in both the list p and Q. Repeat the same process until one list is scanned completely.

expo(P) = expo(Q).

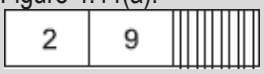
The possible conditions can be stated as follows:

1. If exponent of list p is greater than corresponding exponent of list Q, insert the term of p into the list R and forward the pointer in list p to access the next term. In this case, the pointer in the list Q will point to the same term (will not be forwarded).
2. If exponent of list Q is greater than exponent P, the term from Q is inserted in the list R. The pointer of list Q will be forwarded to point to next node. The pointer in the list p will remain at the same node.
3. If exponents of both nodes are equal, addition of coefficients is taken and inserted in the list R. In this case, pointers in both the lists are forwarded to access the next term.

The steps involved,

1. Traverse the two lists (P) and (Q) and inspect all the elements.
2. Compare corresponding exponents p and Q of two polynomials. The first terms of the two polynomials contain exponents 8 and 9, respectively. Exponent of first term of first polynomial is smaller than the second one. Hence 8<9. Here, the first exponent of list Q is greater than the first exponent of list P. Hence, the term having larger exponent will be inserted in the list R. The list R initially looks like as shown in Fig. 4.44(a).

Figure 4.44(a).



Next, check the next term of the list Q. Compare it with the present term (exponent) of list P. The next node from p will be taken when current node is inserted in the list R because 8>5. Here, the exponent of current node of list p is greater that list Q. Hence, current term (node) of list p will be inserted in the list R and the list R becomes as Fig. 4.44 (b).

Figure 4.44(b).

After moving to next node in list P, the exponent is 4, and exponent of Q is 5. Compare 4 with 5. Of course 4<5. Here, the term of list Q is greater than term of P. Therefore, the term of list, Q (7,5) will be inserted to list R. The list R becomes Fig. 4.44 (c).

Figure 4.44(c).

In this step a node from list Q is inserted and therefore, the pointer in the list Q will be forwarded and point to the term (–3,4) and from list p we have the current node (5,4). Compare exponent of these two terms. The condition observed here is 4==4. Here, exponents of both the terms are equal. Therefore, addition of coefficients is taken and result is inserted in the list R. The addition is 2 (5–3). The list R becomes as the Fig. 4.44 (d).

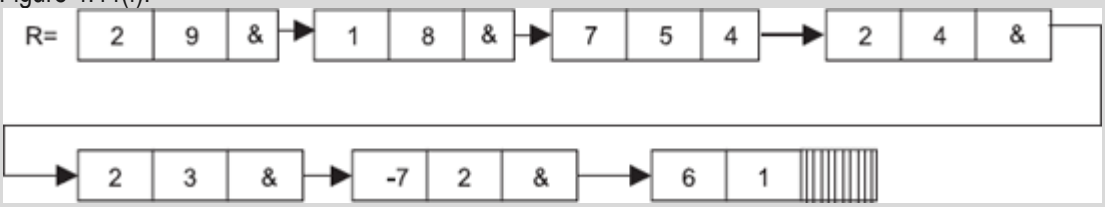
Figure 4.44(d).

Move forward the pointers to next nodes in both the lists, since, the previous terms were having same exponents. The next comparison is 2<3. Here, the exponent of current node of list Q is greater than of P. The node from Q will be inserted to list R. The list R will be shown as Fig. 4.44(e).

Figure 4.44(e).

The list Q is completely scanned and reached to end. The remaining node from the list p will be inserted to list R. The list R is as Fig. 4.44(f).

Figure 4.44(f).



CHAPTER 5
TREES

INTRODUCTION

In the previous chapters, we have studied various data structures such as arrays, stacks, queues and linked list. All these are linear data structures. In these data structures the elements are arranged in a

linear manner, i.e. one after another. Tree is an equally useful data structure of non-linear type. In tree, elements are arranged in non-linear fashion. A tree structure means data is organized in branches. The following Fig. 5.1 is a sample tree.

A tree is a non-linear data structure and its elements are arranged in sorted order.
Tree has several practical applications. It is immensely useful in manipulating data and to protect hierarchical relationship among data. The fundamental operations such as insertion, deletion etc. is easy and efficient in tree data structure than in linear data structures. Fig. 5.2 represents family hierarchy, which keeps relations among them. The hierarchy gives relations between associates of family members. In this tree, node 'A' can be assumed as a parent of 'B' and 'C', 'D' and 'E' are the children of 'B'. 'F' is the child of 'C'. This tree represents the relation between the family members.

Figure 5.1. Sample tree

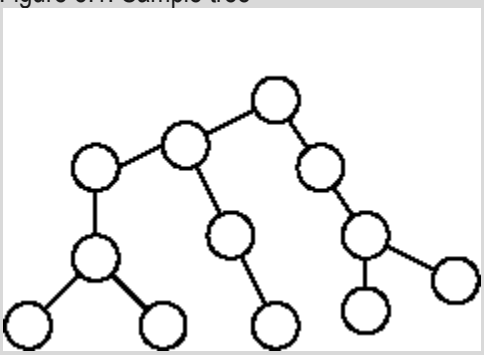
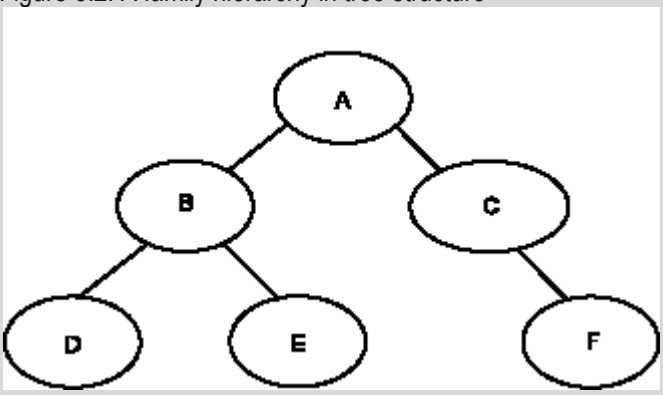


Figure 5.2. A family hierarchy in tree structure



For example, suppose the left side member is male and right side member is female. Then, various relations such as sister, brother, grandfather, grandmother can also be implied.
The algebraic expression can be represented with a tree. Consider the following example of an algebraic expression.

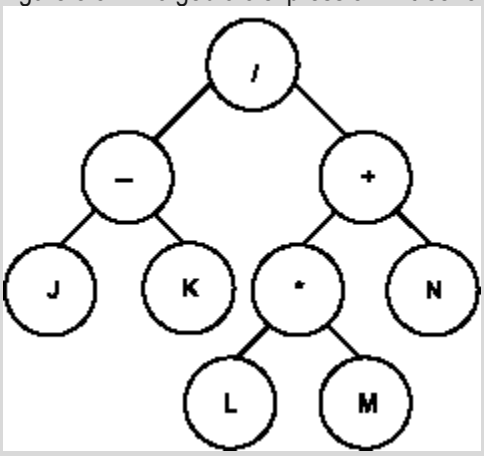
Z=(J-K)/((L*M)+N)

The operators of the above equation have different priority. The priority of the operators can be represented by the tree structure. The operators of high priority are at low level and operator and associated operands are represented in tree structure. The Fig. 5.3 illustrates the representation of an algebraic expression.

BASIC TERMS

Some of the basic concepts relevant to trees are described in this section. These are node, parents, roots, child, link, leaf, level, height, degree of node, sibling, terminal nodes, path length, and forest.

Figure 5.3. An algebraic expression in tree form



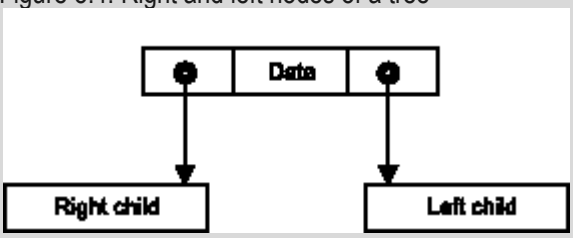
Root

It is the mother node of a tree structure. This is the most important node of any tree. This node does not have parent. It is the first node in the hierarchical arrangement.

Node

It is the main component of the tree. The node of a tree stores the data and its role is same as the linked list. Nodes are connected by means of links with other nodes. This is shown in Fig. 5.4.

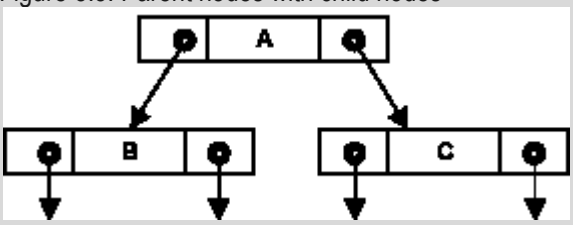
Figure 5.4. Right and left nodes of a tree



Parent

It is an immediate predecessor of a node. In the Fig. 5.5 A is parent of B and C.

Figure 5.5. Parent nodes with child nodes



Child

When a predecessor of a node is parent then all successor nodes are called child nodes. In Fig. 5.5, B and C are child nodes of A. The node at left side is called left child node and node at right side is called right child node.

Link

The link is nothing but pointer to node in a tree structure. In other words, link connects the two nodes. The line drawn from one node to other node is called a link. Fig. 5.5 shows left and right child. Here, two links are shown from node A. More than two links from a node may be drawn in a tree. In a few textbooks the term edge is used instead of link. Functions of both of them are same.

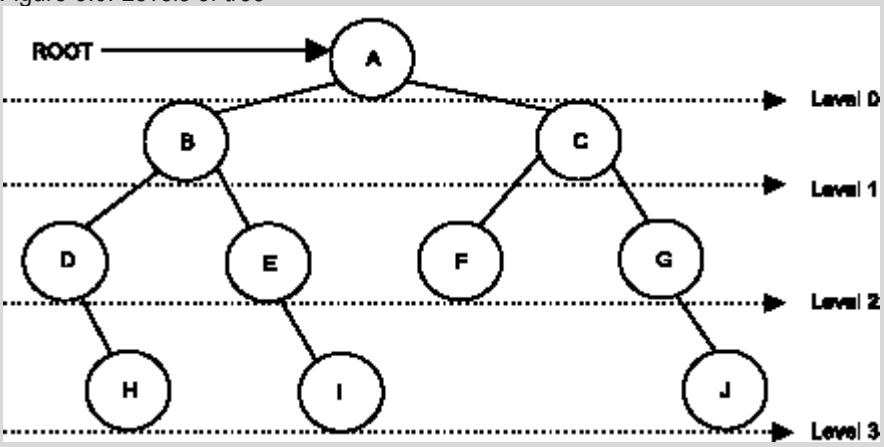
Leaf

This node is located at the end of the tree. It does not have any child hence it is called as leaf node. Here, 'H', 'I', 'F', and 'J' are leaf nodes in Fig. 5.6.

Level

Level is a rank of tree hierarchy. The whole tree structure is levelled. The level of root node is always at 0. The immediate children of root are at level 1 and their immediate children are at level 2 and so on. If children nodes are at level n+1 then parent node would be at level n. The Fig. 5.6 shows the levels of tree.

Figure 5.6. Levels of tree



Height

The highest number of nodes that is possible in a way starting from the first node (root) to a leaf node is called the height of tree. In Fig. 5.6, the height of tree is 4. This value can be obtained by referring three different paths from the source node to leaf node. The paths A-B-D-H, A-B-E-I, and A-C-G-J have the same height. The height can be obtained from the number of levels, which exists in the tree. The formula for finding the height of the tree $h = i_{max} + 1$, where h is the height and i_{max} is maximum level of the tree. In the above Fig. 5.6 the maximum level of the tree is 3(i_{max}=3). By substituting the value into the formula the h will be 4. The term depth can be used in place of height.

Degree of a Node

The maximum number of children that can exist for a node, is called as the degree of the node. In Fig. 5.6 the node A, B and C have maximum two Children. So, the degree of A, B and C is same and it is equal to 2.

Sibling

The child nodes of same parent are called sibling. They are also called brother nodes. A, B and C nodes in the Fig. 5.6 have two child nodes. B and C are the siblings of the node A, whereas D and E are the siblings of the node B.

Terminal Node

A node with degree zero is called terminal node or leaf. Fig. 5.6 shows 4 terminal nodes and they are H, I, F and J.

Path Length

It is the number of successive edges from source node to destination node. In the above Fig. 5.6 the path length from the root node A to H is three because there are three edges.

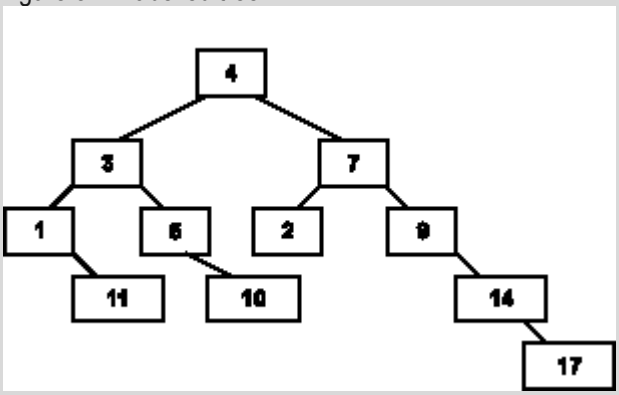
Forest

It is a group of disjoint trees. If we remove a root node from a tree then it becomes the forest. If we remove root node A then the two disjoint sub-trees will be observed. They are left sub-tree B and right sub-tree C.

Labelled Trees

In the labelled tree the nodes are labelled with alphabetic or numerical values. The labels are the values allotted to the nodes in the tree. Fig. 5.7 shows the diagram of a labelled tree. Finally obtained tree is called as labelled tree.

Figure 5.7. Labelled tree



The Fig. 5.7 shows the labelled tree having 11 nodes; root node is 4 and leaf nodes 11,10,17, and 2. The parent and children relationship between them is shown in Table 5.1.

Table 5.1. Parent and children relationship

Parent nodes	Children nodes
4	3,7
3	1,5
1	11
5	10
7	2,9
9	14
14	17

There is one more relationship, which is called left node left child and right node right child. This is useful for traversing the binary tree.

Table 5.2. Relationship between parent, left and right nodes

Parent node	Left child	Right child
4	3	7
3	1	5
1	-	11
5	-	10
7	2	9
9	-	14
14	-	17

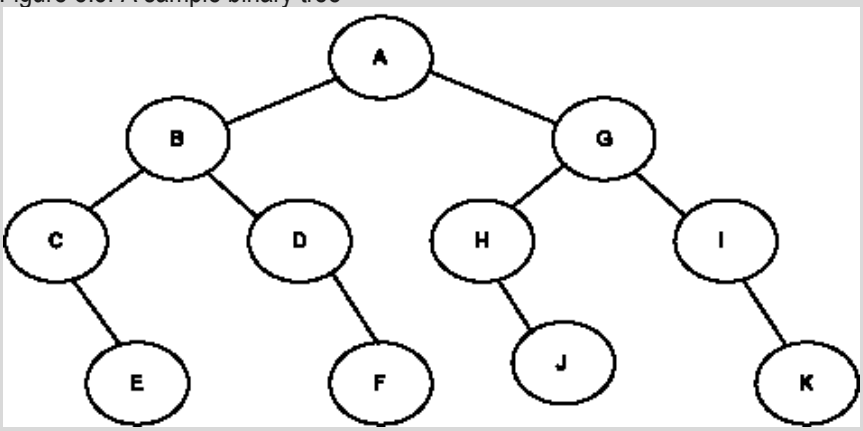
Table 5.2 describes the relationship between the parents, left and right nodes, which is drawn from the Fig. 5.7. The '—' indicate that root node (parent) does not have child node. The 11,10 and 17 labelled nodes are the leaf nodes of the tree.

BINARY TREES

A binary tree is a finite set of data elements. A tree is binary if each node of it has a maximum of two branches. The data element is either empty or holds a single element called root along with two disjoint trees called left sub-tree and right sub-tree, i.e. in a binary tree the maximum degree of any node is two. The binary tree may be empty. However, the tree cannot be empty. The node of tree can have any

number of children whereas the node of binary tree can have maximum two children. Fig. 5.8 shows a sample binary tree.

Figure 5.5. A sample binary tree



In Fig. 5.8, A is the root and B and G are its child nodes. The nodes B and G are non-empty nodes, hence they are called left successor and right successor of the root A. The node root without successor is called the terminal node of that root. In Fig. 5.8 the node E, F, J, and K are the terminal nodes.

The right tree is G and left tree is B. Next B has left tree C and right tree D. The right tree further has left tree H and right tree I. This will be continued up to last level.

COMPLETE BINARY TREE

A tree is called complete binary tree if each of its nodes has two children, except the last nodes. In other words, every non-terminal node of it must have both children except the last leaf nodes. So, at any level the maximum number of nodes is equal to 2. At level 0, there must be only one node and that is the root node. A at level 1 the maximum nodes must be 2. At level 3 the maximum nodes must be equal to 8. A complete binary tree can be obtained from Fig. 5.8.

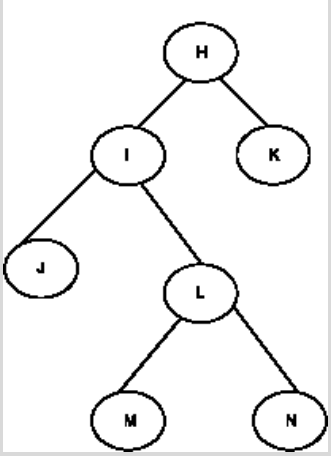
The advantage of this complete binary tree is that one can very easily locate the position of the parent and also left and right child nodes of a complete binary tree. Left child node and right child nodes are located at 2N and 2N+1. Similarly, the parent node of any node would be at floor (N/2).

Parent of D would be floor (5 / 2)=2, i.e. B is the parent and its left and right child are 2*2=4 and 2*2+1=5. 4 and 5 are the C and D child nodes in Fig. 5.8.

STRICTLY BINARY TREE

When every non-leaf node in binary tree is filled with left and right sub-trees, the tree is called strictly binary tree. It is shown in Fig. 5.9.

Figure 5.9. Strictly binary trees



In the strictly binary tree as shown in Fig. 5.9, L and I are non-terminal nodes with non-empty left and right sub trees.

EXTENDED BINARY TREE

When every node of a tree has either 0 or 2 children then such a tree is called extended binary tree or 2-tree. The nodes with two children are called internal nodes. The nodes without children are known as external nodes. At some places in order to identify internal nodes in figures 5.10 to 5.13 circles are used. To identify external nodes squares are used. The nodes in binary tree that have only one child can be extended with one more child. This extended binary tree can be used for implementing the algebraic equation because in the algebraic equation the left and right child nodes are operands and the parent of the child represents the operator.

