What is data structure?

- Data structure is a way of organizing all data items and establishing relationship among those data items.

- Data structures are the building blocks of a program. Data structure mainly specifies the following four things:

· Organization of data.
· Accessing methods
· Degree of associativity
· Processing alternatives for information

To develop a program of an algorithm, we should select an appropriate data structure for that algorithm. Therefore algorithm and its associated data structures form a program.

Algorithm + Data structure = Program

A **static data structure** is one whose capacity is fixed at creation. For example, array.

A **dynamic data structure** is one whose capacity is variable, so it can expand or contract at any time. For example, linked list, binary tree etc.

Classification of data structure:

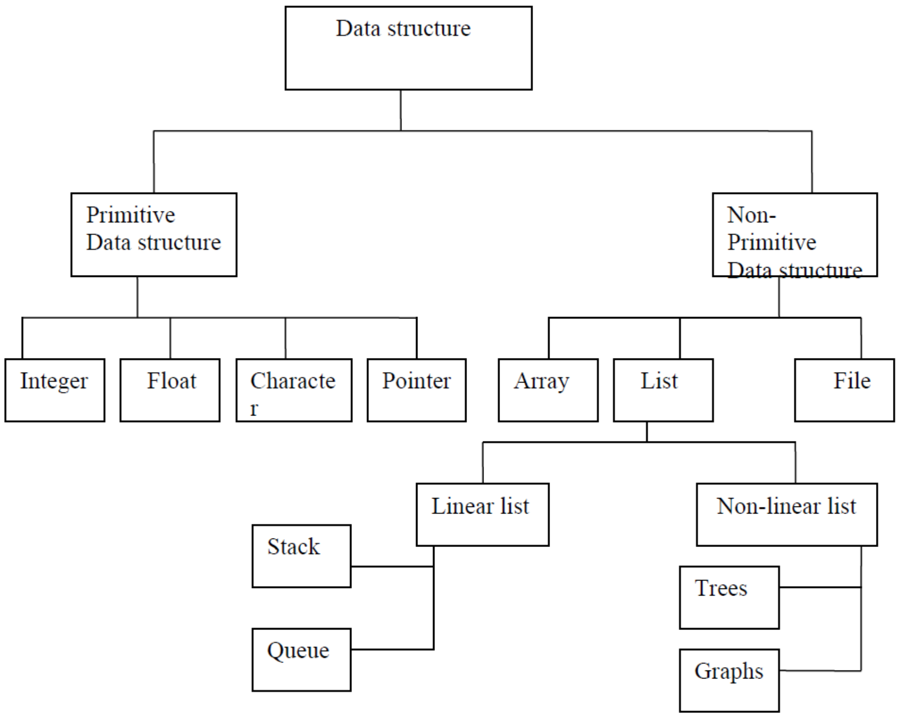


Fig:- Classification of data structure

Data structures are normally divided into two broad categories.

(i) Primitive data structures
(ii) Non-primitive data structures

<u>Primitive Data Structures</u>

These are basic structures and are directly operated upon by the machine instructions. These, in general, have different representations on different computers. For example, integers, floating point numbers, characters constants, pointers etc. are primitive data structures.

<u>Non-Primitive Data Structures</u>

These are more sophisticated data structures. These are derived from the primitive data structures. The non-primitive data structures emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items. Arrays, lists, files etc. are examples.

- ❖ Linear data structures

  The data structures in which the elements form a sequence or the data structures in which the processing of data items is possible in linear fashion i.e. data can be processed one by one sequentially are called linear data structures. E.g. arrays, linked list, stack, queues etc.

- ❖ Non- linear data structures

  The data structures in which the elements do not form a sequence or the data structure in which insertion and deletion is not possible in linear fashion are known as non linear data structures. E.g. trees, graph etc.

Data types

A data type is collection of values and set of operations on those values. For example the type integer (int) consists of values {0,+1, -1, +2, -2, +3, -3, ………………., max int, min int} and different types of operation which can be performed on those values are addition subtraction, multiplication, division etc.
Alternatively, it can be defined as the collection of different types of operation and operands is known as data types.

Abstract Data Types (ADTs)

An **abstract data type** is a data type whose representation is in hidden from, and of no concern to the application code. For example, when writing application code, we don't care how strings are represented: we just declare variables of type *String*, and manipulate them by using string operations.
Once an abstract data type has been designed, the programmer responsible for implementing that type is concerned only with choosing a suitable data structure and coding up the methods.
On the other hand, application programmers are concerned only with using that type and calling its methods without worrying much about how the type is implemented.
An abstract data type is a mathematical construct of data type that is organized in such a way that the specification of the values and specification of the operation on the values is separated from the representation of the values and the implementation of the operation or a mathematical model with collection of operations defined on that model is an abstract data type.
An ADT only defines a formal description, what the operations to do; it is not concerned with implementation detail. For example; set of integers together with the operations of union, intersection and set difference is an ADT. Other examples are lists, stacks, queues, rational numbers, trees, graph etc

Algorithm:

An algorithm is a precise specification of a sequence of instructions to be carried out in order to solve a given problem. Each instruction tells what task is to be done. There should be a finite number of instructions in an algorithm and each instruction should be executed in a finite amount of time.

Properties of Algorithms:

- **Input:** A number of quantities are provided to an algorithm initially before the algorithm begins. These quantities are inputs which are processed by the algorithm.
- **Definiteness:** Each step must be clear and unambiguous.
- **Effectiveness:** Each step must be carried out in finite time.
- **Finiteness:** Algorithms must terminate after finite time or step

- **Output:** An algorithm must have output.
- **Correctness:** Correct set of output values must be produced from the each set of inputs.

Write an algorithm to find the greatest number among three numbers:
Step 1: Read three numbers and store them in X, Y and Z
Step 2: Compare X and Y. if X is greater than Y then go to step 5 else step 3
Step 3: Compare Y and Z. if Y is greater than Z then print "Y is greatest" and go to step 7 otherwise go to step 4
Step 4: Print "Z is greatest" and go to step 7
Step 5: Compare X and Z. if X is greater than Z then print "X is greatest" and go to step 7 otherwise go to step 6
Step 6: Print "Z is greatest" and go to step 7
Step 7: Stop

Analyzing algorithm

Analyzing algorithm means predicting the computer resources that an algorithm requires. There are various resources of concern like memory, communication bandwidth, computer hardware etc. but two most important resources are the computation time and the memory needed for it. An algorithm is efficient if it uses less memory space and runs fast. Hence to measure the efficiency of algorithm we must check time and space complexity.

Space complexity

Space occupied by the algorithm in memory while in runs.

Time complexity

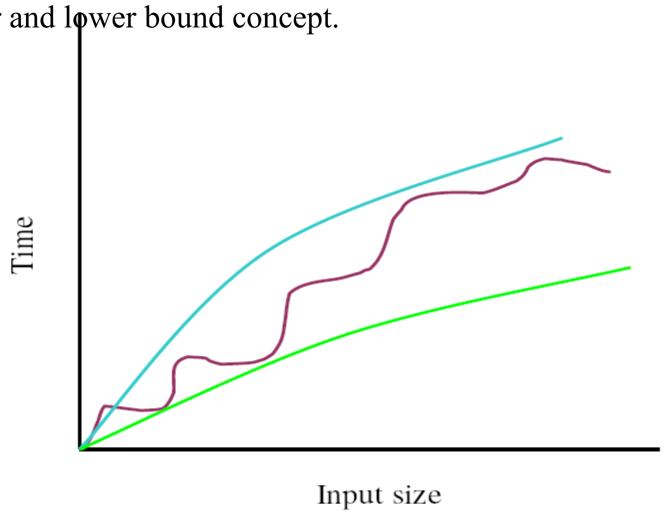
Time complexity of an algorithm is the computer time it needs to run i.e. the sum of compile time and run time. Since the compile time doesn't depend on the instance characteristics our major concern is just the run time.

Best, Worst and Average case

Best case complexity gives lower bound on the running time of the algorithm for any instance of input(s). This indicates that the algorithm can never have lower running time than best case for particular class of problems.
Worst case complexity gives upper bound on the running time of the algorithm for all the instances of the input(s). This insures that no input can overcome the running time limit posed by worst case complexity.
Average case complexity gives average number of steps required on any instance of the input(s).

Asymptotic Notation

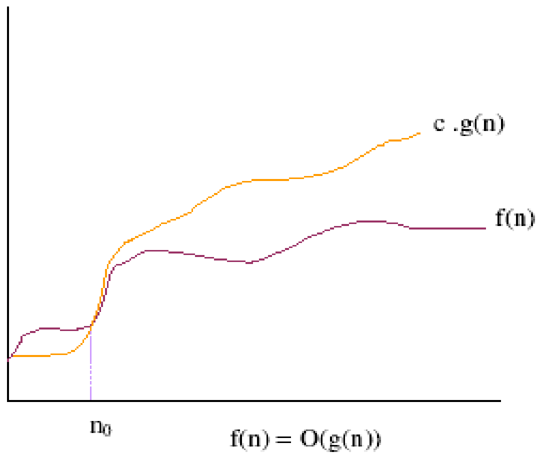
Complexity analysis of an algorithm is very hard if we try to analyze exact. we know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input. So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier. For this purpose we need the concept of asymptotic notations. The figure below gives upper and lower bound concept.