Figure 5.10. Binary tree

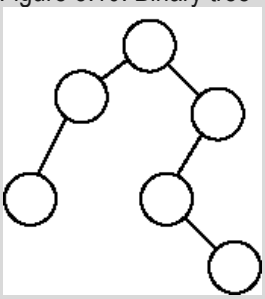


Figure 5.11. Extended 2-tree

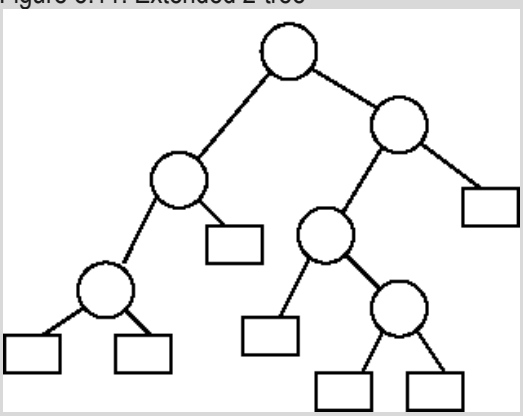


Figure 5.12. Binary trees

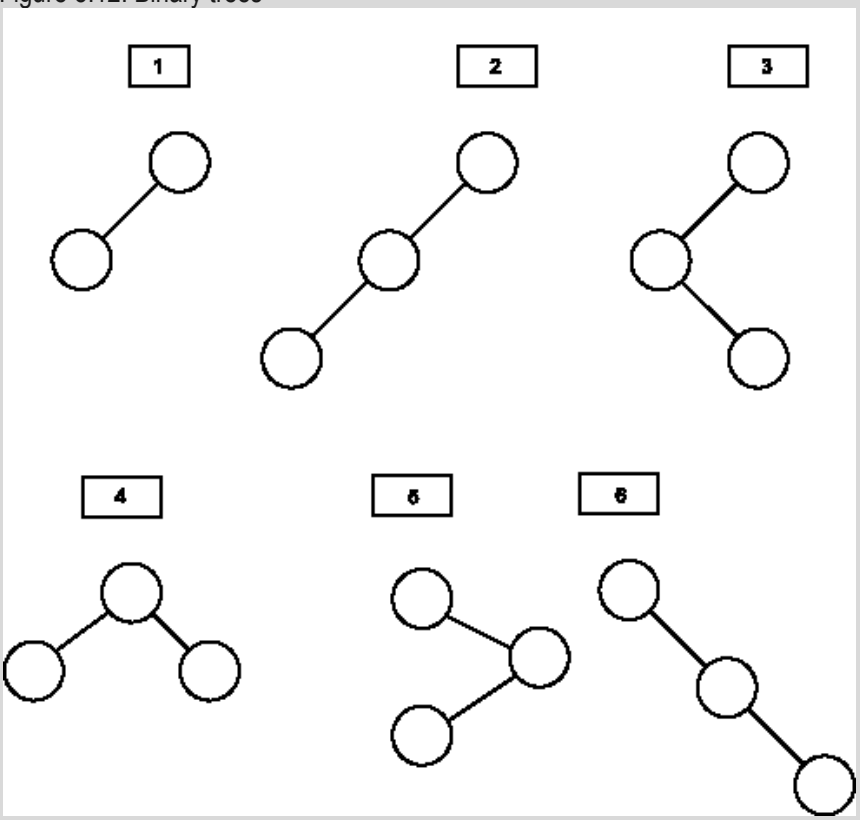
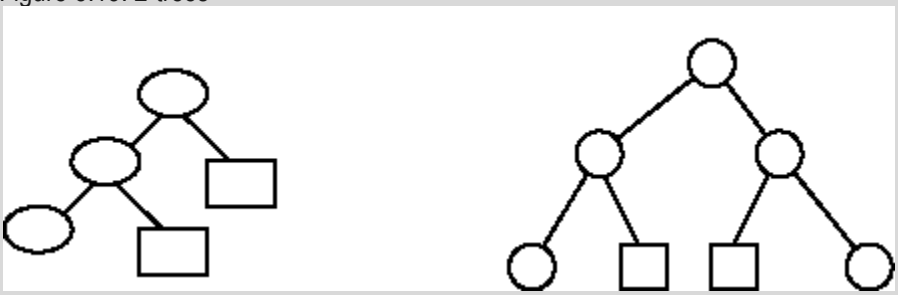


Figure 5.13. 2-trees



BINARY SEARCH TREE

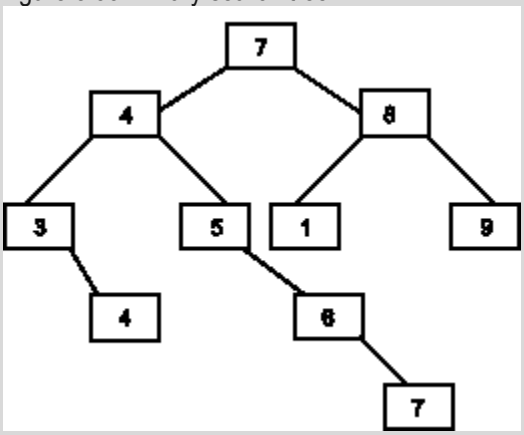
A binary search tree is also called as binary sorted tree. Binary search tree is either empty or each node N of tree satisfies the following property:

1. The key value in the left child is not more than the value of root.
2. The key value in the right child is more than or identical to the value of root.
3. All the sub-trees, i.e. left and right sub-trees follow the two rules mentioned above.

Binary search tree is shown in Fig. 5.33.

In Fig. 5.33 number 7 is the root node of the binary tree. There are two sub-trees to root 7. The left sub-tree is 4 and right sub-tree is 8. Here, the value of left sub-tree is lower than root and value of right sub-tree is higher than root node. This property can be observed at all levels in the tree.

Figure 5.33. Binary search tree



5.11.1 Searching an Element in Binary Search Tree

The item which is to be searched is compared with the root node. If it is less than the root node then the left child of left sub tree is compared otherwise right child is compared. The process would be continued till the item is found.

Insertion of an Element in Binary Search Tree

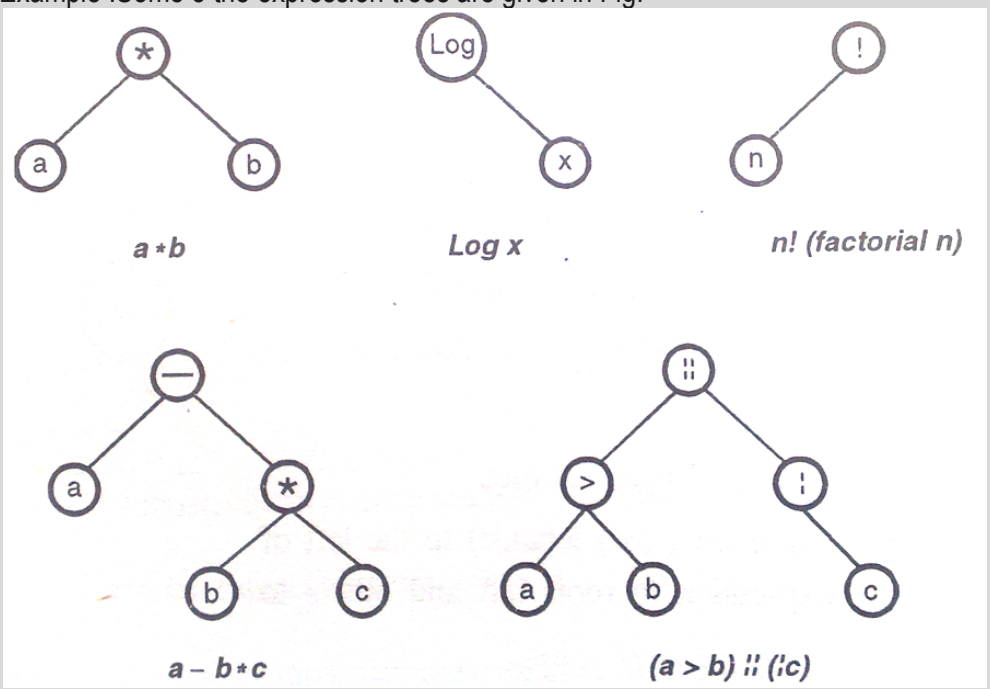
Insertion of an element in binary search tree needs to locate the parent node. The element to be inserted in the tree may be on the left sub-tree or right sub-tree. If the inserted number is lesser than the root node then left sub-tree is recursively called, otherwise right sub-tree is chosen for insertion.

Expression Tree

Definition :

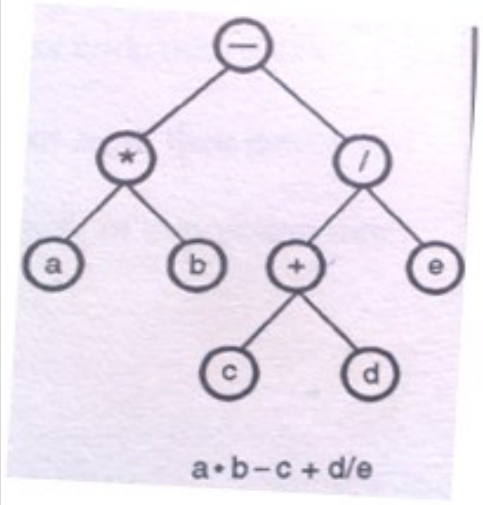
An expression tree is a tree built up from the simple operands as the leaves of binary tree and operators as the interior nodes.

Example :Some o the expression trees are given in Fig.



Let us see one more example, construct an expression tree for a * b – c + d/e.

This is shown in Fig



But one may ask the question is it the only possible tree for the given expression? Definitely, one may select any operator as root node, and which will result in different results. Thus to obtain the correct result, one has to consider the priority of operators. But, if we are given with infix and one of the other notations, then we can construct uniquely the expression tree.

Construction of an Expression tree from infix and prefix expressions :

Consider the inorder/infix sequence :

a – b * c + d

and preorder sequence,

* –ab + cd

To construct the equivalent binary tree follow the steps given below :

Step 1 : Read the preorder sequence, the first element becomes the root.

Step 2 : Now scan the infix sequence, till you get an element found in step 1. Place all the elements left of this element (of infix expression) to the left of root and others to right.

Step 3 : Repeat steps 1 and 2, till all the elements from infix sequence gets placed in tree.

Step 4 : Stop.

The conversion process is given below :

Infix : a – b*c + d

Prefix : *–ab + cd

↑

* becomes root :

Place all elements (from leftside) to the left of '*' from infix expression to roots left and others to the right.Now scan next element from prefix, which becomes root for subtree. Check the subexpression(subtree). Again separate left is right subexpression.

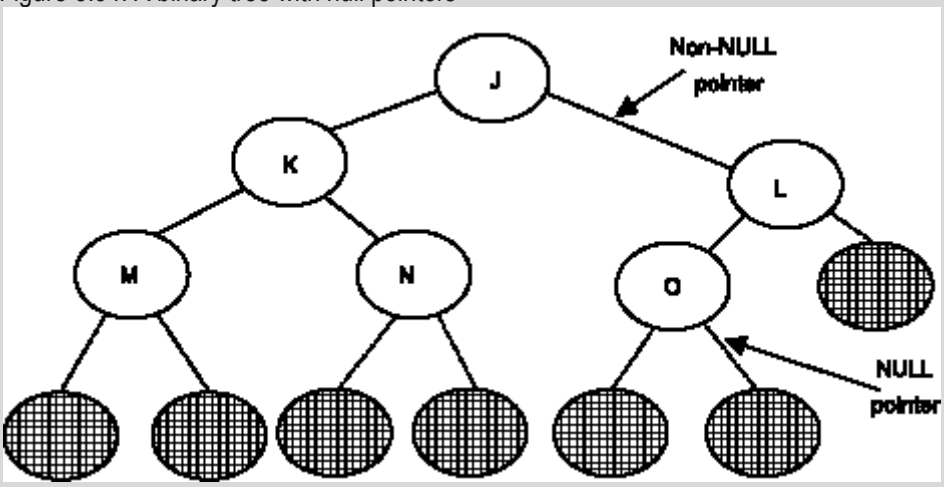
We now get, When we read next element from prefix we get (a, b) which have been already dealt with (they have become leaves). So no further processing required, i.e. a, b an operand.
Next we get '+' in prefix, which is not yet processed, (it has not become leaf nor a root of subtree). Hence scan the sub expression, and rearranging the elements we get, Thus it shows that we get the desired result.

Construction of an expression tree from a given postfix expression :
Let us consider the postfix expression
abc +*,
Now to construct an expression tree, we follow the steps given below :
Step 1 : Read the input character.
Step 2 : If it is an operand then push address of this node onto stack and goto step 4.
Step 3 : If the character is an operator, then pop twice, which gives the address of two operands. Create a new node for this operator and attach the operands to its left and right branch. Push the address of this node onto stack.
Step 4 : If all the characters from postfix expression are not read, then goto step 1.
Step 5 : Pop the content of stack which gives the root address of expression tree.
Step 6 : Stop.

THREADED BINARY TREE

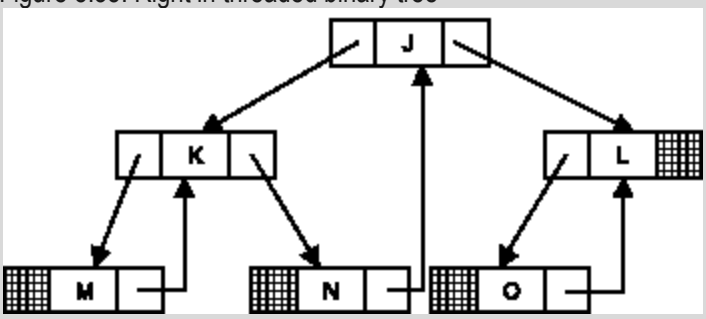
While studying the linked representation of a binary tree, it is observed that the number of nodes that have null values are more than the non-null pointers. The number of left and right leaf nodes has number of null pointer fields in such a representation. These null pointer fields are used to keep some other information for operations of binary tree. The null pointer fields are to be used for storing the address fields of higher nodes in tree, which is called thread. Threaded binary tree is the one in which we find these types of pointers from null pointer fields to higher nodes in a binary tree. Consider the following tree:

Figure 5.34. A binary tree with null pointers



In Fig. 5.34, in the binary tree there are 7 null pointers. These are shown with the dotted lines. There are total 12 node pointers out of which 5 are actual node pointers, i.e. non-null pointer (solid lines). For any binary tree having n nodes there will be (n+1) null pointers and 2n total pointers. All the null pointers can

be replaced with appropriate pointer value known as thread. The binary tree can be threaded according to appropriate traversal method. The null pointer can be replaced as follows:
Threaded binary tree can be traversed by any one of the three traversals, i.e. preorder, postorder and inorder. Further, in inorder threading there may be one-way inorder threading or two-way inorder threading. In one way inorder threading the right child of the node would point to the next node in the sequence of the inorder traversal. Such a threading tree is called right in threaded binary tree. Also, the left child of the node would point to the previous node in the sequence of inorder traversal. This type of tree is called as left in threaded binary tree. In case both the children of the nodes point to other nodes then such a tree is called as fully threaded binary tree.
Fig. 5.35 describes the working of right in threaded binary tree in which one can see that the right child of the node points to the node in the sequence of the inorder traversal method.
Figure 5.35. Right in threaded binary tree



The inorder traversal shown in the Fig. 5.35 will be as M-K-N-J-O-L. Two dangling pointers are shown to point a header node as shown below:

- rchild of M is made to point to K
- rchild of N is made to point to J
- rchild of O is made to point to L.

Similarly, the working of the left in binary threaded tree is illustrated in Fig. 5.36. In this case the left child of node points to the previous node in the sequence of inorder traversal.
As shown in Fig. 5.36, thread of N points to K. Here, K is the predecessor of N in inorder traversal. Hence, the pointer points to K. In this type of tree the pointers pointing to other nodes are as follows:

- lchild of N is made to point to K
- lchild of O is made to point to J.

Figure 5.36. Left in threaded binary tree

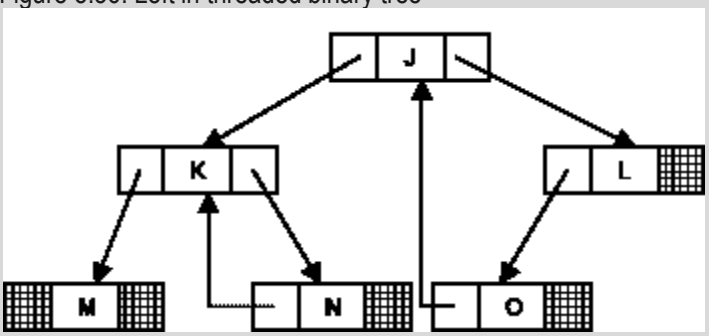
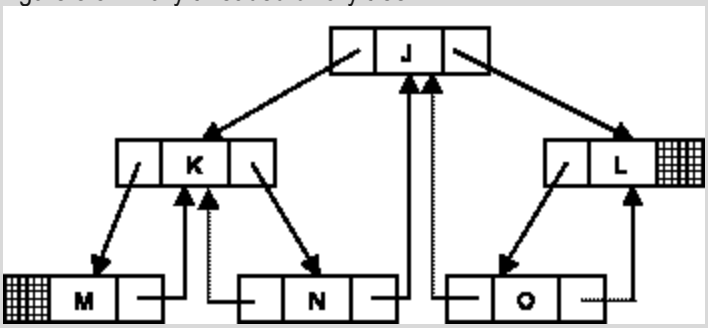


Fig. 5.37 illustrates the operation of fully threaded binary tree. Right and left children are used for pointing to the nodes in inorder traversal method.

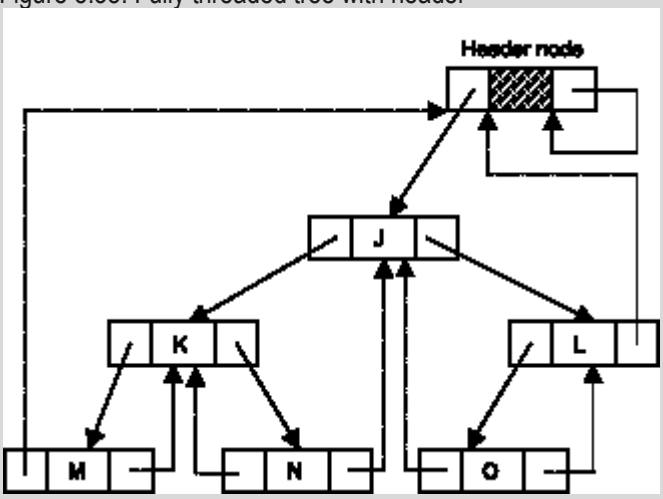
- rchild of M is made to point to K
- lchild of N is made to point to K
- rchild of N is made to point to J
- lchild of O is made to point to J
- rchild of O is made to point to L.

Figure 5.37. Fully threaded binary tree



Fully threaded binary tree with header is described in Fig. 5.38.
rchild of M is made to point to K
lchild of M is made to point to Header
lchild of N is made to point to K
rchild of N is made to point to J
lchild of O is made to point to J
rchild of O is made to point to L
rchild of L is made to point to Header.

Figure 5.38. Fully threaded tree with header



The working of the fully threaded binary tree is illustrated in Fig. 5.38. In this case the left child of node points to the previous node in the sequence of inorder traversal and right child of the node points to the successor node in the inorder traversal of the node. In the previous two methods left and right pointers of the first and last node in the inorder list are NULL. But in this method the left pointer of the first node points to the header node and the right pointer of the last node points to the header node. The header node's right pointer points to itself, and the left pointer points to the root node of the tree. The use of the header is to store the starting address of the tree. In the fully threaded binary thread each and every pointer points to the other nodes. In this tree we do not find any NULL pointers.

In the Fig. 5.38 the first node in the inorder is M and its left pointer points to the left pointer of the header node. Similarly, the last node in the inorder is L and its right pointer points to the left pointer of the header.

In memory representation of threaded binary tree, it is very important to consider the difference between thread and normal pointer. The threaded binary tree node is represented in Fig. 5.39.

Figure 5.39. Representation of the node

Each node of any binary tree stores the three fields. The left field stores the left thread value and the right field stores the right thread value. The middle field contains the actual value of the node, i.e. data.

AVL TREE

HEIGHT-BALANCED TREE

The efficiency of searching process in binary tree depends upon the method in which the data is organised. A binary tree is said to be completely balanced binary tree if all its leaves present at nodes of level h or h-1 and all its nodes at level less than h-1 contain two children.

Figure 5.40. Full binary tree

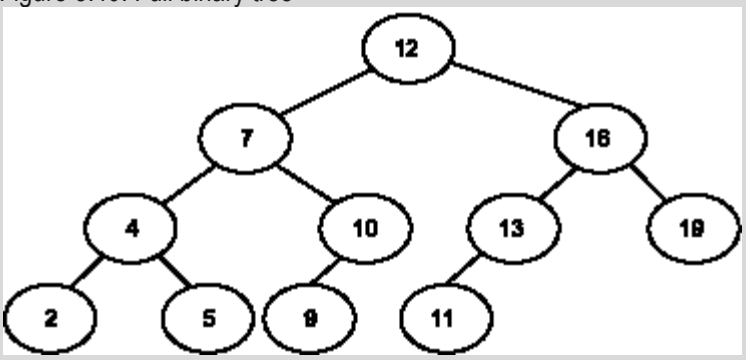
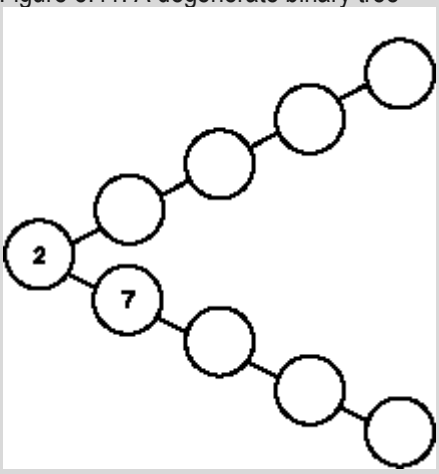


Figure 5.41. A degenerate binary tree

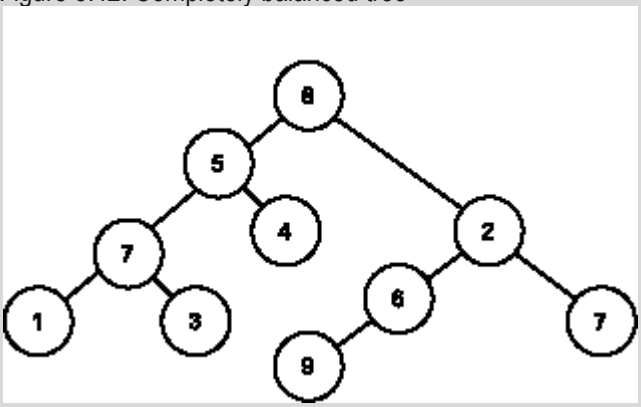


A tree is height balanced tree when each node in the left sub-tree varies from right sub-tree by not more than one.

A nearly height balanced tree is called an AVL. This form of tree is studied and defined by Russian mathematician G.M. Adel'son-Velskii and E.M. Landis in 1962.

We can conclude number of nodes might be present in a balanced tree having height h. At level one there is only one node, i.e. root. In every successive level the number of nodes increases i.e. 2 nodes at level 2, 4 nodes at level 3, and so on. There will be 2^{1-1} nodes at level 1. Thus, we can calculate total number of nodes from level 1 through level h-1 will be $1+2+2^2+2^3+.....+2^{h-2} = 2^{h-1} -1$. The number of nodes at level h may be from 1 to 2h-1 nodes. The total number of nodes (n) of tree range from 2h-1 to 2h-1 or (2h-1-1+1).

Figure 5.42. Completely balanced tree



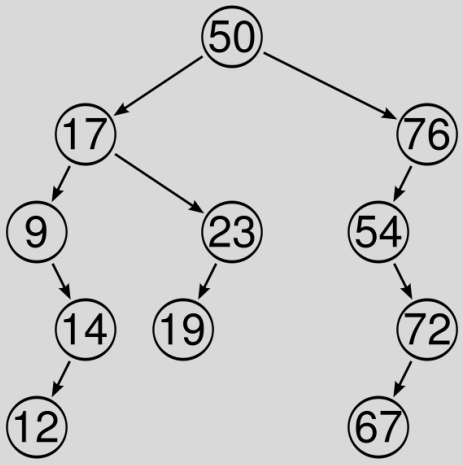
An AVL tree should satisfy the following rules:

1. A node is known as left heavy if the longest path in its left child (left sub-tree) is longer than the longest path in right sub-tree.
2. A node is known as right heavy if the longest path in its right sub-tree is longer than left sub-tree.
3. A node is known as balanced if the longest path in left and right sub-trees are identical.

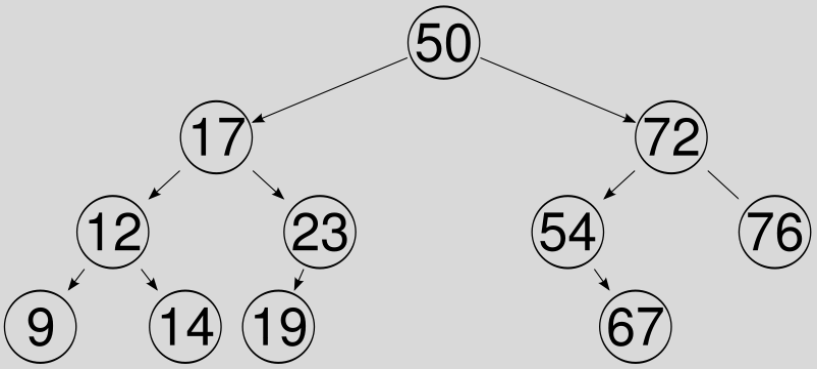
AVL tree

In computer science, an AVL tree is a self-balancing binary search tree, and it is the first such data structure to be invented.[1] In an AVL tree, the heights of the two child subtrees of any node differ by at most one; therefore, it is also said to be height-balanced. Lookup, insertion, and deletion all take O(log n) time in both the average and worst cases, where n is the number of nodes in the tree prior to the operation. Insertions and deletions may require the tree to be rebalanced by one or more tree rotations.

he balance factor of a node is the height of its right subtree minus the height of its left subtree and a node with balance factor 1, 0, or -1 is considered balanced. A node with any other balance factor is considered unbalanced and requires rebalancing the tree. The balance factor is either stored directly at each node or computed from the heights of the subtrees.



Unbalanced tree



balanced

Operations

The basic operations of an AVL tree generally involve carrying out the same actions as would be carried out on an unbalanced binary search tree, but preceded or followed by one or more operations called tree rotations, which help to restore the height balance of the subtrees.

Insertion

Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

If the balance factor becomes -1, 0, or 1 then the tree is still in AVL form, and no rotations are necessary.

If the balance factor becomes 2 or -2 then the tree rooted at this node is unbalanced, and a tree rotation is needed. At most a single or double rotation will be needed to balance the tree.

There are basically four cases which need to be accounted for, of which two are symmetric to the other two. For simplicity, the root of the unbalanced subtree will be called P, the right child of that node will be called R, and the left child will be called L. If the balance factor of P is 2, it means that the right subtree outweighs the left subtree of the given node, and the balance factor of the right child (R) must then be checked. If the balance factor of R is 1, it means the insertion occurred on the (external) right side of that node and a left rotation is needed (tree rotation) with P as the root. If the balance factor of R is -1, this means the insertion happened on the (internal) left side of that node. This requires a double rotation. The first rotation is a right rotation with R as the root. The second is a left rotation with P as the root.

The other two cases are identical to the previous two, but with the original balance factor of -2 and the left subtree outweighing the right subtree.

Only the nodes traversed from the insertion point to the root of the tree need be checked, and rotations are a constant time operation, and because the height is limited to O(log(n)), the execution time for an insertion is O(log(n)).

Deletion

If the node is a leaf, remove it. If the node is not a leaf, replace it with either the largest in its left subtree (inorder predecessor) or the smallest in its right subtree (inorder successor), and remove that node. The node that was found as replacement has at most one subtree. After deletion retrace

the path back up the tree (parent of the replacement) to the root, adjusting the balance factors as needed.

The retracing can stop if the balance factor becomes -1 or 1 indicating that the height of that subtree has remained unchanged. If the balance factor becomes 0 then the height of the subtree has decreased by one and the retracing needs to continue. If the balance factor becomes -2 or 2 then the subtree is unbalanced and needs to be rotated to fix it. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one. This is in contrast to an insertion where a rotation resulting in a balance factor of 0 indicated that the subtree's height has remained unchanged.

The time required is O(log(n)) for lookup plus maximum O(log(n)) rotations on the way back to the root; so the operation can be completed in O(log n) time.

B-TREE (BALANCED MULTI-WAY TREE)

Binary search tree is, in general called multi-way search tree. The integer m is called the order of the tree. Each node should have maximum m children. If k ≤ m, where m is number of children, then node has accurately k-1 keys which divides all the keys into k number of sets. In case some sets are empty, the children are also empty.

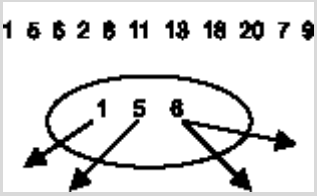
The B-tree is also known as the balanced sort tree. The B-tree is used in external sorting. The B-tree is not a binary tree. While implementing B-tree following conditions are followed:

1. The height of the tree must be minimum.
2. There should be no empty sub-trees after the leaves of the tree.
3. The leaves of the tree should be at the same level.
4. All nodes excepting the leaves should have at least few children.

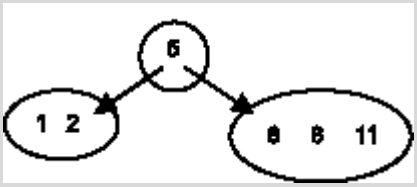
5.14.1 B-Tree Insertion

In B-tree insertion at first search is made where the new element is to be placed. If the node is suitable to the given element then insertion is straightforward. The element is inserted by using an appropriate pointer in such an order that number of pointers will be one more than the number of records. In case, the node overflows due to upper bound of node, splitting is mandatory. The node is divided into three parts. The middle part is passed upward. It will be inserted into the parent. Partition may spread the tree. This is because the parent into which element is to be inserted spilts into its child nodes. If the root is needed to be split, a new root is created with two children. The tree grows by one level.

Example: Consider the following B-tree of degree 4. It can be balanced in four ways. Here, each node holds elements. It also has four branches. Suppose, it has the following values:

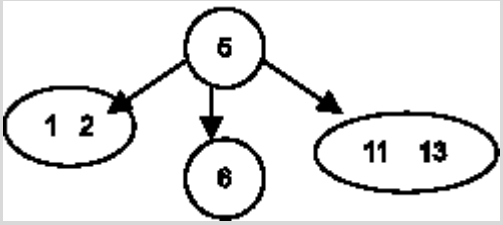


The value 1 is put in a new node. This node can also hold next two values.

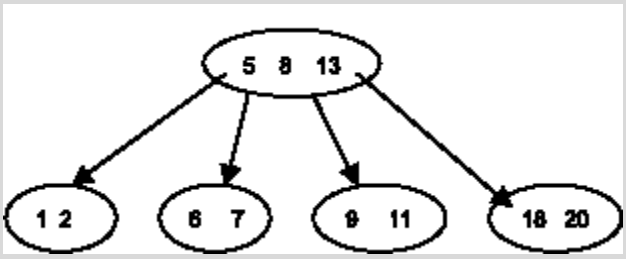


When value 2 (4th value) is put, the node is split at 5 into leaf nodes. Here, 5 is parent. The element 8 is added in leaf node. The search for its accurate position is done in the node having value 6. The element 8 also is present in the same node.

The element 13 is to be inserted. However, the right leaf node, in which 1 to 3 values have appropriate plane, is occupied. Hence, the node splits at median 8 and this moves it up to the parent.



By following the above procedure the remaining nodes can be included. The final figure would be as follows:



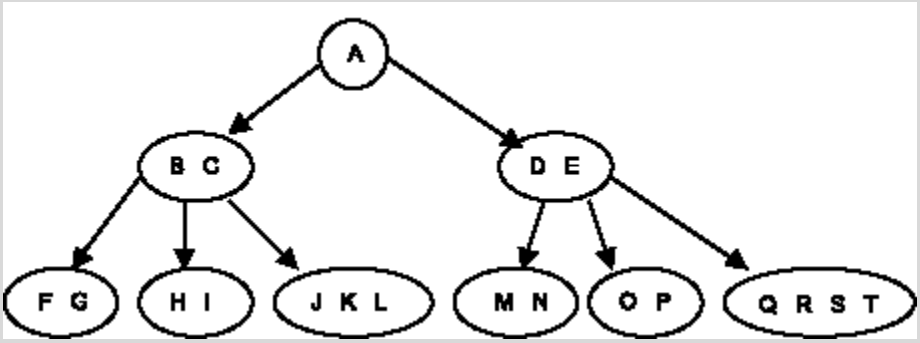
B-Tree Deletion

In B-Tree deletion the element which is to be deleted is searched. In case the element is terminal node, the deletion process is straightforward. The element with a suitable pointer is deleted.

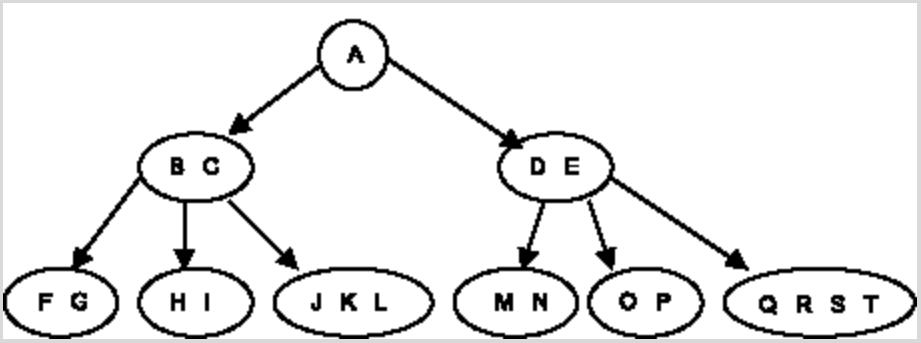
If the element fixed for deletion is not a terminal node, it is replaced by its successor element. Actually, a copy of successor is taken. The successor is an element with higher value. The successor of any node which is not at lowest level, be a terminal node. Thus, deletion is nothing but removing of a particular element or record from a terminal node. While deleting the record the new node size is more than minimum, i.e. the deletion is complete. If the node size is less than minimum, an underflow happens.

Rearrangement is done if either of adjacent siblings has more than the minimum elements (records). For rearrangement, the contents of the node (only those nodes having less than minimum records) along with sorting out records from parent node are gathered. The central record is written back to the parent and left and right halves are written back to two siblings.

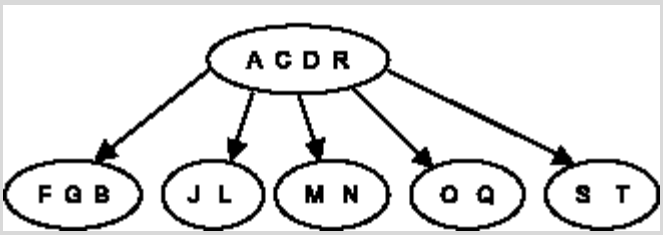
Concatenation is applied if the node with less than minimum number of records has no adjacent sibling. The node is combined with its neighbouring sibling and element is moved from its parent.



1. Deleting K is straightforward because it is a leaf node.
2. Deleting E is not simple. Hence, its successor is moved up. E is moved down and deleted.



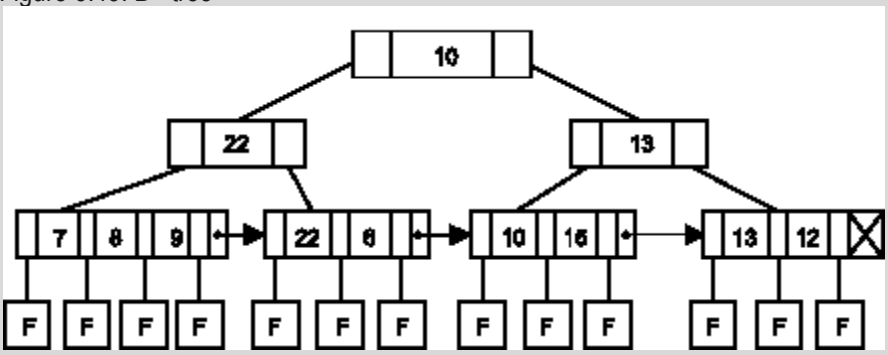
3. To delete P, the node has less than minimum numbers of keys. The sibling is carried. R moves up and Q moves down.
4. Deleting H, again node has less than minimum keys than required. The parent is left with only one key. Here, sibling cannot be applied. Hence, A, C, D and R form a new node.