Big Oh (O) notation

When we have only asymptotic upper bound then we use O notation. A function f(x)=O(g(x)) (read as f(x) is big oh of g(x)) iff there exists two positive constants c and x_0 such that for all x >= x0, 0 <= f(x) <= c*g(x)

The above relation says that g(x) is an upper bound of f(x)



For all values of n >= n0, plot shows clearly that f(n) lies below or on the curve of c*g(n)

Examples

1. $f(n) = 3n^2 + 4n + 7$

   $g(n) = n^2$, then prove that f(n) = O(g(n)).

   Proof: let us choose c and n_0 values as 14 and 1 respectively then we can have

   f(n) <= c*g(n), n>=n_0 as

   $3n^2 + 4n + 7 <= 14*n^2$ for all n >= 1

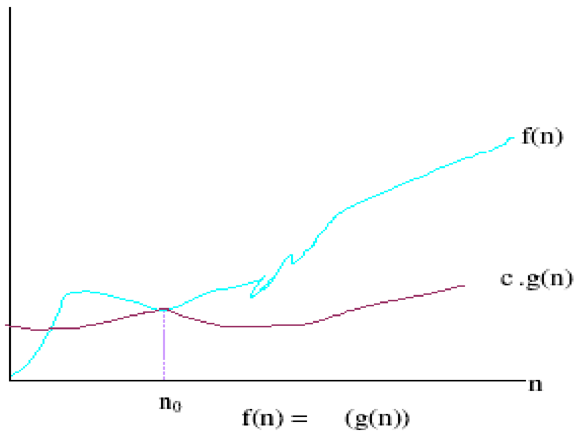
   the above inequality is trivially true

   hence f(n) = O(g(n))

Big Omega (Ω) notation

Big omega notation gives asymptotic lower bound. A function f(x) =Ω (g(x)) (read as f(x) is big omega of g(x)) iff there exists two positive constants c and x_0 such that for all x >= x_0, 0 <= c*g(x) <= f(x).

The above relation says that g(x) is a lower bound of f(x).



For all values of n >= n_0, plot shows clearly that f(n) lies above or on the curve of c*g(n).

Examples

1. $f(n) = 3n^2 + 4n + 7$

$g(n) = n^2$, then prove that $f(n) = \Omega(g(n))$.

Proof: let us choose c and n_0 values as 1 and 1, respectively then we can have

$f(n) >= c*g(n), n>=n_0$ as

$3n^2 + 4n + 7 >= 1*n^2$ for all $n >= 1$

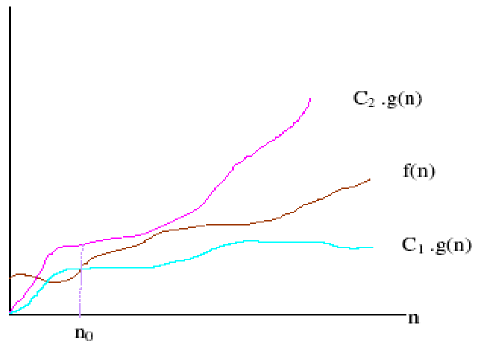
the above inequality is trivially true

hence $f(n) = \Omega(g(n))$

Big Theta (Θ) notation

When we need asymptotically tight bound then we use notation. A function $f(x) = \Theta(g(x))$ (read as f(x) is big theta of g(x)) iff there exists three positive constants c_1, c_2 and x_0 such that for all $x >= x_0$, $0 <= c_1*g(x) <= f(x) <= c_2*g(x)$

The above relation says that f(x) is order of g(x)



$f(n) = \Theta(g(n))$

For all values of $n >= n_0$, plot shows clearly that f(n) lies between $c_1*g(n)$ and $c_2*g(n)$.

Examples

1. $f(n) = 3n^2 + 4n + 7$

   $g(n) = n^2$, then prove that $f(n) = (g(n))$.

Proof: let us choose c_1, c_2 and n_0 values as 14, 1 and 1 respectively then we can have,

$f(n) <= c_1*g(n), n>=n_0$ as $3n^2 + 4n + 7 <= 14*n^2$, and

$f(n) >= c_2*g(n), n>=n_0$ as $3n^2 + 4n + 7 >= 1*n^2$

for all $n >= 1$(in both cases).

So $c_2*g(n) <= f(n) <= c_1*g(n)$ is trivial.

hence $f(n) = \Theta(g(n))$.

Example : Fibonacci Numbers

Input: *n*

Output: n_{th} Fibonacci number.

Algorithm: assume *a* as first (previous) and *b* as second(current) numbers

```
fib(n)
{
        a = 0, b= 1, f=1 ;
        for(i = 2 ; i <=n ; i++)
        {
        f = a+b ;
        a=b ;
        b=f ;
        }
        return f ;
}
```

Efficiency:
Time Complexity: The algorithm above iterates up to n-2 times, so time complexity is O(n).
Space Complexity: The space complexity is constant i.e. O(1).

Example : Bubble sort
Algorithm

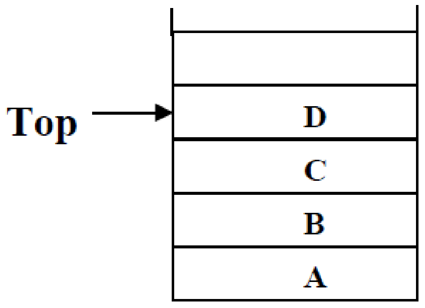
```
BubbleSort(A, n)
{
        for(i = 0; i <n-1; i++)
        for(j = 0; j < n-i-1; j++)
        if(A[j] > A[j+1])
        {
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
        }

}
```

Time Complexity:
Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:
Time complexity = (n-1) + (n-2) + (n-3) + ………………………….. +2 +1
= O(n2)

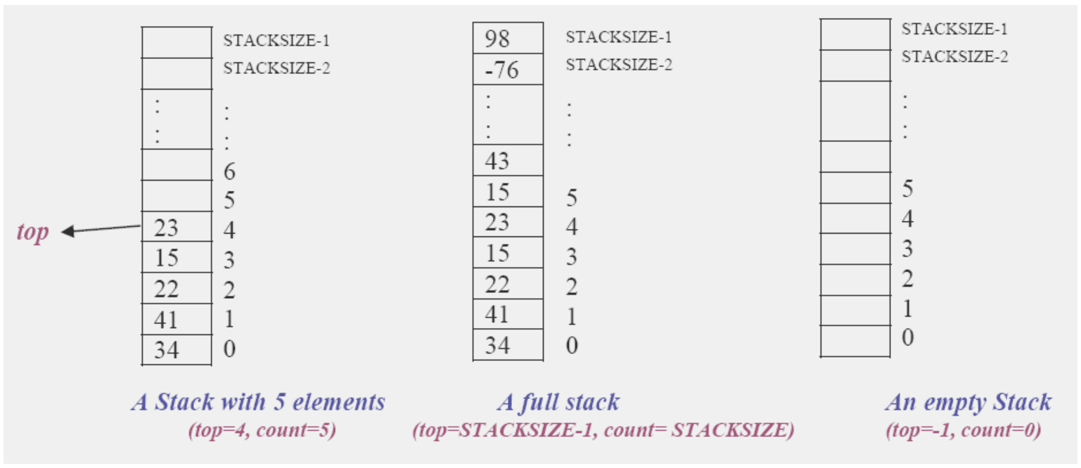
The stack

Introduction to Stack

***A stack is an ordered collection of items into which new items may be inserted and from which items may be deleted at one end, called the* top *of the stack*.** The deletion and insertion in a stack is done from top of the stack. The following fig shows the stack containing items:



(Fig: A stack containing elements or items)
Intuitively, a stack is like a pile of plates where we can only (conveniently) remove a plate from the top and can only add a new plate on the top. In computer science we commonly place numbers on a stack, or perhaps place records on the stack



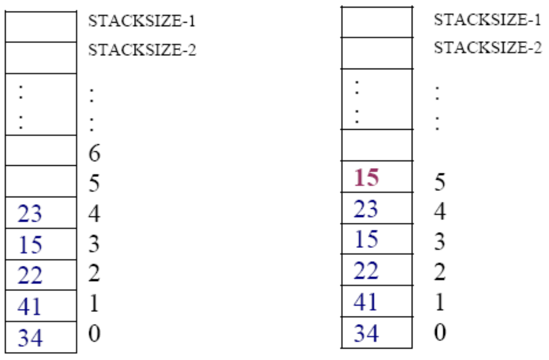
Applications of Stack:
Stack is used directly and indirectly in the following fields:

- To *evaluate the expressions (postfix, prefix)*
- To keep the p*age-visited history in a Web browser*
- To perform the u*ndo sequence in a text editor*
- Used in *recursion*
- To pass the parameters between the f*unctions in a program*
- Can be used as an a*uxiliary data structure for implementing algorithms*
- Can be used as a *component of other data structures*

Stack Operations:

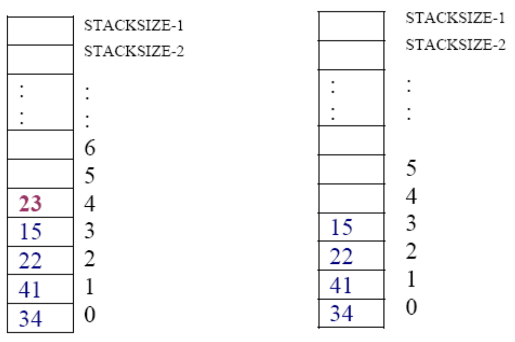
The following operations can be performed on a stack:

- ***PUSH*** operation: The push operation is used to add (or push or insert) elements in a Stack .When we add an item to a stack, we say that we ***push*** it onto the stack. The last item put into the stack is at the top



Before PUSH
(top=4, count=5)
After PUSH
(top=5, count= 6)

- ***POP*** operation: The pop operation is used to remove or delete the top element from the stack. When we remove an item, we say that we ***pop*** it from the stack. When an item is popped, it is always the top item which is removed.



Before POP
(top=4, count=5)

After POP
(top=3 count=4)

The **PUSH** and the **POP** operations are the ***basic or primitive*** operations on a stack.
Some others operations are:

- ***CreateEmptyStack*** operation: This operation is used to create an empty stack.
- ***IsFull*** operation: The isfull operation is used to check whether the stack is full or not (i.e. stack overflow)
- ***IsEmpty*** operation: The isempty operation is used to check whether the stack is empty or not. (i. e. stack underflow)
- ***Top operations:*** This operation returns the current item at the top of the stack, it doesn't remove it

The Stack ADT:

A **stack** of elements of type ***T*** is a finite sequence of elements of ***T*** together with the operations

- **CreateEmptyStack(S):** Create or make stack ***S*** be an empty stack
- **Push(S, x):** Insert ***x*** at one end of the stack, called its **top**
- **Top(S):** If stack ***S*** is not empty; then retrieve the element at its **top**
- **Pop(S):** If stack ***S*** is not empty; then delete the element at its **top**
- **IsFull(S):** Determine if ***S*** is full or not. Return **true** if ***S*** is full stack; return **false** otherwise
- **IsEmpty(S):** Determine if ***S*** is empty or not. Return **true** if ***S*** is an empty stack; return **false** otherwise.

Implementation of Stack:

Stack can be implemented in two ways:
1. Array Implementation of stack (or static implementation)
2. Linked list implementation of stack (or dynamic)

Array (static) implementation of a stack:

It is one of two ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value (an index of an array) that indicates the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack. By convention, in C implementation the empty stack is indicated by setting the value of top to -1(top=-1).

```
#define MAX 10
sruct stack
{
        int items[MAX]; //Declaring an array to store items
        int top; //Top of a stack
};
Typedef struct stack st;
```

Creating Empty Stack:

The value of top=-1 indicates the empty stack in C implementation.

```
/*Function to create an empty stack*/
void create_empty_stack(st *s)
{
        s->top=-1;
}
```

Stack Empty or Underflow:

This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates the empty. The following function return 1 if the stack is empty, 0 otherwise.

```
int isempty(st *s)
{
        if(s->top==-1)
        return 1;
        else
        return 0;
}
```

Stack Full or Overflow:

This is the situation when the stack becomes full, and no more elements can be pushed onto the stack. At this point the stack top is present at the highest location (MAXSIZE-1) of the stack. The following function returns true (1) if stack is full, false (0) otherwise.

```
int isfull(st *s)
{
        if(s->top==MAX-1)
        return 1;
        else
        return 0;
}
```

Algorithm for PUSH and POP operations on Stack

Let Stack[MAXSIZE] be an array to implement the stack. The variable top denotes the top of the stack.

i) Algorithm for PUSH (inserting an item into the stack) operation:

This algorithm adds or inserts an item at the top of the stack

```
1. [Check for stack overflow]
        if top=MAX-1 then
        print "Stack Overflow" and Exit
        else
        Set top=top+1 //[Increase top by 1]
        Set Stack[top]:= item //[Inserts item in new top position]
2. Exit
```

ii) Algorithm for POP (removing an item from the stack) operation

This algorithm deletes the top element of the stack and assigns it to a variable *item*

```
1.[Check for the stack Underflow]
        If top<0 then
        Print "Stack Underflow" and Exit
        else
        Set item=Stack [top] //[Remove the top element]
        Set top=top-1//[Decrement top by 1]
        Return the deleted item from the stack
2. Exit
```

The PUSH and POP functions

The C function for **push** operation

```
void push(st *s, int element)
```

```
{
        if(isfull(s)) /* Checking Overflow condition */
        printf("\n \n The stack is overflow: Stack Full!!\n");
        else
        s->items[++(s->top)]=element; /* First increase top by 1 and store element at top position*/
}
```

OR

Alternatively we can define the push function as give below:

```
void push()
{
        int item;
        if(top == MAXSIZE - 1) //Checking stack overflow
        printf("\n The Stack Is Full");
        else
        {
        printf("Enter the element to be inserted");
        scanf("%d",&item); //reading an item
        top= top+1; //increase top by 1
        stack[top] = item; //storing the item at the top of the stack
        }
}
```

The C function for POP operation

```
void pop(st *s)
{
        if(isempty(s))
        printf("\n\nstack Underflow: Empty Stack!!!");
        else
        printf("\nthe deleted item is %d:\t",s->items[s->top--]);/*deletes top element and decrease
top by 1 */
}
```

OR

Alternatively we can define the push function as give below:

```
void pop()
{
        int item;
        if(top <0) //Checking Stack Underflow
        printf("The stack is Empty");
        else
        {
        item = stack[top]; //Storing top element to item variable
        top = top-1; //Decrease top by 1
        printf(“The popped item is=%d”,item); //Displaying the deleted item
        }
}
```

The following complete program illustrates the implementation of stack with operations:

Program: 1

```
/* Array Implementation of Stack */
#include<stdio.h>
#include<conio.h>
#define MAX 10
struct stack
{
        int items[MAX]; //Declaring an array to store items
```

```
            int top; //Top of a stack
};
typedef struct stack st;
void create_empty_stack(st *s); //function prototype
void push(st *s, int);
void pop(st *s);
void display(st *s);
//Main Function
void main()
{
            int element, choice;
st s;
int flag=1
            create_empty_stack(&s); /* s->top=-1; indicates empty stack */
            while(flag)
            {
            printf("\n\n Enter your choice");
            printf(" \n\n\t 1:Push the elements");
            printf(" \n\n\t 2: To display the elements");
            printf(" \n\n\t 3: Pop the element");
            printf(" \n\n\t 4: Exit");
            printf("\n\n\n Enter of your choice:\t");
            scanf("%d",&choice);
switch(choice)
{
case 1:
            printf("\n Enter the number:");
            scanf("%d", &element); /*Read an element from keyboard*/
            push(&s,element);
            break;
case 2:
            display(&s);
            break;
case 3: clrscr();
            pop(&s);
            break;
case 4:
            flag=0;
            break;
default:
printf("\n Invalid Choice");
}
}
getch();
}

/*Function to create an empty stack*/
void create_empty_stack(st *s)
{
            s->top=-1;
}

/*Function to check whether the stack is empty or not */
int isempty(st *s)
{
        if(s->top==-1)
        return 1;
        else
        return 0;
}

/*function to check whether the stack is full or not*/
int isfull(st *s)
{
        if(s->top==MAX-1)
        return 1;
        else
        return 0;
}
/* push() function definition */
void push(st *s, int element)
{
        if(isfull(s)) /* Checking Overflow condition */
        printf("\n \nThe stack is overflow: Stack Full!!\n");
        else
        s->items[++(s->top)]=element;
}
/* Function for displaying elements of a stack*/
void display(st *s)
{
int i;
        if(isempty(s))
        printf("\nThe Stack does not contain any Elements");
        else
        {
        printf("\nThe elements in the stack is/are:\n");
        for(i=s->top;i>=0;i--)
        printf("%d\n",s->items[i]);
        }
}
/* the POP function definition*/
void pop(st *s)
{
        if(isempty(s))
        printf("\n\nstack Underflow: Empty Stack!!!");
        else
        printf("\n\nthe deleted item is %d:\t",s->items[s->top--]);
}
```

Recursion:

Recursion is the process of defining a problem in terms of itself but with a smaller instance. In programming it is implemented by the use of recursive functions. A recursive function is a function which calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result.