B+ TREE

A B^+ tree can be obtained with slight modification of indexing in B tree. A B^+ tree stores key values repeatedly. Fig. 5.43 shows B^+ tree indexing.

Figure 5.43. B+ tree



By observing Fig. 5.43, we will come to know that the key values 10, 22 and 13 are stored repeatedly in the last level (terminal nodes). In these leaf nodes a link is maintained so that the traversing can be done from the leaf node at the extreme left to the leaf node at the extreme right. B tree and B^+ tree are same except the above differences.

Every leaf node of B^+ tree has two parts:

1. Index part It is the interior node stored repeatedly.
2. Sequence set It is a set of leaf nodes.

The interior nodes or keys can be accessed directly or sequentially. B^+ tree is useful data structure in indexed sequential file organization.

Huffman Tree/ Encoding :

Suppose that we have an algorithm of n symbols and a long message consisting of symbols from this alphabet. We wish to encode the message as a long bit string (a bit is either 0 or 1) by assigning a bit string code to each symbol of the alphabet and concatenating the individual codes of the symbols making up the message to produce an encoding for the message.

For example, suppose that the message is ABACCDA. Each of the letters B and D appears only once in the message, whereas the letter A appears three times. If a code is chosen so that the letter A is assigned a shorted bit string than the letters B and D, the length of the encoded message would be small. This is because the short code (representing the letter A) would appear more frequently than the long code. Indeed the code can be represented as follows :

Symbol	Code
A	0
B	110
C	10
D	111

Using this code, the message ABACCDA is encoded as 0110010101110, which requires only 13 bits. In very long messages containing symbols that appear very infrequently the savings are substantial. Ordinarily, codes are not constructed on the basis of frequency of characters within a single message alone, but on the basis of their frequency within a whole set of messages.

In our example decoding proceeds by scanning a bit from left to right. If a 0 is encountered as the first bit, the symbol is an A; otherwise it is a B,C, or D, and the next bit is examined. If the second bit is 0, the symbol is C; otherwise it must be a B or a D, and the third bit must be examined. If the third bit is 0, the symbol is a B; if it is a 1, the symbol is a D. As soon as the first symbol has been identified the process is repeated starting at the next bit to find the second symbol.

This suggests a method for developing an optimal encoding scheme, given the frequency of occurrence of each symbol in a message.

Construction of Huffman Tree :

Find the two symbols that appear least frequently. In our example, these are B and D. The last bit of their codes differentiates one from the other: 0 for B and 1 for D. combine these two symbols into the single symbol BD, whose code represents the knowledge that a symbol is either a B or d. The frequency of occurrence of this new symbol is the sum of the frequencies of its two constituent symbols. Thus the frequency of BD is 2.

There are now three symbols : A(frequency 3), C(frequency 2) and BD (frequency 2). Again choose the two symbols with smallest frequency : C and BD. The last bit of there codes again differentiates one from the other: 0 for C and 1 for BD. The two symbols are then combined into the single symbol CBD with frequency 4. there are now only two symbols remaining. A and CBD. These are combined into the single

symbol ACBD. The last bits of the codes for A and CBD differentiate one from the other: 0 for A and 1 for CBD.
The symbol ACBD contains the entire alphabet; it is assigned the null bit string of length 0 as its code. At the start of the decoding, before any bits have been examined it is certain that any symbol is contained in ACBD. The two symbols that make up ACBD (A and CBD) are assigned the code 0 and 1, respectively. If a 0 is encounted the encoded symbol is in A if a 1 is encounted, it is a C or B or D. Similarly, the two symbols that constitute CBD (C abd BD) are assigned the codes 10 and 11, respectively.
The first bit indicates that symbol is one of the constituents of CBD and the second bit indicates whether it is a C or BD. The symbols that make up BD(B and D) are then assigned the codes 110 and 111. By this process, symbols that appear frequently in the message are assigned shorter codes than symbols that appear infrequently.

The action of combining two symbols into one suggests the use of a binary tree. Each node of the tree represents a symbol and each leaf represents a symbol of the original alphabet. Fig shows the binary tree constructed using the previous example. Each node contains a symbol and its frequency. Such trees are called Huffman Trees after the discoverer of this encoding method. Huffman tree is strictly binary.

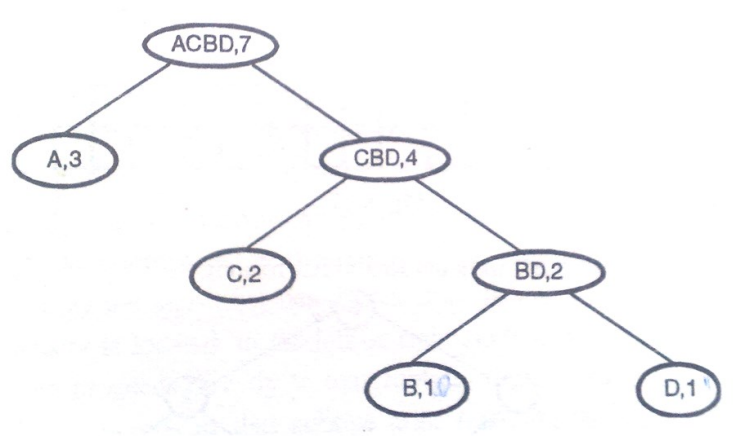


Fig. Huffman tree

Once the Huffman tree is constructed, the code of any symbol in the alphabet can be constructed by starting at the leaf representing that symbol and climbing up to the root. The code is initialized to null. Each time that a left branch is climbed, 0 is appended to the beginning of the code; each time that a right branch a climbed, 1 is appended to the beginning of the code.

BINARY TREE REPRESENTATION

Binary tree can be represented by two ways:

1. Array representation
2. Linked representation.

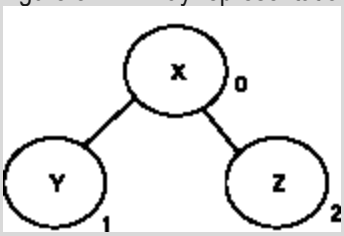
1. Array Representation of Binary Tree

In any type of data structure array representation plays an important role. The nodes of trees can be stored in an array. The nodes can be accessed in sequence one after another. In array, element counting starts from zero to (n-1) where n is the maximum number of nodes. In other words, the numbering of binary tree nodes will start from 0 rather than 1.
Assume an integer array. Its declaration is as follows:

```
int array tree[n];
```

The root node of the tree always starts at index zero. Then, successive memory locations are used for storing left and right child nodes. Consider the following Fig. 5.14,

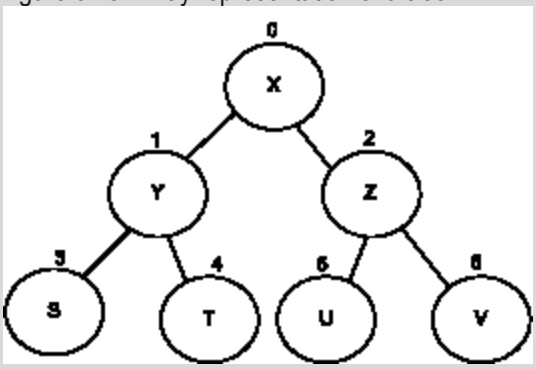
Figure 5.14. Array representation of tree



TREE[0]	*X*
TREE[1]	*Y*
TREE[2]	*Z*

In the above figure, X is the root and Y and Z are children. X is father of child Y and Z. Consider the following Fig. 5.15 with one more level.

Figure 5.15. Array representation of a tree



The array representation with one more level would be as follows:	
0	X
1	Y
2	Z
3	S
4	T
5	U
6	v

It is very easy in this representation to identify the father, left and right child of an arbitrary node. For any node n, 0 ≤ n ≤ (MAXSIZE -1), the following can be used to identify the father and child nodes.

Father (n)

The location of the father node can be identified in a given tree by using the ((n-1)/2) where n is the index of child node, provided that n! =0 (not equal to zero). In case if n=0, the said node is root node. It has no father node. Consider the node 3 in Fig. 5.15 i.e. S. The father of node S is Y and the index of Y is 1. By substituting these values in the equation we identify the index of the father node.

Floor ((3-1)/2)) i.e. (2/2)=1

Lchild(n)

The left child of a node (n) is at position (2n+1).

1. Lchild(X) =lchild(0)
   =2 * 0+1
   =1
   The node with index 1 is Y.

2. lchild (Z) = lchild(2)
   = 2*2+1 = 5
   The node with index 5 is U.

rchild (n)

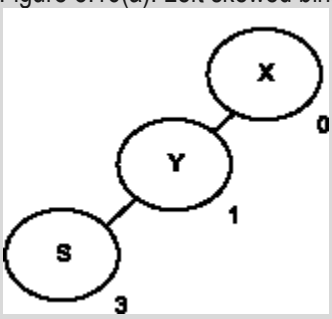
The right child of node n can be recognized by (2n+2)

1. rchild (X)=rchild (0)
   = 2 * 0+2
   = 2
   The node with index 2 is Z.
2. rchild (Y): rchild(1)
   = 2 *1+2=4
   The node with index 4 is T.

Siblings

In case the left child at index n is known then its right sibling is at (n+1). Likewise, if the right child at index n is known then is left sibling is at (n-1). The array representation is perfect for complete binary tree. However, this is not appropriate for other tree types and if we use array for their representation, it results in inefficient use of memory. Consider the binary trees shown in Figs 5.16(a) and 5.16(b).

Figure 5.16(a). Left skewed binary tree



The above is skewed binary tree. Here, every left sub-tree again represents left sub-tree. Such type of binary tree is called left skewed binary tree. Similarly, right skewed binary tree is also present [See Figs 5.16(a) and 5.16(b)]. Fig. 5.17 shows array representation of left and right skewed trees.

Figure 5.16(b). Right skewed binary tree

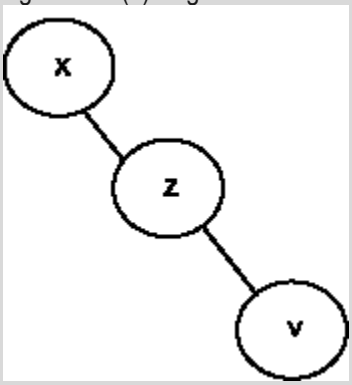
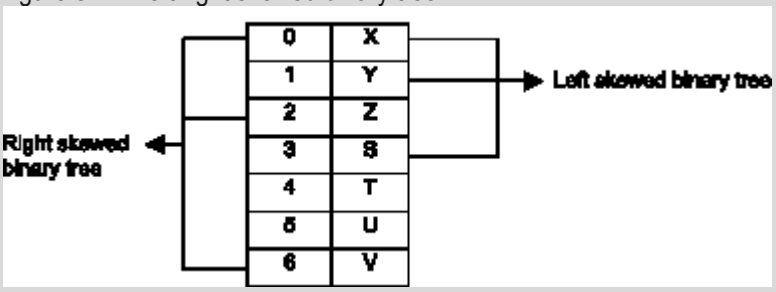


Figure 5.17. Left-right skewed binary tree

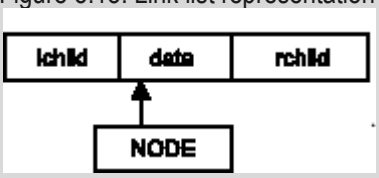


2.Linked Representation of Binary Tree

Another way of representation of binary tree is linked list, which is known to be more memory efficient than array representation. The fundamental component of binary tree is node. We know that the node consists of three fields, which is shown in Fig. 5.18.

- Data
- Left child
- Right child.

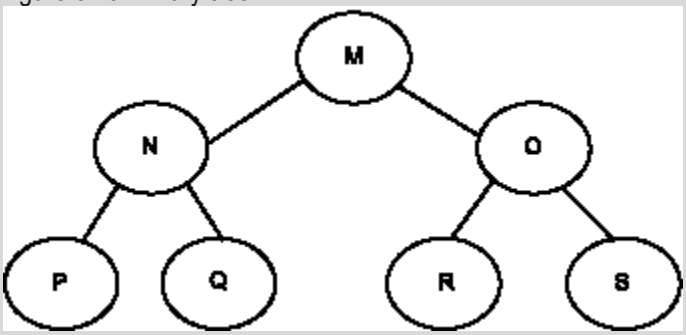
Figure 5.18. Link list representation of binary tree



We have already learnt about nodes in Chapter 6. The data field stores the given values. The lchild field is link field and holds the address of its left node. Similarly, the rchild holds the address of right node. The node can be logically represented as follows. Figs 5.19 and 5.20 show the linked list representation of binary tree.

```
struct node
{
  int data;
  struct node *rchild;
  struct node *lchild;
};
```

Figure 5.19. Binary tree



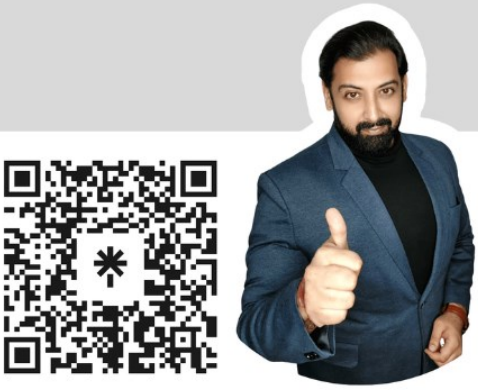
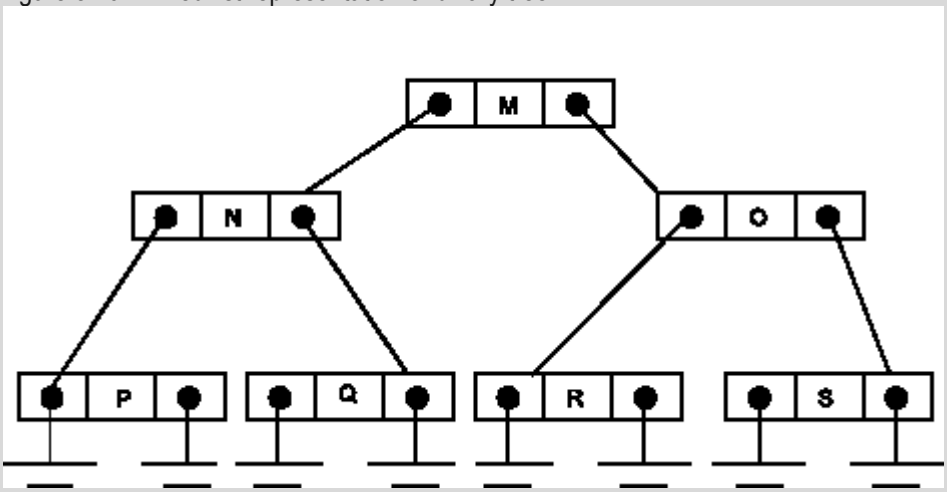


Figure 5.20. Linked list representation of binary tree



Binary tree has one root node and few non-terminal and terminal nodes. The terminal nodes are called leafs. The non-terminal nodes have their left and right child nodes. However, the terminal nodes are without child. While implementing this, fields of lchild and rchid are kept NULL. The non-terminal nodes are known as internal nodes and terminal nodes are known as external nodes.

OPERATION ON BINARY TREE

Following fundamental operations are performed on binary tree:

Create

This operation creates an empty binary tree.

Make

This operation creates a new binary tree. The data field of this node holds some value.

Empty

When binary tree is empty, this function will return true else it will return false.

Lchild

A pointer is returned to left child of the node. When the node is without left child, a NULL pointer is returned.

Rchild

A pointer is returned to right child and if the node is without right child a NULL pointer is returned.

Father

A pointer to father of the node is returned or else the NULL pointer is returned.

Sibling (Brother)

A pointer to brother of the node is returned or else NULL pointer is returned.

Data

Value stored is returned.

In addition to above mentioned operations, following operations can also be performed on binary tree:

1. Tree traversal
2. Insertion of nodes
3. Deletion of node
4. Searching for a given node
5. Copying the binary tree.

TRAVERSAL OF A BINARY TREE

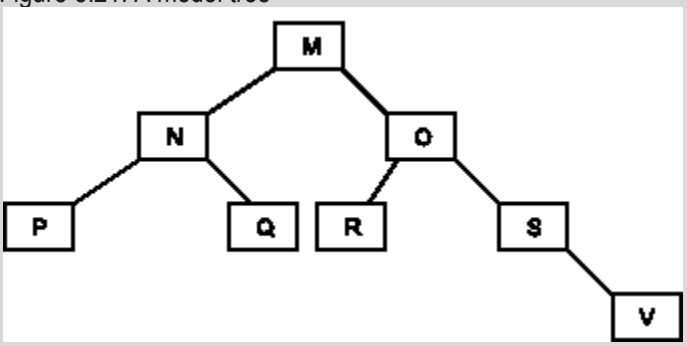
Three parameters are needed for formation of binary tree. They are node, left and right sub-trees. Traversing is one of the most important operations done on binary tree and frequently this operation is carried on data structures. Traversal means passing through every node of the tree one by one. Every

node is traversed only once. Assume, root is indicated by O, left sub-tree as L and right sub-tree as R. Then, the following traversal combinations are possible:

1. ORL - ROOT - RIGHT-LEFT
2. OLR - ROOT - LEFT-RIGHT
3. LOR - LEFT - ROOT- RIGHT
4. LRO - LEFT - RIGHT- ROOT
5. ROL- RIGHT - ROOT - LEFT
6. RLO- RIGHT - LEFT-ROOT

Out of six methods only three are standard and are discussed in this chapter. In traversing always right sub-tree is traversed after left sub-tree. Hence, the OLR is preorder, LOR is inorder and LRO is postorder.

Figure 5.21. A model tree

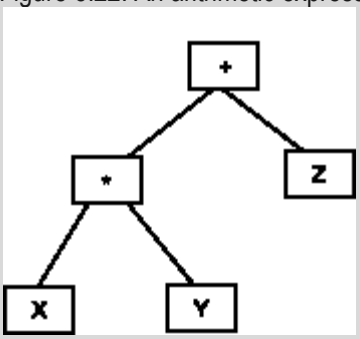


The inorder representation of above tree is P-N-Q-M-R-O-S-V.

Traversing is a common operation on binary tree. The binary tree can be used to represent an arithmetic expression. Here, divide and conquer technique is used to convert an expression into a binary tree. The procedure to implement it is as follows.

The expression for which the following tree has been drawn is (X*Y)+Z. Fig. 5.22 represents the expression.

Figure 5.22. An arithmetic expression in binary tree form



Using the following three methods, the traversing operation can be performed. They are:

1. Preorder traversal
2. Inorder traversal
3. Postorder traversal.

All the above three types of traversing methods are explained below.

5.9.1 Inorder Traversal

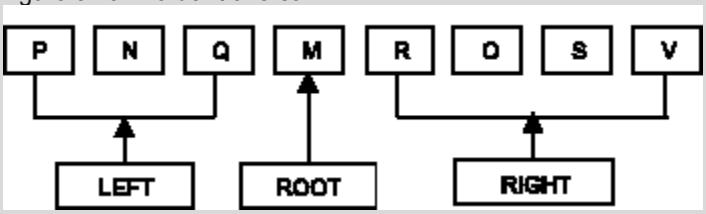
The functioning of inorder traversal of a non-empty binary tree is as follows:

1. Firstly, traverse the left sub-tree in inorder.
2. Next, visit the root node.
3. At last, traverse the right sub-tree in inorder.

In the inorder traversal firstly the left sub-tree is traversed recursively in inorder. Then the root node is traversed. After visiting the root node, the right sub-tree is traversed recursively in inorder. Fig. 5.21

illustrates the binary tree with inorder traversal. The inorder traversing for the tree is P-N-Q-M-R-O-S-V. It can be illustrated as per Fig. 5.23.

Figure 5.23. Inorder traversal



The left part constitutes P, N, and Q as the left sub-tree of root and R, O, S, and V are the right sub-tree.

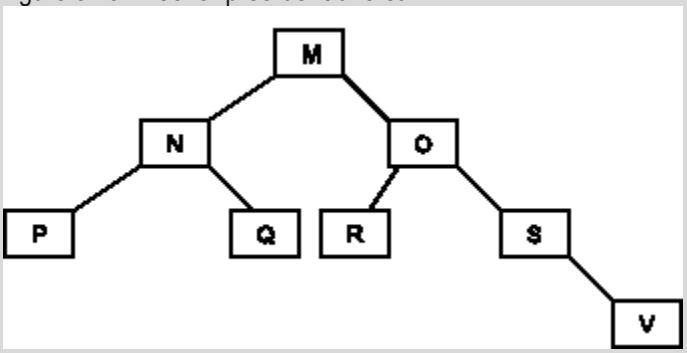
Preorder Traversal

The node is visited before the sub-trees. The following is the procedure for preorder traversal of non-empty binary tree.

1. Firstly, visit the root node (N).
2. Then, traverse the left sub-tree in preorder (L).
3. At last, traverse the right sub-tree in preorder R.

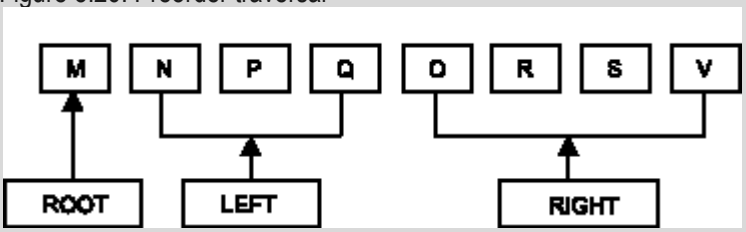
The preorder is recursive in operation. In this type, first root node is visited and later its left and right sub-trees. Consider the following Fig. 5.25 for preorder traversing.

Figure 5.25. Tree for preorder traversal



The preorder traversing for Fig. 5.25 is M, N, P, Q, O, R, S, and V. This can also be shown in Fig. 5.26. In this traversing the root comes first and the left sub-tree and right sub-tree at last.

Figure 5.26. Preorder traversal



In the preorder the left sub-tree appears as N, P, and Q and right sub-tree appears as O,R,S, and V.

Postorder Traversal

In the postorder traversal the traversing is done firstly left and right sub-trees before visiting the root. The postorder traversal of non-empty binary tree is to be implemented as follows:

1. Firstly, traverse the left sub-tree in postorder style.
2. Then, traverse the right sub tree in postorder.
3. At last, visit the root node (N).

In this type, the left and right sub-trees are processed recursively. The left sub-tree is traversed first in postorder. After this, the right sub-tree is traversed in post order. At last, the data of the root node is shown. Fig. 5.28 shows the postorder traversal of a binary tree.

The postorder traversing for the above Fig. 5.28 is P, Q, N, R, V, S, O, and M. This can also be shown as in Fig. 5.29. In this traversing the left sub-tree is traversed first, then right sub-tree and at last root. In the postorder the left sub-tree is P, Q and N and the right sub-tree is R, V, S and O.

Figure 5.28. Tree for post order

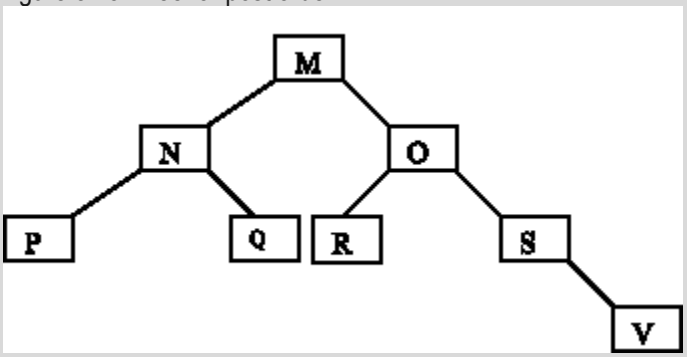
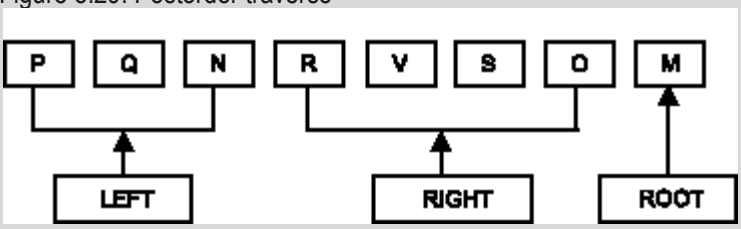


Figure 5.29. Postorder traverse



CHAPTER 6
GRAPH

INTRODUCTION

Graphs are frequently used in every walk of life. Every day we come across various kinds of graphs appearing in newspapers or television. The countries in a globe are seen in a map. A map depends on the geographic location of the places or cities. As such, a map is a well-known example of a graph. In the map, various connections are shown between the cities. The cities are connected via roads, rail or aerial network. How to reach a place is indicated by means of a graph. Using the various types of the links the maps can be shown.

Fig. 6.1 illustrates a graph that contains the cities of India connected by means of road. Assume that the graph is the interconnection of cities by roads. As per the graph, Mumbai, Hyderabad and Kolkata are directly connected to all the other cities by road. Delhi is directly connected to Mumbai, Hyderabad, and Kolkata. Delhi is connected to Chennai via Hyderabad.

Generally, we provide the address of our office/residence to a stranger who is not aware of our address and location of city. At this juncture we use the graph for the easiest representation of our residential location. Fig. 6.2 shows the location of place by graph. By using the graph any stranger can easily find location. For example, a pilgrim wishes to reach the Gurudwara in Nanded. The path is shown in the figure for reaching the Gurudwara. The devotee has to reach the destination via Shivaji statue, Mahatma

Gandhi statue and then Gurudwara as per the graph. Once a map is provided, any stranger can reach any destination by using appropriate conveyance.

Figure 6.1. Map representation of connection of cities

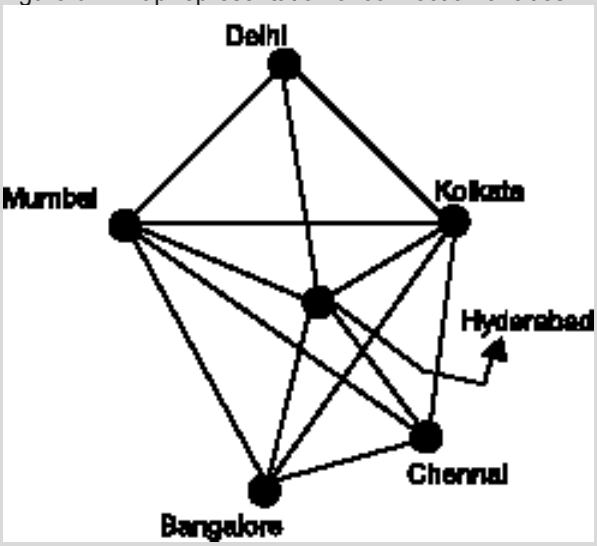
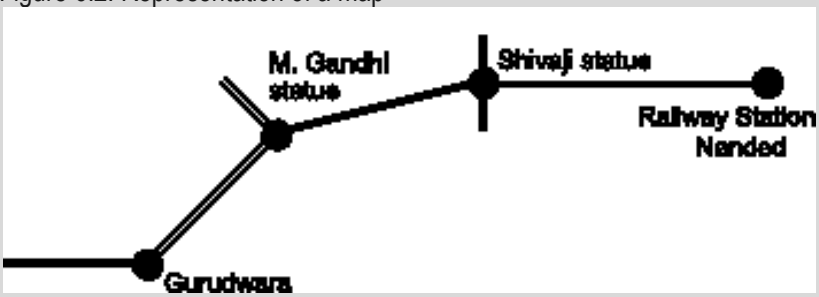


Figure 6.2. Representation of a map



In Fig. 6.2, four places are connected by road links. In the graph the road links are called as edges and the places are called as vertices. The graph is a collection of the vertices and the edges, hence a map is treated as graph. The following section describes the graph and relevant theories.

Like tree, graphs are nonlinear data structures. Tree nodes are arranged hierarchically from top to bottom like a parent is placed at the top and child as successor at the lower level. Tree has some specific structure whereas graph does not have a specific structure. It varies from application to application.

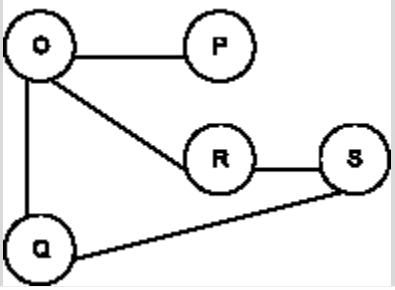
GRAPH

A graph is set of nodes and arcs. The nodes are also termed as vertices and arcs are termed as edges. The set of nodes as per the Fig. 6.3 is denoted as {O, P, R, S, Q}. The set of arcs in the following graph are {(O,P), (O,R), (O,Q), (R,S), (Q,S)}. Graph can be represented as,

G= (V,E) and V(G)= (O, Q, P, R, S) or group of vertices.

Similarly, E(G)= ((O,P), (O,R), (O,Q),(R,S),(Q,S)) or group of edges.

Figure 6.3. Graph



A graph is linked if there is pathway between any two nodes of the graph, such a graph is called connected graph or else, it is non-connected graph. Fig. 6.4 and 6.5 are connected and non-connected graphs, respectively. In both the figures four nodes are depicted. In the latter all the nodes are not connected by links whereas in the former case all the nodes are joined by paths or links.

Figure 6.4. Connected graph

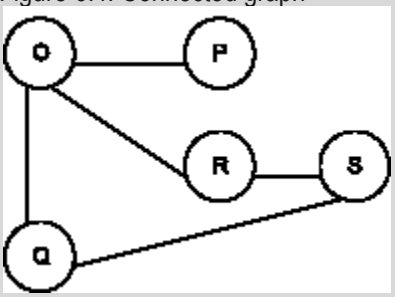
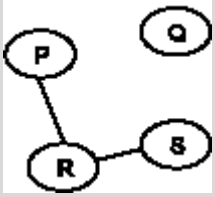


Figure 6.5. Non-connected graph

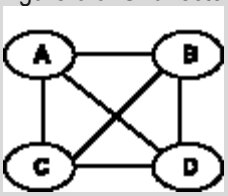


Undirected Graph

A graph containing unordered pair of nodes is termed as undirected graph.

The vertices are {A, B, C, D} and edges are {(A, B), (A, D), (A, C), (B, D), (B, C), (D, C)}. The graph has four nodes and six edges. This type of graph is known as completely connected network, in which every node is having out going path to all nodes in the network. For a complete network the number of links =N (N-1)/2, where N is the number of vertices or nodes. In the Fig. 6.6 N is 4. By substituting its value the number of edges obtained will be equal to 6.

Figure 6.6. Undirected graph

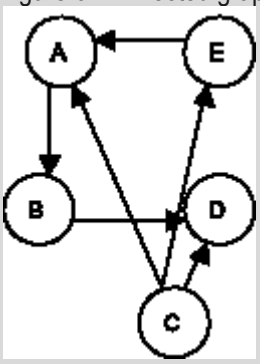


Directed Graph

This kind of graph contains ordered pairs of vertices. For example, graph vertices are {A,B,C,D,E} and edges are {(A,B),(B,D),(C,D)(C,A), (C,E), (E, A) }. Fig. 6.7 represents a graph having five nodes and six edges.

A direction is associated with each edge. The directed graph is also known as digraph.

Figure 6.7. Directed graph



V(G) = {A, B, C, D, E} and
group of directed edges = {(A,B), (B,D), (C,D)(C,A), (C,E), (E, A)}.

TERMINOLOGIES OF GRAPH

Weighted Graph

A graph is supposed to be weighted if its every edge is assigned some value which is greater than or equal to zero, i.e. non-negative value. The value is equal to the length between two vertices. Weighted graph is also called as network. Weighted graph is shown in Fig. 6.8.

Adjacent Nodes

When there is an edge from one node to another then these nodes are called adjacent nodes.

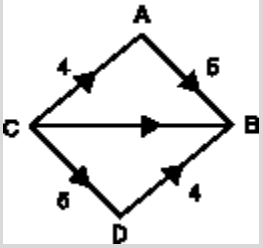
Incidence

In an undirected graph the edges v0 , v1 is incident on nodes. In a direct graph the edge v0, v1 is incident from node v0. It is incident to node v1.

Path

A path from edges u_0 to node u_n is a sequence of nodes u_0, u_1,u_2, u_3.. .u_{n-1}, u_n. Here, u_0 is adjacent to u_1, u_1 is adjacent to u_2 and u_{n-1} is adjacent to u_n.

Figure 6.8. A weighted graph



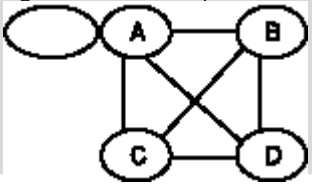
Length of Path

Length of path is nothing but total number of edges included in the path from source node to destination node.

Closed Path

When first and last nodes of the path are same, such path is known as closed path. In Fig. 6.9 closed path at node A is shown.

Figure 6.6. Closed path for node A



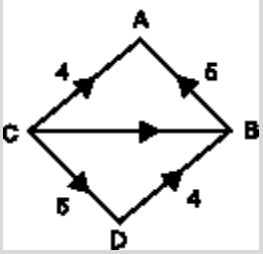
Simple Path

In this path all the nodes are different with an exception that the first and last nodes of the path can be similar.

Cycle

Cycle is a simple path. The first and last nodes are same. In other words, a closed simple path is a cycle. In a digraph a path is known as cycle if it has one or more nodes. The starting node is connected to the last node. In an undirected graph a path is called cycle if it has at least three nodes. The starting node is connected to last node. In the following figure path ACDBA is a closed path. Example of a cycle is shown in Fig. 6.10.

Figure 6.10. Example of a cycle



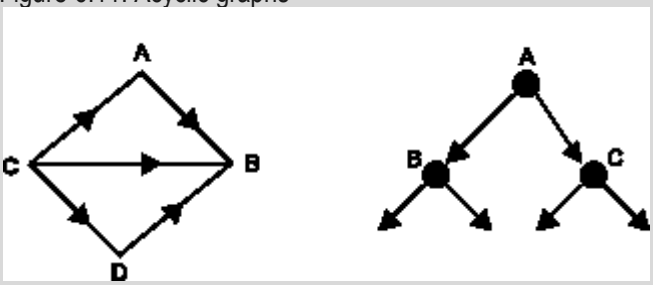
Cycle Graph

A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle. This is same as closed path shown in Fig. 6.10.

Acyclic Graph

A graph without cycle is called acyclic graph. Examples of acyclic graphs are shown in Fig. 6.11.

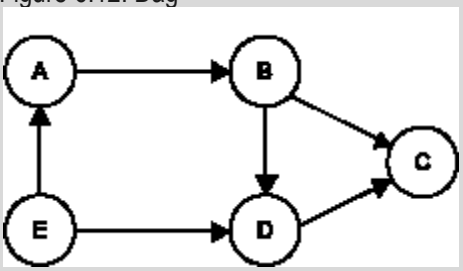
Figure 6.11. Acyclic graphs



Dag

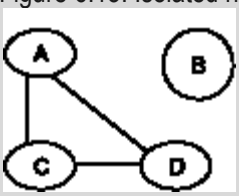
A directed acyclic graph is called dag after its acronym (reduction). Fig. 6.12 is a graph showing the dag.

Figure 6.12. Dag

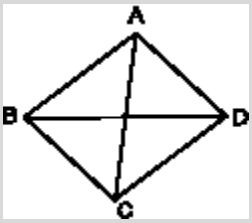


Degree
In an undirected graph, the total number of edges linked to a node is called degree of that node. In a digraph there are two degrees for every node called indegree and outdegree. In the above Fig. 6.12, E has two edges hence degree is 2. Similarly, D has degree three and so on.
Indegree
The indegree of a node is the total number of edges coming to that node. In Fig. 6.12, C is receiving two edges hence, the indegree is two.
Outdegree:
The outdegree of a node is the total number of edges going outside from that node. In the above Fig. 6.12 the outdegree of D is one.
Source
A node, which has only outgoing edges and no incoming edges, is called a source. The indegree of source is zero. In Fig. 6.12 the node E is the source since it does not have any incoming edges. It has only the outgoing edges.
Sink
A node having only incoming edges and no outgoing edges is called sink node. Node C in Fig. 6.12 is a sink node because it has only incoming edges but no outgoing edges.
Pendant Node
When indegree of node is one and outdegree is zero then such a node is called pendant node.
Reachable
If a path exists between two nodes it will be called reachable from one node to other node.
Isolated Node
When degree of node is zero, i.e. node is not connected with any other node then it is called isolated node. In Fig. 6.13 B node is the isolated node.