In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition.

Example:

/*calculation of the factorial of an integer number using recursive function*/

```
#include<stdio.h>
#include<conio.h>
long int factorial(int n);
void main()
{
        int n;
        long int facto;
        printf("Enter value of n:");
        scanf("%d",&n);
        facto=factorial(n);
        printf("%d! = %ld",n,facto);
        getch();
}

long int factorial(int n)
{
        if(n == 0)
        return 1;
        else
        return n * factorial(n-1);
}
```

Let's trace the evaluation of factorial(5):

Factorial(5)=
5*Factorial(4)=
5*(4*Factorial(3))=
5*(4*(3*Factorial(2)))=
5*(4*(3*(2*Factorial(1))))=
5*(4*(3*(2*(1*Factorial(0)))))=
5*(4*(3*(2*(1*1))))=
5*(4*(3*(2*1)))=
5*(4*(3*2))=
5*(4*6)=
5*24=
120

/* Program to generate Fibonacci series up to n terms using recursive function*/

```
#include<stdio.h>
#include<conio.h>
void main()
{
        int n,i;
        int fibo(int);
        printf("Enter n:");
        scanf("%d",&n);
printf("Fibonacci numbers up to %d terms:\n",n);
        for(i=1;i<=n;i++)
        printf("%d\n",fibo(i));
        getch();
}
int fibo(int k)
{
        if(k == 1 || k == 2)
        return 1;
        else
        return fibo(k-1)+fibo(k-2);
}
```

/* Program to find sum of first n natural numbers using recursion*/

```
#include<stdio.h>
#include<conio.h>
int sum_natural(int );
void main()
{
        int n;
        printf("n = ");
        scanf("%d",&n);
printf("Sum of first %d natural numbers = %d",n,sum_natural(n));
        getch();
}
int sum_natural(int n)
{
        if(n == 1)
        return 1;
        else
        return n + sum_natural(n-1);
}
```

Tower of Hanoi problem:

Initial state:

- There are three poles named as origin, intermediate and destination.
- **n** number of different-sized disks having hole at the centre are stacked around the origin pole in decreasing order.
- The disks are numbered as 1, 2, 3, 4, ……………….,n.

Objective:

- Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

Conditions:

- Move only one disk at a time.
- Each disk must always be placed around one of the pole.
- Never place larger disk on top of smaller disk.

Algorithm: - To move a tower of *n* disks from *source* to *dest* (where *n* is positive integer):

```
1. If n ===1:
1.1. Move a single disk from source to dest.
2. If n > 1:
2.1. Let temp be the remaining pole other than source and dest.
2.2. Move a tower of (n – 1) disks form source to temp.
2.3. Move a single disk from source to dest.
2.4. Move a tower of (n – 1) disks form temp to dest.
3. Terminate.
```

Example: Recursive solution of tower of Hanoi:

```
#include <stdio.h>
#include <conio.h>
void TOH(int, char, char, char); //Function prototype
void main()
{
        int n;
        printf("Enter number of disks");
        scanf("%d",&n);
        TOH(n,'O','D','I');
        getch();
}
void TOH(int n, char A, char B, char C)
{
if(n>0)
{
        TOH(n-1, A, C, B);
        Printf("Move disk %d from %c to%c\n", n, A, B);
        TOH(n-1, C, B,A );
}
}
```

/* Program to find multiplication of first n natural numbers using recursion*/

```
#include<stdio.h>
#include<conio.h>
int mul_natural(int );
void main()
{
int n;
        printf("n = ");
        scanf("%d",&n);
        printf("Product of first %d natural numbers = %d", n, mul_natural(n));
        getch();
}


int mul_natural(int n)
{
if(n == 1)
        return 1;
else
        return (n * mul_natural(n-1));
}
```

Advantages of Recursion:

- The code may be much easier to write.
- To solve some problems which are naturally recursive such as Tower of Hanoi.

Disadvantages of Recursion:

- Recursive functions are generally slower than non-recursive functions.
- May require a lot of memory to hold intermediate results on the system stack.
- It is difficult to think recursively so one must be very careful when writing recursive functions.

Infix, Prefix and Postfix Notation

One of the applications of the stack is to evaluate the expression. We can represent the expression following three types of notation:

- Infix
- Prefix
- Postfix
- **Infix expression:** It is an ordinary mathematical notation of expression where operator is written in between the operands. Example: A+B. Here '+' is an ***operator*** and ***A*** and ***B*** are called ***operands***
- **Prefix notation:** In prefix notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notation. Example: +AB
- **Postfix notation**: In postfix notation the operators are written after the operands so it is called the postfix notation (post mean after). In this notation the operator follows the two operands. Example: AB+
- Both prefix and postfix are parenthesis free expressions. For example

(A + B) * C Infix form
* + A B C Prefix form
A B + C * Postfix form

Infix	Postfix	Prefix
A+B	AB+	+AB
A+B-C	AB+C-	-+ABC
(A+B)*(C-D)	AB+CD-*	*+AB-CD

Converting an Infix Expression to Postfix

First convert the sub-expression to postfix that is to be evaluated first and repeat this process. You substitute intermediate postfix sub-expression by any variable whenever necessary that makes it easy to convert.

- Remember, to convert an infix expression to its postfix equivalent, we first convert the innermost parenthesis to postfix, resulting as a new operand.
- In this fashion parenthesis can be successively eliminated until the entire expression is converted
- The last pair of parenthesis to be opened within a group of parenthesis encloses the first expression within the group to be transformed. This last in, first-out behaviour suggests the use of a stack

Precedence rule:

While converting infix to postfix you have to consider the **precedence rule**, and the precedence rules are as follows

1. Exponentiation (the expression A$B is A raised to the B power, so that 3$2=9)
2. Multiplication/Division
3. Addition/Subtraction

When un-parenthesized operators of the same precedence are scanned, the order is assumed to be left to right except in the case of exponentiation, where the order is assumed to be from right to left.

- A+B+C means (A+B)+C
- A$B$C means A$(B$C)

By using parenthesis we can override the default precedence.

Consider an example that illustrate the converting of infix to postfix expression, A + (B* C).

Use the following **rule** to convert it in postfix:

1. Parenthesis for emphasis

2. Convert the multiplication
3. Convert the addition
4. Post-fix form

Illustration:

A + (B * C). Infix form
A + (B * C) Parenthesis for emphasis
A + (BC*) Convert the multiplication
A (BC*) + Convert the addition
ABC*+ Post-fix form

Consider an example:

(A + B) * ((C - D) + E) / F Infix form
(AB+) * ((C – D) + E) / F
(AB+) * ((CD-) + E) / F
(AB+) * (CD-E+) / F
(AB+CD-E+*) / F
AB+CD-E+*F/ Postfix form

Examples

Infix	*Postfix*
A + B	AB +
A + B – C	AB + C -
(A + B) * (C – D)	AB + CD - *
A $ B * C – D + E / F / (G + H)	AB $ C * D – EF / GH + / +
((A + B) * C – (D - E)) $ (F + G)	AB + C * DE - - FG + $
A – B / (C * D $ E)	ABCDE $ * / -

Exercise: Convert the infix expression listed in the above table into postfix notation and verify yourself.

Algorithm to convert infix to postfix notation

Let *opstack* be the stack to store operators, *poststring is the* string used to store final postfix expression and *otos* represents the *opstack* top.

```
1. Scan one character at a time of an infix expression from left to right
2. opstack=the empty stack
3. Repeat till there is data in infix expression
        3.1 if scanned character is '(' then push it to opstack
        3.2 if scanned character is operand then append it to poststring
        3.3 if scanned character is operator then
        if(otos!=-1)
while(precedence (opstack[otos])>=precedence(scan character)) then pop from opstack and append it to poststring
        otherwise
        push into opstack
        3.4 if scanned character is ')' then
        pop from opstack and append to poststring until '(' is not found and ignore both symbols
4. pop from opstack and append to poststring until opstack is not empty.
5. print poststring
```

6. return

Trace of Conversion Algorithm

The following tracing of the algorithm illustrates the algorithm. Consider an infix expression

((A-(B+C))*D)$(E+F)

Scan character	Poststack	opstack
(	……	(
(	……	((
A	A	((
-	A	((-
(	A	((-(
B	AB	((-(
+	AB	((-(+
C	ABC	((-(+
)	ABC+	((-
)	ABC+-	(
*	ABC+-	(*
D	ABC+-D	(*
)	ABC+-D*	…….
$	ABC+-D*	$
(	ABC+-D*	$(
E	ABC+-D*E	$(
+	ABC+-D*E	$(+
F	ABC+-D*EF	$(+
)	ABC+-D*EF+	$
……	ABC+-D*EF+$ (postfix)	…………….

Converting an Infix expression into prefix expression

The precedence rule for converting from an expression from infix to prefix is identical. Only changes from postfix conversion is that the operator is placed before the operands rather than after them. The prefix of

A+B-C is -+ABC.