Figure 6.13. Isolated node



Successor and Predecessor
In digraph if a node V_0 is adjacent to node V_1 then V_0 is the predecessor of V_1 and V_1 is the successor of V_0.
Complete Graph
The graph in which any V_0 node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges= n(n-1)/2. The following figure shows the complete graph. The 'n' is the number of vertices present in the graph.



Articulation Point
On removing the node the graph gets disconnected, then that node is called the articulation point.
Biconnected Graph
The biconnected graph is the graph which does not contain any articulation point.

Multigraph

A graph in which more than one edge is used to join the vertices is called multigraph. Edges of multigraph are parallel to each other.

Figure 6.14. Multigraph

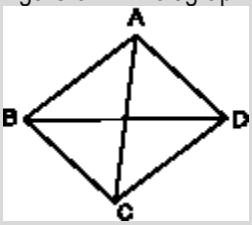


Fig. 6.14 shows the multigraph in which one can see the parallel edges between A and D, D and C, B and C, and A and B.

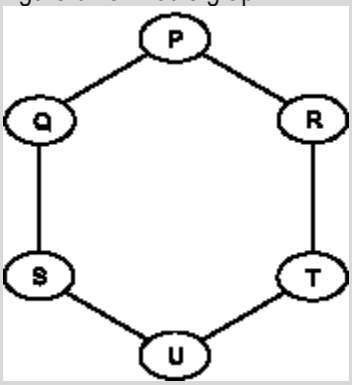
Regular Graph

Regular graph is the graph in which nodes are adjacent to each other i.e. each node is accessible from any other node.

GRAPH REPRESENTATION

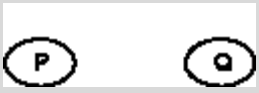
The graph can be implemented by linked list or array. Fig. 6.15 illustrates a graph and its representation and implementation is also described.

Figure 6.15. Modle graph



Different possibilities of graph representations are dependent on two cases:

1. If there is no edge between two nodes.



There is no edge between nodes P and Q.

2. If there is an edge between any two nodes.

The nodes P and Q are having an edge.

Hence, following Table 6.1 provides the representation of the graph (Fig. 6.15).

Table 9.1. Representation of a graph

Nodes	P	Q	R	S	T	U
P	x	✓	✓	x	x	x
Q	✓	x	x	✓	x	x
R	✓	x	x	x	✓	✓
S	x	✓	x	x	x	✓
T	x	x	✓	x	x	✓
U	x	x	x	✓	✓	x

As per the Table 6.1, there is an edge in between the nodes P and Q, P and R, and there is no edge between nodes P and S. The symbol indicates existence of edge and indicates absence of edge between two nodes.

From the above table one can predict the path to reach a particular node. For example, initial node is P and the destination node is U. We have to find the path to reach node U from P.

There is no edge between P and U. Then, find out the edge for nearest node in forward direction. By observing, we know there are two edges from P to Q and P to R. We can select either Q or R. Suppose, we have selected node Q, again find out next nearest successive node to Q by observing column Q. The next successive forward node will be S. Then, refer column S and it provides two edges Q and U. The node U is our solution. Thus, by using the above table, paths between any two nodes can be determined. The path should be P-Q-S-U. The graph can be represented by sequential representation and linked list representation.

Adjacency Matrix

The matrix can be used to represent the graph. The information of adjacent nodes will be stored in the matrix. Presence of edges from a particular node can be determined easily. The matrix can be represented by two-dimensional array. In a two-dimensional array [][], the first sub-script indicates row and second, column. For example, there are five nodes in the graph then the 0th row indicates node1 and so on. Likewise, column represents node1, node2, and so on. For example, consider a two-dimensional array.

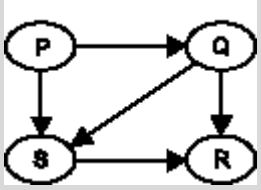
Nodes[j[k]

1 indicates presence of edge between two nodes j and k.

0 indicates absence of an edge between two nodes j and k.

Thus, the matrix will contain only 0 and 1 values.

Figure 6.16. An example of graph

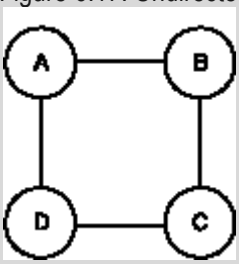


The matrix for the graph given in Fig. 6.16 would be

		P	Q	R	S
Matrix X =	P	0	1	0	1
	Q	0	0	1	1
	R	0	0	0	0
	S	0	0	1	0

In the above matrix ,Mat[0][1]=1, which represents an edge between node P and Q. Entry of one in the matrix indicates that there is an edge and 0 for no edge. Thus, adjacency is maintained in the matrix X. One can also represent the undirected graph with adjacency matrix. Fig. 6.17 is an undirected graph.

Figure 6.17. Undirected graph



The adjacency matrix for the above graph would be as follows:

		A	B	C	D
Matrix X =	A	0	1	0	1
	B	1	0	1	0
	C	0	1	0	1
	D	1	0	1	0

The above matrix is symmetric since x[i][j]= x[j][i].

In undirected graph the sum of row elements and column elements is the same. The sum represents the degree of the node. In this matrix the sum of any row or any column is 2, which is nothing but the degree of each node is 2.

We can also represent in the same way a weighted graph with adjacency matrix. The contents of the matrix will not be only 0 and 1 but the value is substituted with the corresponding weight.

For a null graph, that contains n vertices but no edges, all the elements of such null graph in an adjacency matrix are 0.

Figure 6.18. Null matrix

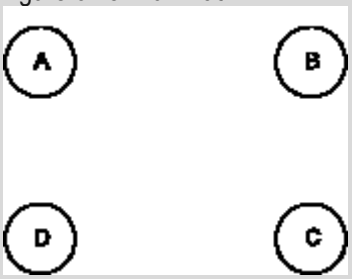


Fig. 6.18 represents the null graph and the adjacency matrix is as follows:

		A	B	C	D
X =	A	0	0	0	0
	B	0	0	0	0
	C	0	0	0	0
	D	0	0	0	0

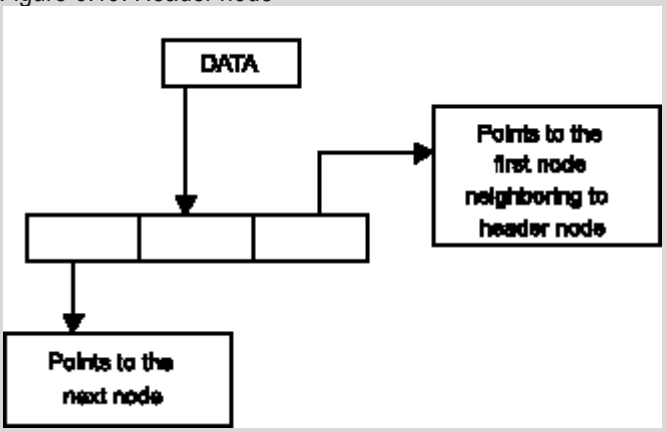
Adjacency List

Two lists are maintained for adjacency of list. The first list is used to store information of all the nodes. The second list is used to store information of adjacent nodes for each node of a graph.
In case a graph comprises of N nodes, then two lists are to be prepared.

1. First list keeps the track of all the N nodes of a graph.
2. The second list is used to keep the information of adjacent nodes of each and every node of a graph. As such there will be N lists that would keep the information of adjacent nodes.

Header node is used to represent each list, which is assumed to be the first node of a list. Fig. 6.19 represents a header node.

Figure 6.19. Header node

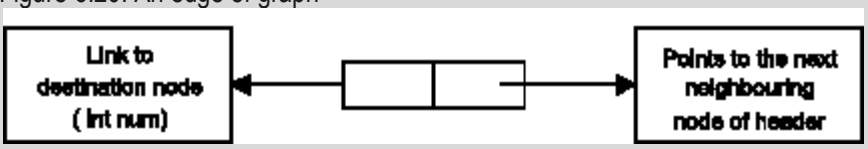



```
struct node
{
  struct *  next
  int num;
  struct  edge *aj;
};
```

Structure of an Edge

Fig. 6.20 is the representation of an edge.

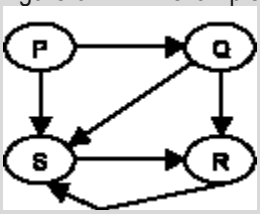
Figure 6.20. An edge of graph




```
struct edge
{
  int num;
  struct edge *ptr;
};
```

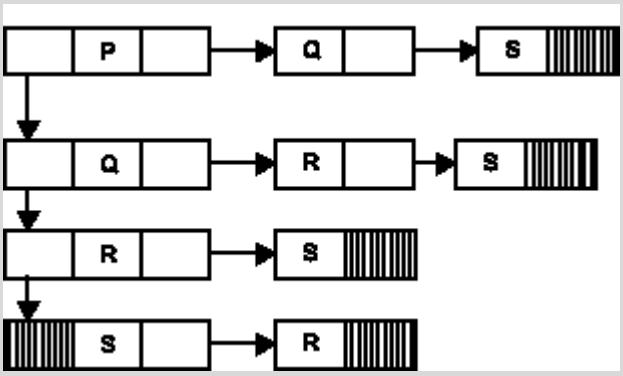
Consider a graph cited in Fig. 6.21.

Figure 6.21. An example of graph



Adjacency list for the above graph would be given in Fig. 6.22.

Figure 6.22. Adjacency list



TRAVERSAL IN GRAPH

Traversing in a graph is nothing but visiting each and every node of a graph. The following points are to be noted in a graph:

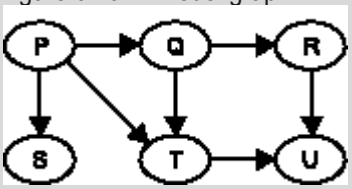
1. The graph has no first node or root. Therefore, the graph can be started from any node.
2. In graph, only those nodes are traversed which are accessible from the current node. For complete traversing of graph, the path can be determined by traversing nodes step by step.
3. In the graph, the particular node can be visited repeatedly. Hence, it is necessary to keep the track of the status of every node whether traversed or not.
4. In graph to reach a particular node, more paths are available.

Two techniques are used for traversing nodes in a graph. They are depth first and breadth first. These techniques have been elaborated in detail in the following sections.

Breadth First Search

This is one of the popular methods of traversing graph. This method uses the queue data structure for traversing nodes of the graph. Any node of the graph can act as a beginning node. Using any node as starting node, all other nodes of the graph are traversed. To shun repeated visit to the same node an array is maintained which keeps status of visited node.

Figure 6.26. A model graph



Take the node P of Fig. 6.26 as a beginning node. Initially, the node P is traversed. After this, all the adjacent nodes of P are traversed, i.e. Q, T and S. The traversal of nodes can be carried in any sequence. For example, the sequence of traverse of nodes is Q, S and T. The traversal will be

P Q S T

First, all the nodes neighbouring Q are traversed, then neighbouring nodes of S and finally T are taken into account. The adjacent node of Q is R and T is U. Similarly, the adjacent node of T is U and S does not have any adjacent node. Hence, in this step the traversal now should be in the following way:
P Q S T R U

Now, the new nodes obtained are R and U after traversing. The new adjacent node of R is U and U node does not have any adjacent node. Node U has been visited in the previous case hence it must be ignored in this step.

Algorithm

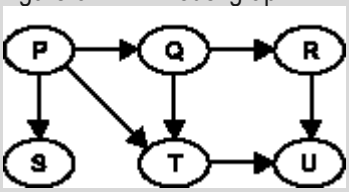
1. Put the root node on the queue.
2. Pull a node from the beginning of the queue and examine it.
    - If the searched element is found in this node, quit the search and return a result.
    - Otherwise push all the (so-far-unexamined) successors (the direct child nodes) of this node into the end of the queue, if there are any.
3. If the queue is empty, every node on the graph has been examined -- quit the search and return "not found".
4. Repeat from Step 2.

6.6.2 Depth First Search

In this method, also a node from graph is taken as a starting node. Traverse through all the possible paths of the starting node. When the last node of the graph is obtained and path is not available from the node; then control returns to previous node. This process is implemented using stack.

Consider the following graph shown in Fig. 6.27

Figure 6.27. A model graph



Consider, P as starting node. Then, traverse the node adjacent to P and we will get Q and then R (adjacent to Q) and U (adjacent to R). The traversal will be
P Q R U

The search is always carried in forward direction. After reaching to U, we reach the end of the path and further movement in forward direction is not possible. Hence, the controls go to the previous node and again traverse through the available paths for non-traversed nodes.
In reverse direction, we get the node R and it has unvisited node. Hence, Q is taken and it gives T. The node T gives U, but it is already visited. Therefore, control in reverse direction checks all the nodes. It takes P and it gives node S. The sequence of traversal will be
P Q R U T S

Algorithm:-

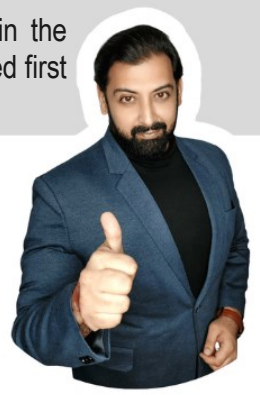
1. Set the starting point of traversal and push it inside the stack.
2. Pop the stack and add the popped vertex to the list of visited vertices.
3. Find the adjacent vertices for the most recently visited vertex(from the Adjacency Matrix).
4. Push these adjacent vertices inside the stack (in the increasing order of their depth) if they are not visited and not there in the stack already.
5. Step-5: Go to step-2 if the stack is not empty.

CHAPTER 7
SORTING & SEARCHING

SORTING

As already defined in the previous section sorting is a process in which records are arranged in ascending or descending order. In real life we come across several examples of sorted information. For example, in telephone directory the names of the subscribers and their phone numbers are written in alphabetial order. The records of the list of these telephone holders are to be sorted by their names. By using this directory, we can access the telephone number and address of the subscriber very easily.

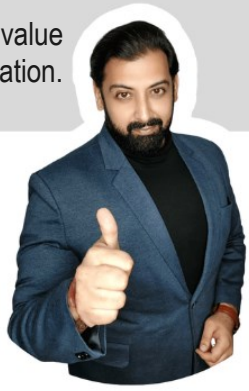
Bankers or businesspersons sort the currency denomination notes received from customers in the appropriate form. Currency denominations of Rs 1000, 500, 100, 50, 20, 10, 5, and 1 are separated first and then separate bundles are prepared.



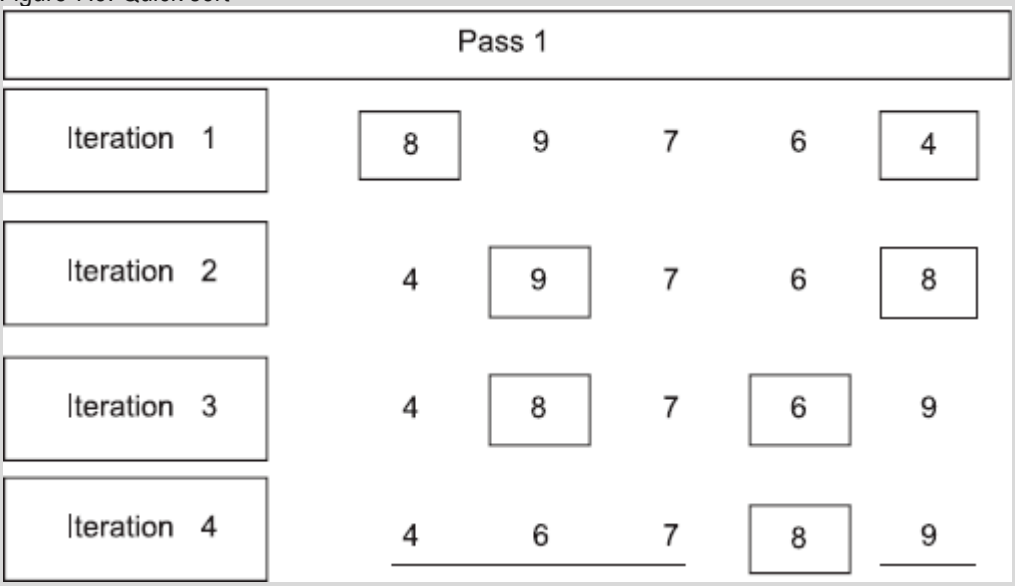
While we play the cards, initially the cards appear random but we keep them in ascending or descending order. In most of the offices the officials keep the files in a specific order for tracing them easily in future. Even at our homes, many times we keep the utensils in particular order such as increasing or decreasing height size. This helps us in accessing a particular item without difficulty. The sort method has great impact on data structures in our dally life.
For example, consider the five numbers 5, 9, 7, 4, 1.
The above numbers can be sorted in ascending or descending order.
The representation of these numbers in
Ascending order (0 to n): 1 4 5 7 9
Descending order (n to 0): 9 7 5 4 1
Similarly, alphabets can also be sorted.
Consider the alphabets B, A, D, C, E. These can be sorted in ascending or descending order.
Ascending order (A to Z): A B C D E
Descending order (Z to A): E D C B A.

QUICK SORT

It is also known as partition exchange sort. It was invented by CAR Hoare. It is based on partition. Hence, the method falls under divide and conquer technique. The main list of elements is divided into two sub-lists. For example, a list of X elements are to be sorted. The quick sort marks an element in a list called as pivot or key. Consider the first element J as a pivot. Shift all the elements whose value is less than J towards the left and elements whose value is greater than J to the right of J. Now, the key element divides the main list into two parts. It is not necessary that selected key element must be in the middle. Any element from the list can act as key element. However, for best performance preference is given to middle elements. Time consumption of the quick sort depends on the location of the key in the list.
Consider the following example in which five elements 8, 9, 7, 6, 4 are to be sorted using quick sort. The Fig. 7.5 illustrates it.
Consider pass 1 under which the following iterations are made. Similar operations are done in pass 2, pass 3, etc.
In iteration 1 the first element 8 is marked as pivot one. It is compared with the last element whose value is 4. Here, 4 is smaller than 8 hence the numbers are swapped. Iteration 2 shows the swapping operation.



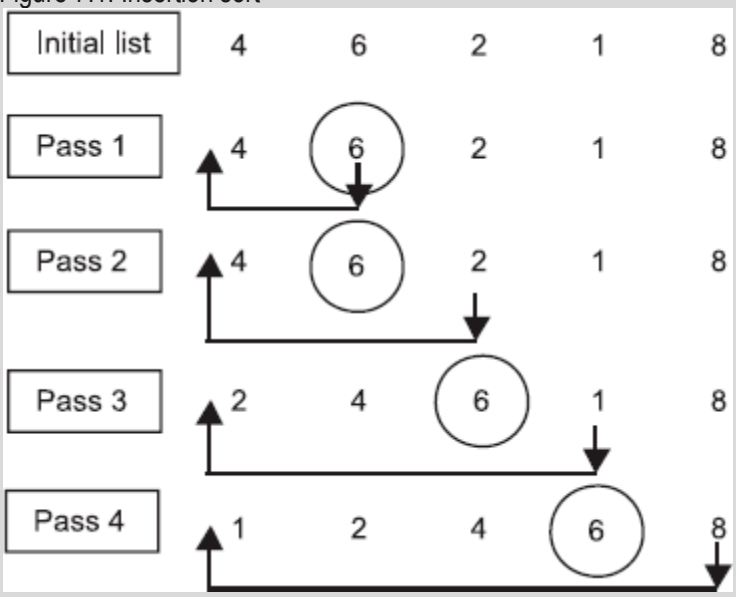
In the iteration 3 and 4, the position of 8 is fixed. In iteration 2, 8 and 9 are compared and swapping is done after iteration 2.
In iteration 3, 8 and 6 are compared and necessary swapping is done. After this, it is impossible to swap. Hence, the position of 8 is fixed. Because of fixing the position of 8 the main list is divided into two parts. Towards the left of 8 elements smaller than 8 are placed and towards the right greater than 8 are placed. Towards the right of 8 only one element is present hence there is no need of further swapping. This may be considered under pass 2.
However, towards the left of 8 there are three elements and these elements are to be swapped as per the procedure described above. This may be considered under pass 3.
Figure 7.5. Quick sort



INSERTION SORT

In insertion sort the element is inserted at appropriate place. For example, consider an array of n elements. In this type, swapping of elements is done without taking any temporary variable. The greater numbers are shifted towards the end of the array and smaller are shifted to beginning. Here, a real life example of playing cards can be cited. We keep the cards in increasing order. The card having least value is placed at the extreme left and the largest one at the other side. In between them the other cards are managed in ascending order.
Consider an example of playing cards. The playing cards having values 4, 6, 2, 1, 8 can be sorted according to the method described above, as
Fig. 7.1 illustrates the insertion sort.

Figure 7.1. Insertion sort



The process of insertion sort can be explained with following steps:

- Get the random integer list. In the above example the list contains elements 4, 6, 2, 1, 8.
- In the first pass the initial key is defined as 6 and compared with 4. As 6 is larger than 4 hence the position of 6 remains as it is.
- In the second pass element 2 is compared with the initial key 6 and 4 and 2 are placed appropriately. That is, its position is now at first.
- In the pass three the elements 1 is compared with the earlier three elements and placed appropriately. Its position is shown in Fig. 7.1.
- In the pass 4 the last element 8 is compared with the initial key and it is found greater hence it is placed on the right side of 6.

The final sorted list appears as 1, 2, 4, 6, 8.
Thus, in the insertion sort we are actually finding the proper place for the element to insert with respect to the key.

RADIX SORT

The radix sort is a technique, which is based on the position of digits. The number is represented with different positions. The number has units, tens, hundreds positions onwards. Based on its position the sorting is done. For example, consider the number 456, which is selected for sorting. In this number 6 is at units position and 5 is at tens and 4 is at the hundreds position. 3 passes would be needed for the

sorting of this number with this procedure. In the first pass, we place all numbers at the unit place. In the second pass all numbers are placed in the list with consents to the tens position digit value. Also, in the third pass the numbers are placed with consent to the value of the hundreds position digit. For example: 23,46,12,34,45,49,58,38,10,21.

In the first pass, the numbers are placed as per the value of the unit position digit. First, take number 23, which is present at the first in the list. Insert 23 in the packet labelled 3. Second number 46 is inserted in packet 6. 12 is inserted in the packet 2. The number 34 is inserted in the packet 4. 45 in the packet 5, 46 in the packet 6, 58 in the packet 8, 38 is also in the packet 8, 10 is in the packet 0 and 21 is in the packet 1. The numbers are inserted in the packets as per the sequence of that number in the original list.

Table 7.1. Sorting the numbers on unit place digit

Pass									38	
1st	10	21	12	23	34	45	46		58	49
Packets for unit place	0	1	2	3	4	5	6	7	8	9

In first pass sort the elements as per the value of the unit place digit. The operation is shown in the above Table 7.1. After the first pass we get the following list as:
10,21,12,23,34,45,46,58,38,49.

Perform the second pass operation on the list, which is sorted list after the first pass. As per the values of the digit of the tens position, the place of the elements is fixed. At first, take the first element 10 of the list, which is obtained after the first pass. Insert 10 into the packet 1; insert 21 in the 2(second) packet. This procedure is repeated until the end of the list. In the second pass, the numbers are as:

Table 7.2. Sorting the numbers on ten's place digit

Pass					49					
		12	23	38	46					
2nd		10	21	34	45	58				
Packets for unit place	0	1	2	3	4	5	6	7	8	9

Table 7.2 shows the second pass for sorting the numbers. The sorted list after the second pass is as:
10,12,21,23,34,38,45,46,49,58.

By appropriately placing numbers, the sorted list of the items or elements is obtained. In this example of radix sort, two passes are required for complete sorting. The number of passes depends upon the length of the greatest number in the list. If the list contains the numbers greater than the 99, then the number of passes must be greater than 3 or equal to 3.

HEAP SORT

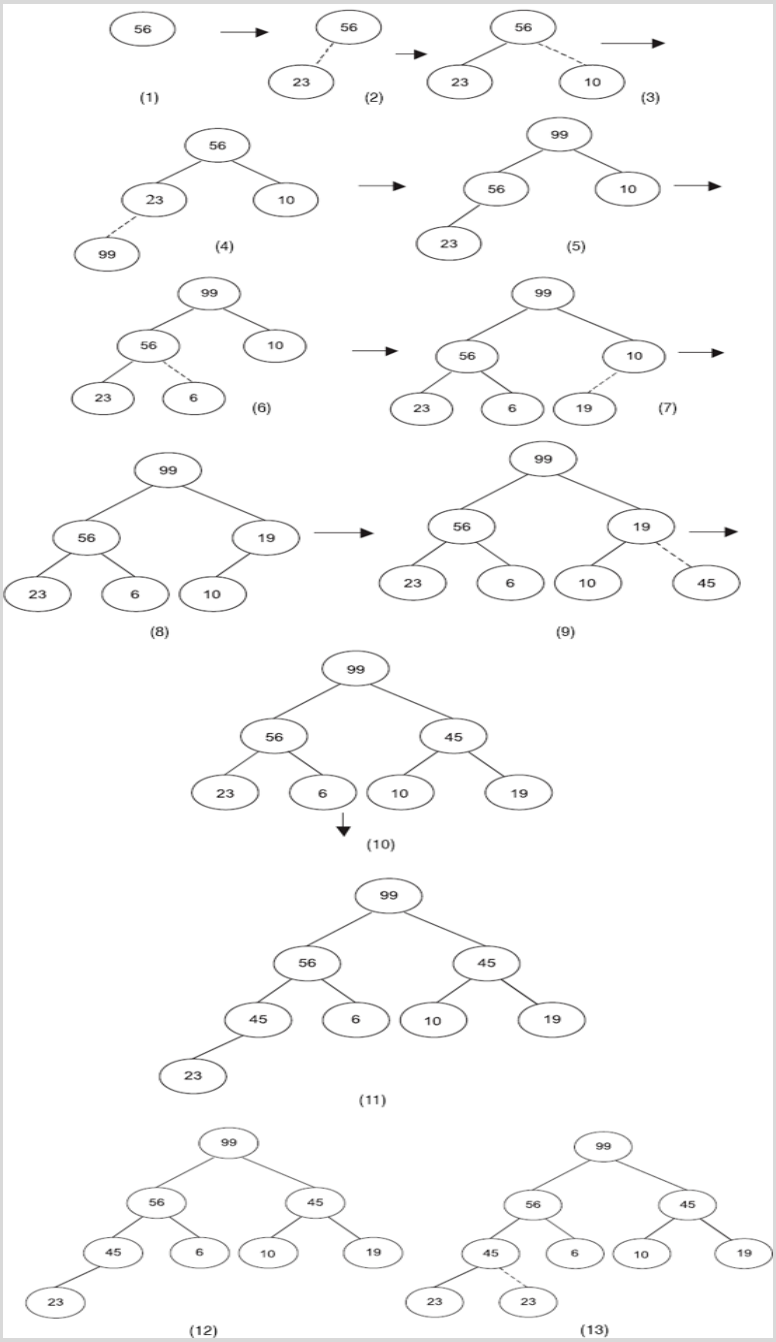
In heap sort, we use the binary tree in which the nodes are arranged in specific prearranged order. The rule prescribes that each node should have bigger value than its child node. The following steps are to be followed in heap sort:

1. Arrange the nodes in the binary tree form.
2. Node should be arranged as per the specific rules.
3. If the inserted node is bigger than its parent node then replace the node.
4. If the inserted node is lesser than its parent node then do not change the position.
5. Do not allow entering nodes on right side until the child nodes of the left are fulfilled.
6. Do not allow entering nodes on left side until the child nodes of the right are fulfilled.
7. The procedure from step 3 to step 6 is repeated for each entry of the node.

Consider the numbers 56, 23, 10, 99, 6, 19, 45, 45, 23 for sorting using heap. Sorting process is illustrated in Fig. 7.7.

1. At first, we pickup the first element 56 from the list. Make it as the root node.
2. Next, take the second element 23 from the list. Insert this to the left of the root node 56. Refer the Fig. 7.7 (2).
3. Then, take the third element 10 from the list for insertion. Insert it to the right of the root node.
4. Take the fourth element 99. Insert it to the left side node 23. The inserted element is greater than the parent element, hence swap the 99 with 23. But the parent node 56 is smaller than the child node 99 hence 99 and 56 are swapped. After swapping 99 becomes the root node. This is shown in Fig. 7.7 (4) and (5).
5. Consider the next element 6 to insert in the tree. Insert it at left side . There exists left node, hence insert it to the right of the 56. Refer Fig. 7.7 (6).
6. Element 19 is inserted to the right side of the 99 because the left side gets full. Insert the element 19 to the right node 10. However, the parent element is lesser than the child, hence swaps the 19 with 10.
7. Now, element 45 is to be inserted at the right side of 19. However, the parent element is having the value lesser than the child element hence swap the 45 with 19. Refer the Fig. 7.7 (9) and (10).
8. Now, the right side is fully filled hence add the next element 45 to the left. The element 45 is inserted at the last node of the left, i.e. 23. However, the parent element is having the value lesser than the child element hence swap 45 with 23. Refer the Fig. 7.7 (11) and (12).
9. Insert the last element 23 to the left side node, i.e. 45. The left of the 45 is already filled hence insert the element 23 at the right of the 45. Refer the Fig. 7.7 (13).
10. At last, we get a sorted heap tree, as shown in Fig. 7.7 (13).

Figure 7.7. Heap sort



MERGE SORT

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p \,..\, r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p \,..\, r]$:

1. **Divide Step**

If a given array *A* has zero or one element, simply return; it is already sorted. Otherwise, split *A*[*p* .. *r*] into two subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*], each containing about half of the elements of *A*[*p* .. *r*]. That is, *q* is the halfway point of *A*[*p* .. *r*].

2. **Conquer Step**

Conquer by recursively sorting the two subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*].

3. **Combine Step**

Combine the elements back in *A*[*p* .. *r*] by merging the two sorted subarrays *A*[*p* .. *q*] and *A*[*q* + 1 .. *r*] into a sorted sequence. To accomplish this step, we will define a procedure MERGE (*A*, *p*, *q*, *r*).

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

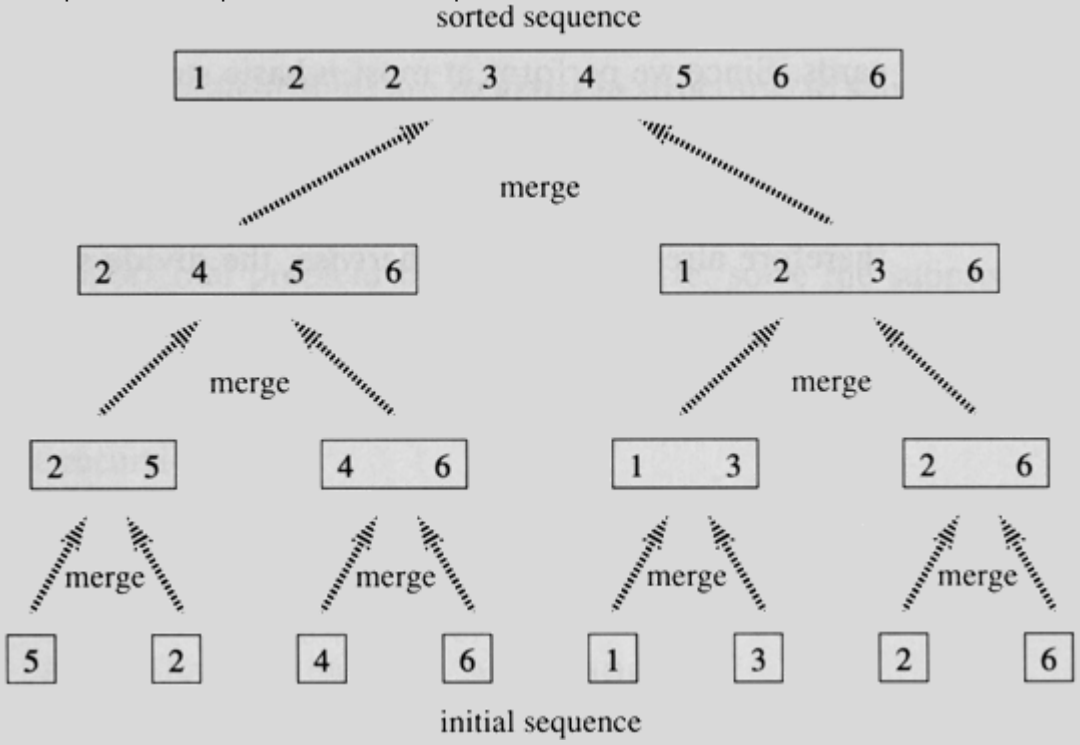
Algorithm: Merge Sort

To sort the entire sequence A[1 .. n], make the initial call to the procedure MERGE-SORT (*A*, 1, *n*).

MERGE-SORT (*A*, *p*, *r*)

```
1.   IF p < r                          // Check for base case
2.      THEN q = FLOOR[(p + r)/2]      // Divide step
3.           MERGE (A, p, q)           // Conquer step.
4.           MERGE (A, q + 1, r)       // Conquer step.
5.           MERGE (A, p, q, r)        // Conquer step.
```

Example: Bottom-up view of the above procedure for *n* = 8.



SHELLSORT

Shellsort is one of the oldest sorting algorithms, named after its inventor D.L. Shell
The idea of Shellsort is the following:

a. arrange the data sequence in a two-dimensional array
b. sort the columns of the array

The effect is that the data sequence is partially sorted. The process above is repeated, but each time with a narrower array, i.e. with a smaller number of columns. In the last step, the array consists of only one column. In each step, the sortedness of the sequence is increased, until in the last step it is completely sorted. However, the number of sorting operations necessary in each step is limited, due to the presortedness of the sequence obtained in the preceding steps.
Example: Let 3 7 9 0 5 1 6 8 4 2 0 6 1 5 7 3 4 9 8 2 be the data sequence to be sorted. First, it is arranged in an array with 7 columns (left), then the columns are sorted (right):

```
3 7 9 0 5 1 6          3 3 2 0 5 1 5
8 4 2 0 6 1 5    →     7 4 4 0 6 1 6
7 3 4 9 8 2            8 7 9 9 8 2
```

Data elements 8 and 9 have now already come to the end of the sequence, but a small element (2) is also still there. In the next step, the sequence is arranged in 3 columns, which are again sorted:

```
3 3 2          0 0 1
0 5 1          1 2 2
5 7 4          3 3 4
4 0 6          4 5 6
1 6 8    →     5 6 8
7 9 9          7 7 9
8 2            8 9
```

Now the sequence is almost completely sorted. When arranging it in one column in the last step, it is only a 6, an 8 and a 9 that have to move a little bit to their correct position.

SEARCHING

Searching is a technique of finding an element from the given data list or set of the elements like an array, list, or trees. It is a technique to find out an element in a sorted or unsorted list. For example, consider an array of 10 elements. These data elements are stored in successive memory locations. We need to search an element from the array. In the searching operation, assume a particular element n is to be searched. The element n is compared with all the elements in a list starting from the first element of an array till the last element. In other words, the process of searching is continued till the element is found or list is completely exhausted. When the exact match is found then the search process is terminated. In case, no such element exists in the array, the process of searching should be abandoned.

In case the given element is existing in the set of elements or array then the search process is said to be successful as per Fig. 8.1. It means that the given element belongs to the array. The search is said to be unsuccessful if the given element does not exist in the array as per Fig. 8.2. It means that the given element does not belong to the array or collection of the items.

The complexity of any searching method is determined from the number of comparisons performed among the collected elements in order to find the element. The time required for the operation is dependent on the complexity of the operation or algorithm. In other words, the performance of searching algorithm can be measured by counting the comparisons in order to find the given element from the list. There are three cases in which an element can be found.

In the best case, the element is found during the first comparison itself. In the worst case, the element is found only in the last comparison. Whereas in the average case, number of comparisons should be more than comparisons in the best case and less than the worst case.

The searching can be classified into following types:

1. Linear search (sequential)
2. Binary search.

LINEAR (SEQUENTIAL) SEARCH

The linear search is a conventional method of searching data. The linear search is a method of searching a target element in a list sequence. The expected element is to be searched in the entire data structure in a sequential method from starting to last element. Though, it is simple and straightforward, it has serious limitations. It consumes more time and reduces the retrieval rate of the system. The linear or sequential name implies that the items are stored in systematic manner. The linear search can be applied on sorted or unsorted linear data structure.

Fig. 8.1 and 8.2 shows the searching by using the linear search method.

Figure 8.1.