A+B-C (infix)

=(+AB)-C

=-+ABC (prefix)
Example Consider an example:
A $ B * C – D + E / F / (G + H) infix form
= A $ B * C – D + E / F /(+GH)
=$AB* C – D + E / F /(+GH)
=*$ABC-D+E/F/(+GH)
=*$ABC-D+(/EF)/(+GH)
=*$ABC-D+//EF+GH
= (-*$ABCD) + (//EF+GH)
=+-*$ABCD//EF+GH which is in prefix form.

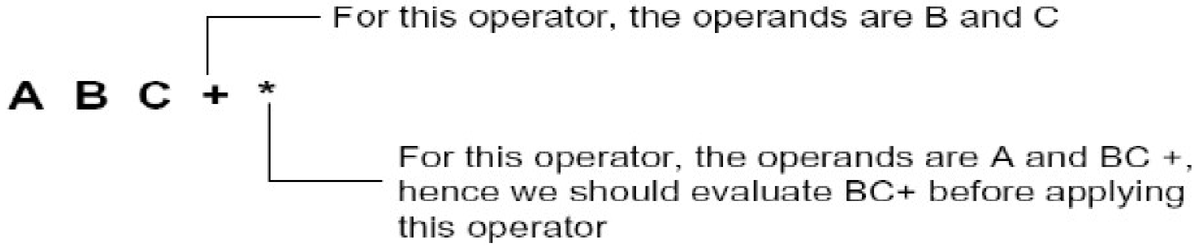
Algorithm to convert Infix expression into Prefix

Let two stacks opstack and prestack are used and otos & ptos represents the opstack top and prestack top respectively.

```
1. Scan one character at a time of an infix expression from right to left.
2. opstack=the empty stack
3. Repeat till there is data in infix expression
        3.1 if scanned character is ')' then push it to opstack
        3.2 if scanned character is operand then push it to prestack
        3.3 if scanned character is operator then
        if(otos!=-1)
        while(precedence (opstack[otos])>precedence(scan character)) then pop from opstackand push it into prestack
        otherwise
        push into opstack
        3.4 if scanned character is '(' then
        pop from opstack and push into prestack until ')' is not found and ignore both symbols
4. pop from opstack and push into prestack until opstack is not empty.
5. Pop and print one character at a time from prestack
5. return
```

Evaluating the Postfix expression

Each operator in a postfix expression refers to the previous two operands in the expression.


For this operator, the operands are B and C
A B C + *
For this operator, the operands are A and BC +, hence we should evaluate BC+ before applying this operator

To evaluate the postfix expression we use the following procedure:
Each time we read an operand we push it onto a stack. When we reach an operator, its operand s will be the top two elements on the stack. We can then pop these two elements perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.

Consider an example
3 4 5 * +
=3 20 +
=23 (answer)

Evaluating the given postfix expression:

6 2 3 + - 3 8 2 / + * 2 $ 3 +
=6 5 - 3 8 2 / + * 2 $ 3 +
=1 3 8 2 / + * 2 $ 3 +
=1 3 4 + * 2 $ 3 +
=1 7 * 2 $ 3 +
=7 2 $ 3 +
=49 3 +
= 52

Algorithm to evaluate the postfix expression

Here we use only one stack called vstack(value stack).

```
1. Scan one character at a time from left to right of given postfix expression until there is data
        1.1 if scanned symbol is operand then
        read its corresponding value and push it into vstack
        1.2 if scanned symbol is operator then
        – pop and place into op2
        – pop and place into op1
        – compute result according to given operator as 'op1 operator  op2' and push result into vstack
2. pop and display the result
3. return
```

Trace of Evaluation:

Consider an example to evaluate the postfix expression tracing the algorithm
ABC+*CBA-+*
123+*321-+*

Scanned character	value	Op2	Op1	Result	vstack
A	1	……	…….	…….	1
B	2	……	……	…..	1 2
C	3	……	……	…..	1 2 3
+	…….	3	2	5	1 5
*	……	5	1	5	5
C	3	…….	……	……	5 3
B	2	…	…….		5 3 2
A	1	…….	…….	……	5 3 2 1
-	…….	1	2	1	5 3 1
+	…….	1	3	4	5 4
*	…….	4	5	20	20

Its final value is 20.

Algorithm to evaluate the prefix expression

Here we use only one stack called vstack(value stack).

```
1. Scan one character at a time from left to right of given prefix expression until there is data
    1.1 if scanned symbol is operand then
        read its corresponding value and push it into vstack
    1.2 if scanned symbol is operator then
        – pop and place into op1
        – pop and place into op2
        – compute result according to given operator as 'op1 operator  op2' and push result into vstack
2. pop and display the result
3. return
```

Illustration: Evaluate the given prefix expression

/ + 5 3 – 4 2 prefix equivalent to (5+3)/(4-2) infix notation
= / 8 – 4 2
= / 8 2
= 4

/*program for evaluating postfix expression*/

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
void push(int);
int pop();
int vstack[100];
int tos=-1;
void main()
{
int i,res,l,op1,op2,value[100];
char postfix[100],ch;
clrscr();
printf("Enter a valid postfix\n");
gets(postfix);
l=strlen(postfix);
for(i=0;i<=l-1;i++)
{
if(isalpha(postfix[i]))
{
printf("Enter value of %c",postfix[i]);
scanf("%d",&value[i]);
push(value[i]);
}e
lse
{
ch=postfix[i];
op2=pop();
op1=pop();
switch(ch)
{
case '+':
push(op1+op2);
break;
case'-':
push(op1-op2);
break;
case'*':
push(op1*op2);
break;
case'/':
push(op1/op2);
break;
case'$':
push(pow(op1,op2));
break;
case'%':
push(op1%op2);
break;
}
}
}
printf("The reault is:");
res=pop();
printf("%d", res);

getch();
}
/***********insertion function*************/
void push(int val)
{
vstack[++tos]=val;
}
/***********deletion function***************/
int pop()
{
int n;
n=vstack[tos--];
return(n);
}
```

/*program to convert infix to postfix expression*/

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
int precedency(char);
void main()
{
int i,otos=-1,ptos=-1,l, l1;
char infix[100],poststack[100],opstack[100];
printf("Enter a valid infix\n");
gets(infix);
l=strlen(infix);
l1=l;
for(i=0;i<=l-1;i++)
{
if(infix[i]=='(')
{
```

```
opstack[++otos]=infix[i];
l1++;
}e
lse if(isalpha(infix[i]))
{
poststack[++ptos]=infix[i];
}e
lse if (infix[i]==')')
{
l1++;
while(opstack[otos]!='(')
{
poststack[++ptos]=opstack[otos];
otos--;
}
otos--;
}e
lse //operators
{
if(precedency(opstack[otos])>precedency(infix[i]))
{
poststack[++ptos]=opstack[otos--];
}
opstack[++otos]=infix[i];
}
}
while(otos!=-1)
{
poststack[++ptos]=opstack[otos];
otos--;
}
/********for displaying***************/
for(i=0;i<l1;i++)
{
printf("%c",poststack[i]);
}
getch();
}
/****************precedency function*********************/
int precedency(char ch)
{
switch(ch)
{
case '$':
return(4);
// break;
case'*':
case'/':
return(3);
// break;
case'+':
case'-':
return(2);
// break;
default:
```

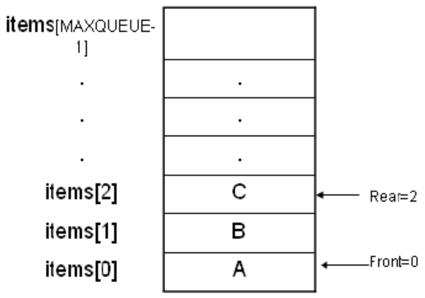
```
return(1);
}
}
```

Queue

What is a queue?

A **Queue** is an ordered collection of items from which items may be deleted at one end (called the ***front*** of the queue) and into which items may be inserted at the other end (the ***rear*** of the queue). The first element inserted into the queue is the first element to be removed. For this reason a queue is sometimes called a ***fifo*** (first-in first-out) list as opposed to the stack, which is a ***lifo*** (last-in first-out).