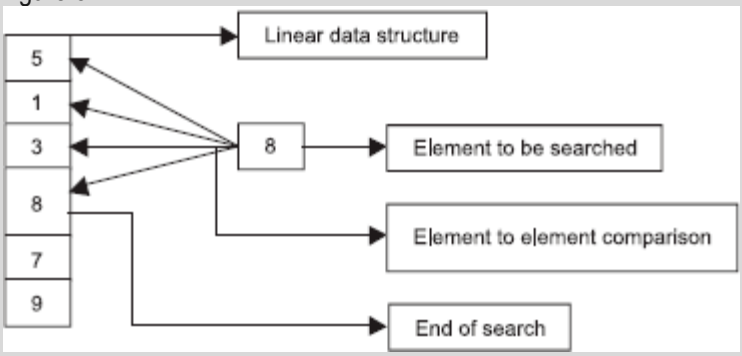
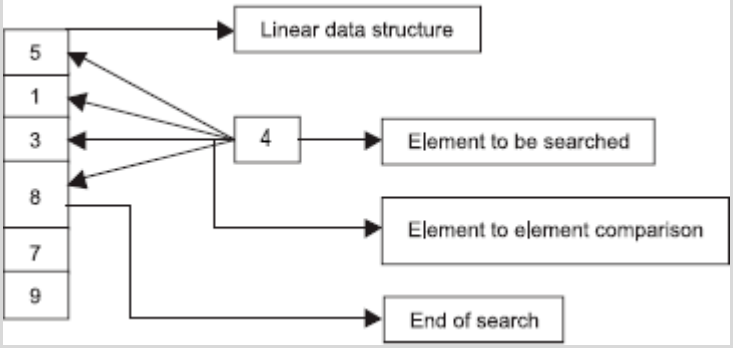


Figure 8.2. Unsuccessful search



Indexed Sequential search

Efficiency of searching sorted files is increased by this method. An auxiliary file, called index, is formed in addition to the sorted file. Each record in the index consists of a key and a pointer to the record in the file corresponding to the key. The records in the file and the index are sorted on the key. Size of the index file determines which records are to be represented in the index. If the index size is 1/r of the file every rth record is represented in the index file. This is shown in Fig.7.1.1.

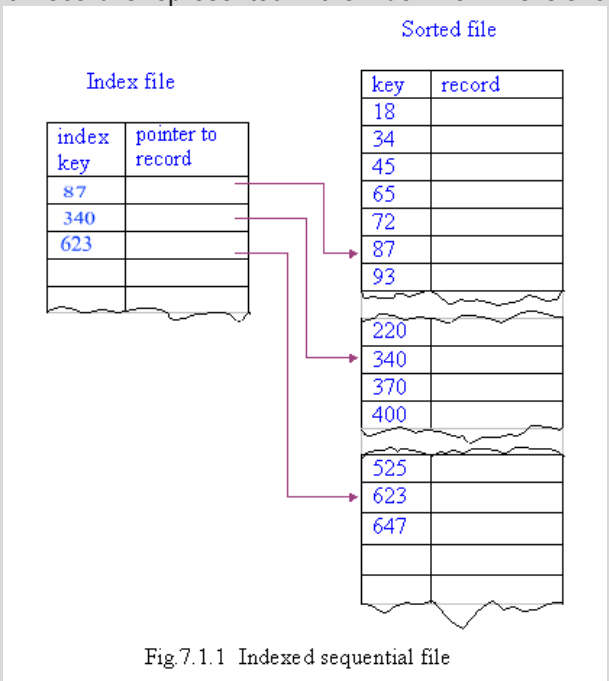


Fig.7.1.1 Indexed sequential file

Key is located by sequentially searching the index file first instead of the sorted file. On completion of search in the index table, a pointer, corresponding to the key identified, is returned. The index pointer value returned is either equal or less than the target key. The sequential search for the target key is then

continued in the sorted file starting from the point specified by the pointer. The search time is sharply reduced here as the search is conducted first in the index table, which is short and when the correct index position is found a second sequential search is performed on a smaller portion of the sorted file starting from the point indicated by the pointer.

When the size of the file is large, the index file itself may become large or if the index is made small the adjacent keys in the index file would be far apart. In such cases a secondary index is created. The secondary index is an index to primary index, which points to entries in the given file. This is shown in Fig.7.1.2. In this case sequential search is carried out first in secondary index. With the pointer obtained primary index is searched and then finally the given original sorted file.

Deletion of records from indexed sequential file is effected by flagging entries. Deleted entries are ignored for searching. They can be over written. Records in indexed sequential file may have to be shifted to accommodate insertion. Alternately insertion can be done in an overflow area and then the records linked with pointers.

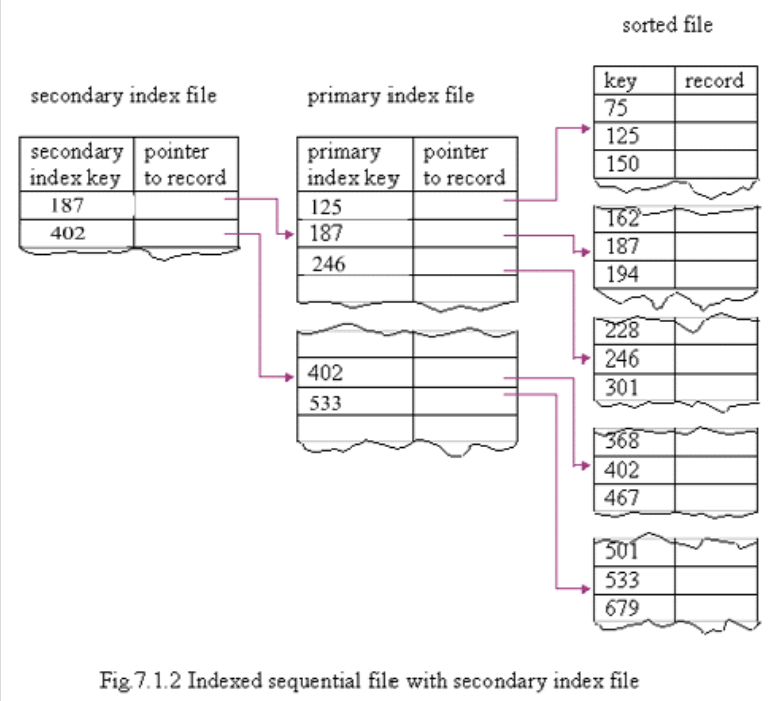


Fig.7.1.2 Indexed sequential file with secondary index file

BINARY SEARCH

The binary search approach is different from the linear search. Here, search of an element is not passed in sequence as in the case of linear search. Instead, two partitions of lists are made and then the given element is searched. Hence, it creates two parts of the lists known as binary search. Fig. 8.3 illustrates the operation of the binary search.

Binary search is quicker than the linear search. It is an efficient searching technique and works with sorted lists. However, it cannot be applied on unsorted data structure. Before applying binary search, the linear data structure must be sorted in either ascending or descending order. The binary search is unsuccessful if the elements are unordered. The binary search is based on the approach divide-and-conquer. In binary search, the element is compared with the middle element. If the expected element falls before the middle element, the left portion is searched otherwise right portion is searched.

The binary searching method is illustrated in Fig. 8.3.

Figure 8.3. Binary search

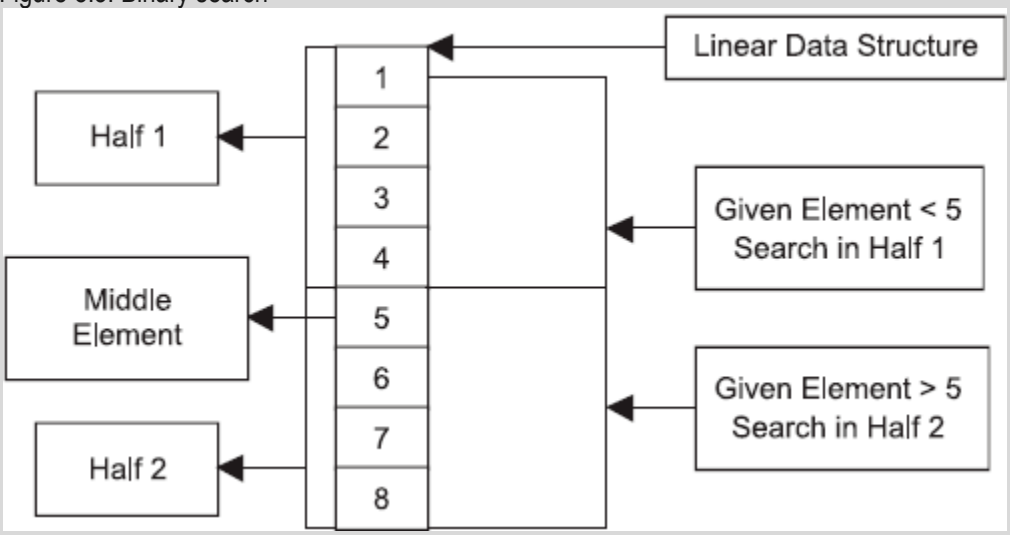
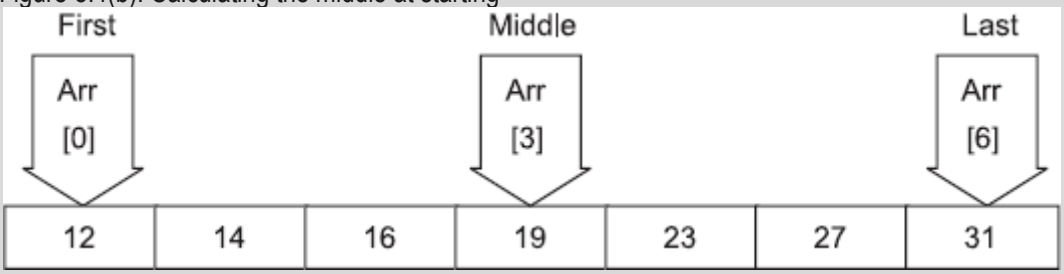


Fig. 8.3 represents the operation of binary search. The total elements in the list are 8, and they are shown in the ascending order. In case the key element is less than 5 then searching is done in lower half otherwise in the upper half. The process of searching can be followed from the following Fig. 8.4.
Consider the array of seven elements. The array can be represented in Fig. 8.4(a). The elements of an array are 12, 14, 16, 19, 23, 27, and 31. Assume that we want to search the element 27 from the list of elements. The steps are as follows:

Figure 8.4(a). Array in memory

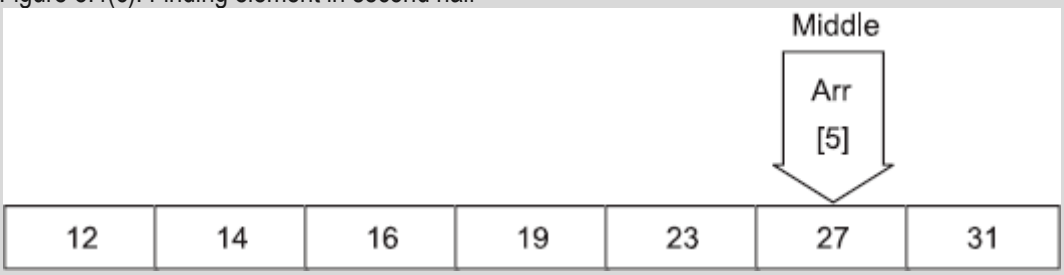
Arr[0]	Arr[1]	Arr[2]	Arr[3]	Arr[4]	Arr[5]	Arr[6]
12	14	16	19	23	27	31

1. The element 12 and 31 are at positions first and last, respectively.
2. Calculate the mid-value which is as
   MID=(FIRST+LAST)/2.
   MID = (0+6)/2.
   MID=3.
3. The key element 23 is to be compared with the mid-value. If the key is less than the mid then the key element is present in the first half else in the other half; in this case the key is on the right half. Hence, first is equal to middle+1.
   Figure 8.4(b). Calculating the middle at starting



4. Calculate the middle of the second half.
   MIDDLE=(FIRST+LAST)/2.
   MIDDLE = (4+6)/2
   MIDDLE=5.
5. Again, the middle divides the second half into the two parts, which is shown in Fig. 8.4(c).
6. The key element 23 is lesser than the middle 27, hence it is present in the left half (towards the left of 27).

Figure 8.4(c). Finding element in second half



7. At last, the middle is calculated as
   MIDDLE = (FIRST + LAST)/2.
   MIDDLE=(4+4)/2.
   MIDDLE=4.
8. Now, the middle is 4 position and the middle element is 23. The key is searched successfully.

Figure 8.4(d). Result of searching

HASHING METHOD

Hashing technique is one of the complex searching techniques. In this section, we consider a class of search techniques whose search time is dependent on the number of entries available in the table. In this method, we fix the position of the key (element) into the table or the file, which is determined by the hash function. The function in which we use this key is known as hashing function.

HASHING FUNCTION

In order to follow the operation of hashing function, consider an array that comprises a list containing some finite number of elements. Each element of it is a key. On performing operations on each key some value is obtained. The key is assigned to the index, which is having the appropriate value. The table is prepared with the keys called a hashing table.

Let us assume the following array that will hold the hash table:
12,22,32,17,19,28, 15,29,16,18,13,8.

Let us use the division method for the following hash table. We divide the key by '3' and the key is placed in the index based on the remainders obtained. The keys having equal remainders are placed in the same index. Any number divided by 3 always returnes 0,1, or 2. Hash table is prepared with three indices such as 0,1,2. All above keys are stored in these three indices as shown in Table 8.1.
Take the first key '12', divide it by '3'. The reminder is '0', hence insert it into the '0' index.
Then, take the next key '22' divide it by the '3', the reminder is '1'. Hence, insert the key in to the '1' index.
In the same way divide the key '32' by the 3, its reminder is 2. Place it into the '2' index in the table.
For the remaining elements, the same procedure is to be continued and by knowing reminders the numbers can be placed appropriately in the respective index table.
Table 8.1 explains hashing table.

Table 8.1. Hash table	
Index	Key
0	12, 15, 18
1	22, 19, 28, 16, 13
2	32, 17, 29, 11

In short, the hashing function takes the key, and maps it to some index to the array. In our example, division function has been selected. The programmer can take any other function. This function maps several different keys to the same index. In the above table, key 12,15 and 18 are assigned to the index 0. While keys, 22,19,28,16 and 13 are placed in index 1. Keys 32,17,29 and 11 are in the index 2.
After preparation of the index table, it becomes easier to search the element from the list. One can use any searching method to search the element from the hashing table.

DIVISION METHOD

The commonly used hashing method is the division method. In this hashing function, the integer key is divided by some number. The remainder value is taken as the hashing value. This method can be expressed as follows:
H(k)=k(mod x)+1.

In above equation the H(k) is the hashing function. In addition, the k is the key, which is to be used in hashing. The k(mod x) shows the result of division k by x.
In the above equation, the 'k' is the key and the 'x' is the size of list. We can take any value for the x.
For example, key is 25 and x=10
H(25)=25(mod 10)+1 =5+1=6.

Consider an example to prepare a hashing table and search the prompted number from the hashing table. Each number (key) is divided (mod operation) by 3. No addition of 1 is done after division. It is up to the user how to deal with the problem. The user can perform simply division operation and note down the remainders and place them in appropriate index. As such, there will be 3 indices.

MID-SQUARE METHOD

In mid-square method the key is multiplied with itself and middle digit is chosen as the address of that element in the hashing table. It is defined as follows:
H(k)=(k*k)

For example, if the key is k=12
H(12)=(12*12)=144.

In this example, the key is 12, its square is 144 and its middle digit is 4. Hence, 12 is placed in the hash table in the index 4 as index 4 is the address of 12. Similarly, for other keys the squares are calculated

and mid digits are evaluated. Based on the mid digits the keys are placed in the appropriate index. Here, the mid digit obtained can be from 0 to 9. Hence, we may need up to 10 indices starting from index 0 through index 9.

For example, if the key is k=10

H(10)=(10*10)=100. Its middle digit is 0. Hence, number 10 would be placed in index 0. Similarly, the same procedure is adopted for all other numbers and they are placed in different indices based on the mid digit value.

Consider the following number for constricting the hashing table:

12,14,18,20,36,31,27,35,23,59.

Table 8.2. Table of key, square and mid

Key	Square	Mid	Key	Square	Mid
12	144	4	31	961	6
14	196	9	27	729	2
18	324	2	15	225	2
20	400	0	23	529	2
26	676	7	23	841	4

Figure 8.3. Hashing table using the mid-square method

Square	Mid	Key	Square
0	20	5	-
1	-	6	31
2	18, 27, 15, 23	7	26
3	-	8	-
4	12, 29	9	14

FOLDING METHOD

In the folding method, the key is divided into number of parts. The parts, of course, are dependent on the length of the key. All the parts are added together. If the carry results from this operation it is to be ignored and the number what remains behind after ignoring the carry is the address of key. By using this method, we fold the key hence this method is called as folding method.

For example,

1. If key k= 879987

Then, by using this method, we divide it into two parts like 879 and 987. They are added and then carry ignored and the number after ignoring the carry becomes the address of the key.

879 +987 = 1866. Its most significant digit is 1. Hence, after ignoring 1, the number that is left is 866 which is the address of the key.

2. If k=678897678

In this, key is divided into the 3 parts. They are 678, 897 and 678.

By adding, 678+897+678 = 2253.

After ignoring last carry 2, the number that is left is 253. Therefore, 253 is the address of the key. Thus, the folding method is useful in converting multiword key into a single word. By applying hashing function, the key can be identified.

MULTIPLICATIVE HASHING FUNCTION

The name itself indicates that the function is having the multiplication operation in it. The multiplicative hashing function can be written as

H(key)=(b(c*key mod 1))+1.

In this hashing method, the key is multiplied with the number 'c'. The c is used in this function as a constant. After performing the mod of this product with 1 again multiplication operation with the number b is done and 1 is added. The number obtained is the index number for the placing the key.
Mainly, there are two types of searching methods used with hashing function. One is external searching and other is internal search. When the records are bulky then the records can be stored in the file and the file is stored on the disk, tape, etc. The record is outside of the computer memory. On that file the searching operation is to be performed. This searching operation is called as the external searching. Two techniques are used for the external search, one is hashing method and other is tree search method. In another case, when the records are shorter and in less numbers the same can be stored in the computer memory like array, list, etc. In general, the linear searching and binary search are used in the internal searching.

DIGIT ANALYSIS METHOD

Digit analysis method is the hashing function in which the analysis is done on the whole digits. In this method some digits of some positions of the number are taken out. Then, by making the reverse of the selected digits a value is obtained. The key is transferred to that location. Assume a key 348526917 and just select the digits at the 2, 5 and 7 position. They are as 1,2 and 8. By combining these digits we get a number 128 and after reversing the number have 821. Now, the key is transferred at the address location 821.

Collision Resolution Schemes

in amar sir's class