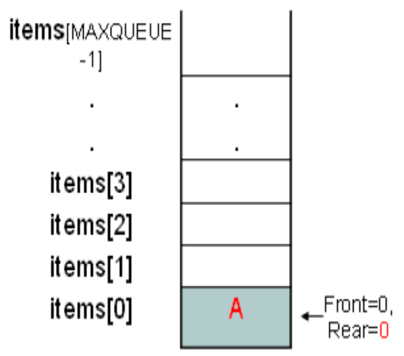
Example:



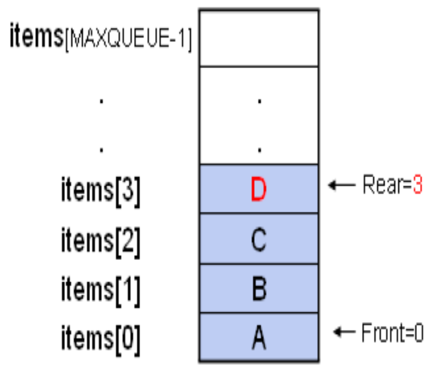
Operations on queue:

- ***MakeEmpty(q):*** To make q as an empty queue
- ***Enqueue(q, x):*** To insert an item x at the rear of the queue, this is also called by names add, insert.
- ***Dequeue(q):*** To delete an item from the front of the queue q. this is also known as Delete, Remove.
- ***IsFull(q):*** To check whether the queue q is full.
- ***IsEmpty(q):*** To check whether the queue q is empty
- ***Traverse (q):*** To read entire queue that is display the content of the queue.

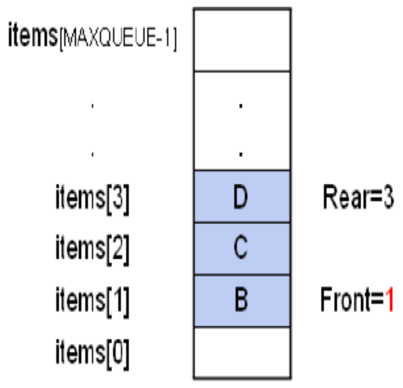
Enqueue(A):



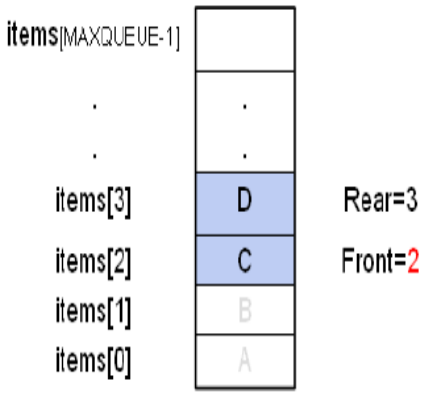
Enqueue(B,C,D):



Dequeue(A):



Dequeue(B):



Initialization of queue:

The queue is initialized by having the *rear* set to *-1*, and *front* set to *0*. Let us assume that maximum number of the element we have in a queue is MAXQUEUE elements

Applications of queue:

- Task waiting for the printing
- Time sharing system for use of CPU
- For access to disk storage
- Task scheduling in operating system

The Queue as an ADT:

A queue q of type T is a finite sequence of elements with the operations

- ***MakeEmpty(q):*** To make q as an empty queue
- ***IsEmpty(q):*** To check whether the queue q is empty. Return true if q is empty, return false otherwise.
- ***IsFull(q):*** To check whether the queue q is full. Return true in q is full, return false otherwise.
- ***Enqueue(q, x):*** To insert an item x at the rear of the queue, if and only if q is not full.
- ***Dequeue(q):*** To delete an item from the front of the queue q. if and only if q is not empty.
- ***Traverse (q):*** To read entire queue that is display the content of the queue.

Implementation of queue:

There are two techniques for implementing the queue:

- Array implementation of queue(static memory allocation)
- Linked list implementation of queue(dynamic memory allocation)

Array implementation of queue:

In array implementation of queue, an array is used to store the data elements. Array implementation is also further classified into two types

- ***Linear array implementation:***

A linear array with two indices always increasing that is rear and front. Linear array implementation is also called linear queue

- ***Circular array implementation:***

This is also called circular queue.

Linear queue:

Algorithm for insertion (or Enqueue) and deletion (Dequeue) in queue:

Algorithm for insertion an item in queue:

```
1. if rear>=MAXSIZE-1
        print "queue overflow" and return
else
        set rear=rear+1
        queue[rear]=item
2. end
```

Algorithm to delete an element from the queue:

```
1. if rear<front
        print "queue is empty" and return
else
        item=queue[front++]
2. end
```

Declaration of a Queue:

```
# define MAXQUEUE 50 /* size of the queue items*/
struct queue
{
int front;
int rear;
int items[MAXQUEUE];
};
typedef struct queue qt;
```

Defining the operations of linear queue:

- ***The MakeEmpty function:***

```
void makeEmpty(qt *q)
{
q->rear=-1;
q->front=0;
}
```

- ***The IsEmpty function:***

```
int IsEmpty(qt *q)
{
if(q->rear<q->front)
return 1;
else
return 0;
}
```

- ***The Isfull function:***

```
int IsFull(qt *q) {
if(q->rear==MAXQUEUEZIZE-1)
return 1;
else
return 0;
}
```

- ***The Enqueue function:***

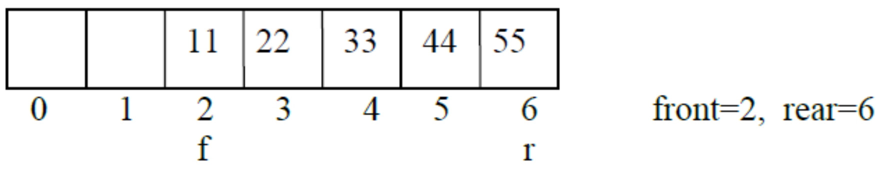
```
void Enqueue(qt *q, int newitem)
{
if(IsFull(q))
{
printf(“queue is full”);
}
else
{
q->rear++;
q->items[q->rear]=newitem;
}
}
```

- ***The Dequeue function:***

```
int Dequeue(qt *q)
{
if(IsEmpty(q))
{
printf(“queue is Empty”);
}
else
{
return(q->items[q->front++]);
}
}
```

Problems with Linear queue implementation:

- Both rear and front indices are increased but never decreased.
- As items are removed from the queue, the storage space at the beginning of the array is discarded and never used again. Wastage of the space is the main problem with linear queue which is illustrated by the following example. This queue is considered full, even though the



space at beginning is vacant.

Circular queue:

A circular queue is one in which the insertion of a new element is done at very first location of the queue if the last location of the queue is full.

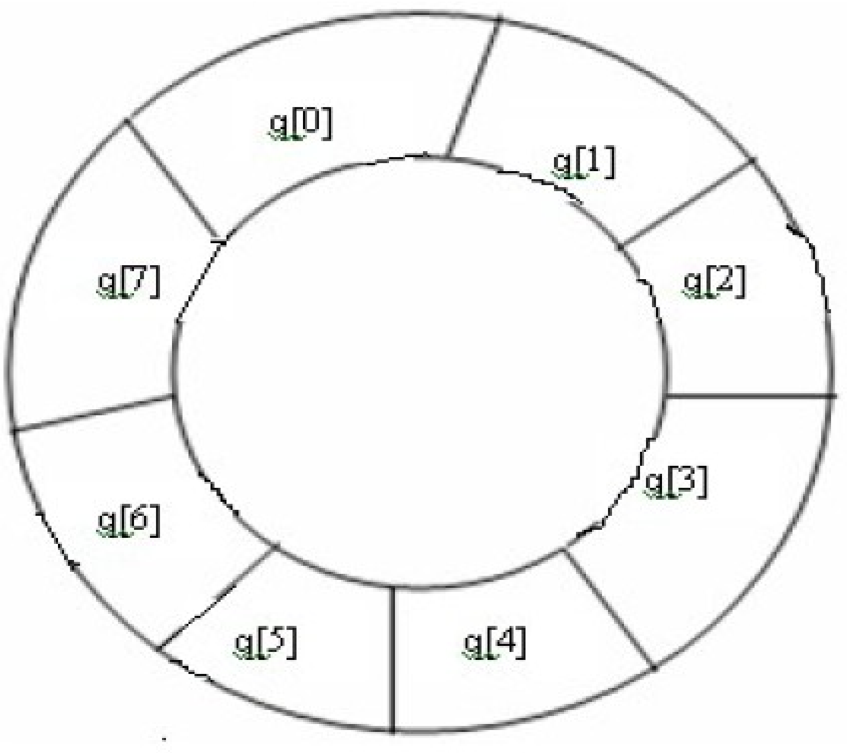


Fig:- Circular queue

A circular queue overcomes the problem of unutilized space in linear queue implementation as array. In circular queue we sacrifice one space of the array thus to insert n elements in a circular queue we need an array of size n+1.(or we can insert one less than the size of the array in circular queue).

Initialization of Circular queue:

rear= MAXSIZE-1
front=0

Algorithms for inserting an element in a circular queue:

This algorithm assumes that rear and front are initially set to MAZSIZE-1.

```
1. if (front==(rear+2)%MAXSIZE)
        print Queue is full and exit
else
        rear=(rear+1)%MAXSIZE; [increment rear by 1]
2. cqueue[rear]=item;
3. end
```

Algorithms for deleting an element from a circular queue:

```
1. if (front==(rear+1)%MAXSIZE) [checking empty condition]
        print Queue is empty and exit
2. item=cqueue[front];
3. front=(front+1)%MAXSIZE; [increment front by 1]
4. return item;
5. end.
```

Declaration of a Circular Queue:

```
# define MAXSIZE 50 /* size of the circular queue items*/
struct cqueue
{
int front;
int rear;
int items[MAXSIZE];
};
typedef struct cqueue cq;
```

Operations of a circular queue:

The MakeEmpty function:

```
void makeEmpty(cq *q)
{
q->rear=MAXSIZE-1;
q->front=MAXSIZE-1;
}
```

The IsEmpty function:

```
int IsEmpty(cq *q)
{
if(q->rear<q->front)
return 1;
else
return 0;
}
```

The Isfull function:

```
int IsFull(cq *q)
{
if(q->front==(q->rear+1)%MAXDIZE)
return 1;
else
return 0;
}
```

The Enqueue function:

```
void Enqueue(cq *q, int newitem)
{
if(IsFull(q))
{
printf("queue is full");
exit(1);
}
else
{
q->rear=(q->rear+1)%MAXDIZE;
q->items[q->rear]=newitem;
}
}
```

The Dequeue function:

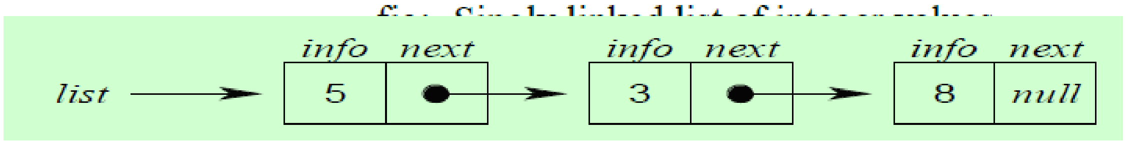
```
int Dequeue(cq *q)
{
if(IsEmpty(q))
{
printf("queue is Empty");
exit(1);
}
else
{
q->front=(q->front+1)%MAXSIZE;
return(q->items[q->front]);
}
}
```

Linked List:

A linked list is a collection of nodes, where each node consists of two parts:
info: the actual element to be stored in the list. It is also called data field.
link: one or two links that points to next and previous node in the list. It is also called next or pointer field.
The nodes in a linked list are not stored contiguously in the memory
You don't have to shift any element in the list.
Memory for each node can be allocated dynamically whenever the need arises.
The size of a linked list can grow or shrink dynamically

Illustration:

fig: Singly linked list of integer values



Operations on linked list:

The basic operations to be performed on the linked list are as follows:
Creation: This operation is used to create a linked list
Insertion: This operation is used to insert a new node in a linked list in a specified position. A new node may be inserted
✔At the beginning of the linked list
✔At the end of the linked list
✔At the specified position in a linked list
Deletion: The deletion operation is used to delete a node from the linked list. A node may be deleted from
✔The beginning of the linked list
✔the end of the linked list
✔ the specified position in the linked list.
Traversing: The list traversing is a process of going through all the nodes of the linked list from on end to the other end. The traversing may be either forward or backward.
Searching or find: This operation is used to find an element in a linked list. In the desired element is found then we say operation is successful otherwise unsuccessful.
Concatenation: It is the process of appending second list to the end of the first list.

Types of Linked List:

Basically we can put linked list into the following four types:

- Singly linked list
- doubly linked list

- circular linked list
- circular doubly linked list

Singly linked list:

A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made. In this type of linked list each node contains two fields one is info field which is used to store the data items and another is link field that is used to point the next node in the list. The last node has a NULL pointer.

Representation of singly linked list:

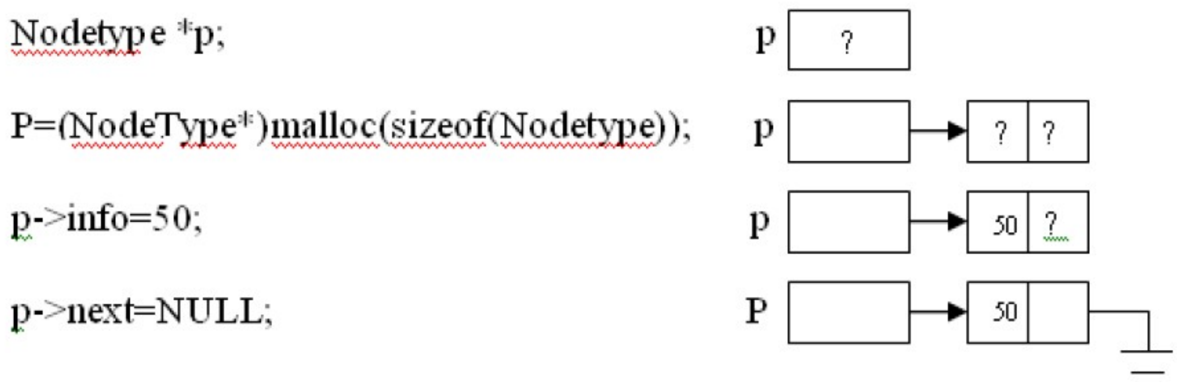
We can create a structure for the singly linked list the each node has two members, one is **info** that is used to store the data items and another is **next** field that store the address of next node in the list. We can define a node as follows:

```
struct Node
{
int info;
struct Node *next;
};
typedef struct Node NodeType;

NodeType *head; //head is a pointer type structure variable
```

This type of structure is called self-referential structure.

Creating a Node:

To create a new node, we use the **malloc** function to dynamically allocate memory for the new node. After creating the node, we can store the new item in the node using a pointer to that node. The following figure clearly shows the steps required to create a node and storing an item.



```
Nodetype *p;
P=(NodeType*)malloc(sizeof(Nodetype));
p->info=50;
p->next=NULL;
```

Note that p is not a node; instead it is a pointer to a node.

The getNode function:

we can define a function getNode() to allocate the memory for a node dynamically. It is user-defined function that returns a pointer to the newly created node.

```
Nodetype *getNode()
{
NodeType *p;
p==(NodeType*)malloc(sizeof(NodeType));
return(p);
}
```

Creating the empty list:

```
void createEmptyList(NodeType *head)
{
head=NULL;
}
```

Inserting Nodes:

To insert an element or a node in a linked list, the following three things to be done:
Allocating a node
Assigning a data to info field of the node
Adjusting a pointer

and a new node may be inserted
At the beginning of the linked list
At the end of the linked list
At the specified position in a linked list
Insertion requires obtaining a new node and changing two links

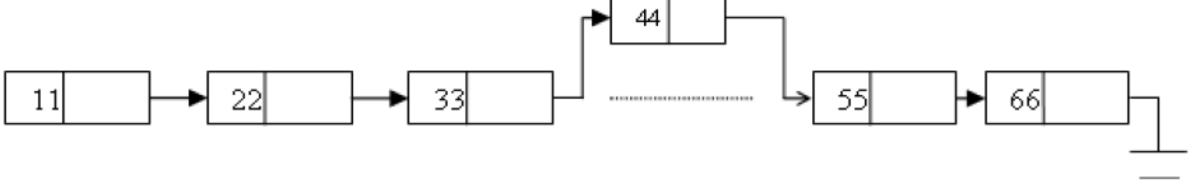


fig:- Inserting the new node with 44 between 33 and 55.

An algorithm to insert a node at the beginning of the singly linked list:

let *head be the pointer to first node in the current list

```
1. Create a new node using malloc function
        NewNode=(NodeType*)malloc(sizeof(NodeType));
2. Assign data to the info field of new node
        NewNode->info=newItem;
3. Set next of new node to head
        NewNode->next=head;
4. Set the head pointer to the new node
        head=NewNode;
5. End
```

An algorithm to insert a node at the end of the singly linked list:

let *head be the pointer to first node in the current list

```
1. Create a new node using malloc function
        NewNode=(NodeType*)malloc(sizeof(NodeType));
2. Assign data to the info field of new node
        NewNode->info=newItem;
3. Set next of new node to NULL
        NewNode->next=NULL;
4. if (head ==NULL)then
        Set head =NewNode.and exit.
5. Set temp=head;
6 while(temp->next!=NULL)
        temp=temp->next; //increment temp
7. Set temp->next=NewNode;
8. End
```

An algorithm to insert a node after the given node in singly linked list:

let *head be the pointer to first node in the current list and *p be the pointer to the node after which we want to insert a new node.

```
1. Create a new node using malloc function
        NewNode=(NodeType*)malloc(sizeof(NodeType));
2. Assign data to the info field of new node
        NewNode->info=newItem;
3. Set next of new node to next of p
        NewNode->next=p->next;
4. Set next of p to NewNode
        p->next =NewNode..
5. End
```

An algorithm to insert a node at the specified position in a singly linked list:

let *head be the pointer to first node in the current list

```
1. Create a new node using malloc function
NewNode=(NodeType*)malloc(sizeof(NodeType));
2. Assign data to the info field of new node
        NewNode->info=newItem;
3. Enter position of a node at which you want to insert a new node. Let this position is pos.
4. Set temp=head;
5. if (head ==NULL)then
        printf("void insertion"); and exit(1).
6. for(i=1; i<pos-1; i++)
        temp=temp->next;
7. Set NewNode->next=temp->next;
        set temp->next =NewNode..
8. End
```

Deleting Nodes:

A node may be deleted:

From the beginning of the linked list

From the end of the linked list

From the specified position in a linked list

Deleting first node of the linked list:

An algorithm to deleting the first node of the singly linked list:

let *head be the pointer to first node in the current list

```
1. If(head==NULL) then
        print "Void deletion" and exit
2. Store the address of first node in a temporary variable temp.
        temp=head;
3. Set head to next of head.
        head=head->next;
4. Free the memory reserved by temp variable.
        free(temp);
5. End
```

An algorithm to deleting the last node of the singly linked list:

let *head be the pointer to first node in the current list

```
1. If(head==NULL) then //if list is empty
        print "Void deletion" and exit
2. else if(head->next==NULL) then //if list has only one node
        Set temp=head;
print deleted item as,
        printf("%d" ,head->info);
        head=NULL;
        free(temp);
3. else
        set temp=head;
        while(temp->next->next!=NULL)
        set temp=temp->next;
        End of while
        free(temp->next);
        Set temp->next=NULL;
4. End
```

An algorithm to delete a node after the given node in singly linked list:
let *head be the pointer to first node in the current list and *p be the pointer to the node after which we want to delete a new node.

```
1. if(p==NULL or p->next==NULL) then
        print "deletion not possible and exit
2. set q=p->next
3. Set p->next=q->next;
4. free(q)
5. End
```

An algorithm to delete a node at the specified position in a singly linked list:
let *head be the pointer to first node in the current list

```
1. Read position of a node which to be deleted, let it be pos.
2. if head==NULL
        print "void deletion" and exit
3. Enter position of a node at which you want to delete a new node. Let this position is pos.
4. Set temp=head
declare a pointer of a structure let it be *p
5. if (head ==NULL)then
        print "void ideletion" and exit
otherwise;.
        6. for(i=1; i<pos-1; i++)
        temp=temp->next;
7. print deleted item is temp->next->info
8. Set p=temp->next;
9. Set temp->next =temp->next->next;
10. free(p);
11. End
```

Linked list implementation of Stack:
Push function:

let *top be the top of the stack or pointer to the first node of the list.

```
void push(item)
{
NodeType *nnode;
int data;
nnode=( NodeType *)malloc(sizeof(NodeType));
if(top==0)
{
nnode->info=item;
nnode->next=NULL;
top=nnode;
}
else
{
nnode->info=item;
nnode->next=top;
top=nnode;
}
}
```

Pop function:

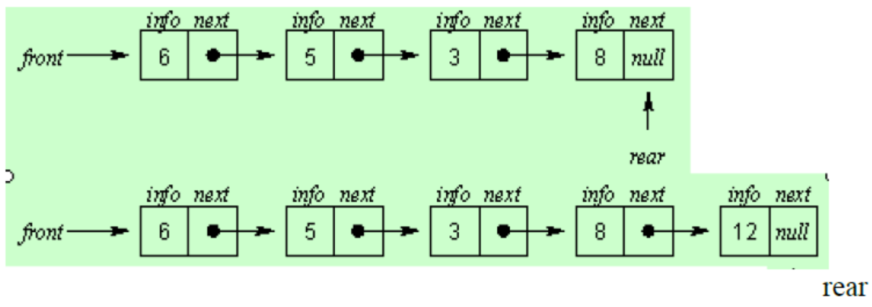
let *top be the top of the stack or pointer to the first node of the list.

```
void pop()
{
NodeType *temp;
if(top==0)
{
printf("Stack contain no elements:\n");
return;
}
else
{
temp=top;
top=top->next;
printf("\ndeleted item is %d\t",temp->info);
free(temp);
}
}
```

Linked list implementation of queue:

Insert function:

Let *rear and *front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.



```
void insert(int item)
{
```

```
NodeType *nnode;
nnode=( NodeType *)malloc(sizeof(NodeType));
if(rear==0)
{
nnode->info=item;
nnode->next=NULL;
rear=front=nnode;
}
else
{
nnode->info=item;
nnode->next=NULL;
rear->next=nnode;
rear=nnode;
}
}
```

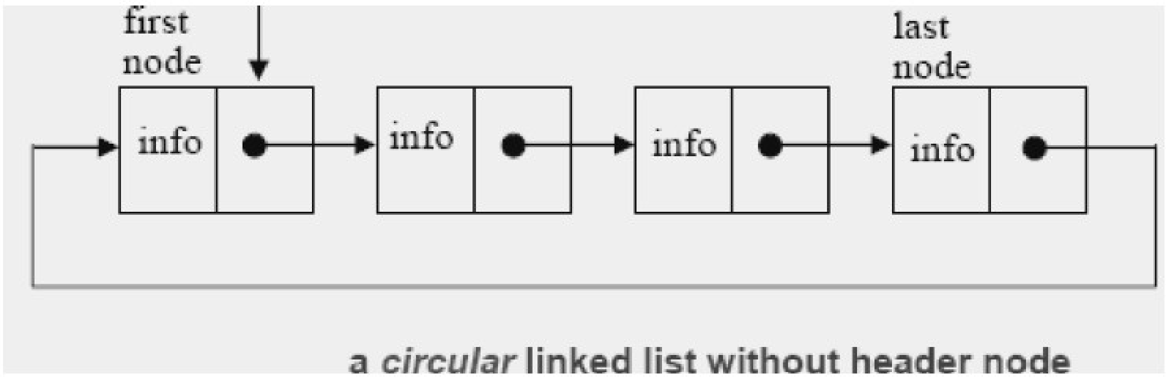
Delete function:

let *rear and *front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.

```
void delet()
{
NodeType *temp;
if(front==0)
{
printf("Queue contain no elements:\n");
return;
}
else if(front->next==NULL)
{
temp=front;
rear=front=NULL;
printf("\nDeleted item is %d\n",temp->info);
free(temp);
}
else
{
temp=front;
front=front->next;
printf("\nDeleted item is %d\n",temp->info);
free(temp);
}
}
```

Circular Linked list:

A circular linked list is a list where the link field of last node points to the very first node of the list. Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node.



a *circular* linked list without header node

In a circular linked list there are two methods to know if a node is the first node or not either a external pointer, ***list***, points the first node or

A ***header node*** is placed as the first node of the circular list. The header node can be separated from the others by either heaving a ***sentinel value*** as the info part or having a dedicated ***flag*** variable to specify if the node is a header node or not.

C representation of circular linked list:

We declare the structure for the circular linked list in the same way as declared it for the linear linked list.

```
struct node
{
int info;
struct node *next;
};
typedef struct node NodeType;
NodeType *start=NULL
NodeType *last=NULL
```

Algorithms to insert a node in a circular linked list:

Algorithm to insert a node at the beginning of a circular linked list:

```
1. Create a new node as
        newnode=(NodeType*)malloc(sizeof(NodeType));
2. if start==NULL then
        set newnode->info=item
        set newnode->next=newnode
        set start=newnode
        set last newnode
        end if
3. else
        set newnode->info=item
        set newnode->next=start
        set start=newnode
        set last->next=newnode
        end else
4. End
```

Algorithm to insert a node at the end of a circular linked list:

```
1. Create a new node as
        newnode=(NodeType*)malloc(sizeof(NodeType));
2. if start==NULL then
        set newnode->info=item
        set newnode->next=newnode
set start=newnode
        set last newnode
        end if
3. else
        set newnode->info=item
        set last->next=newnode
        set last=newnode
        set last->next=start
        end else
4. End
```

Algorithms to delete a node from a circular linked list:

Algorithm to delete a node from the beginning of a circular linked list:

```
1. if start==NULL then
        PRINT "empty list" and exit
2. else
        set temp=start
        set start=start->next
        print the deleted element=temp->info
        set last->next=start;
        free(temp)
        end else
3. End
```

Algorithm to delete a node from the end of a circular linked list:

```
1. if start==NULL then
        Print "empty list" and exit
2. else if start==last
        set temp=start
        print deleted element=temp->info
        free(temp)
        start=last=NULL
3. else
        set temp=start
        while( temp->next!=last)
        set temp=temp->next
        end while
        set hold=temp->next
        set last=temp
        set last->next=start
        print the deleted element=hold->info
        free(hold)
        end else
4. End
```

Stack as a circular List:

To implement a stack in a circular linked list, let pstack be a pointer to the last node of a circular list. Actually there is no any end of a list but for convention let us assume that the first node(rightmost node of a list) is the top of the stack. An empty stack is represented by a null list.

The structure for the circular linked list implementation of stack is:

```
struct node
{
int info;
struct node *next;
};
```

typedef struct node NodeType;
NodeType *pstack=NULL;

C function to check whether the list is empty or not as follows:

```
int IsEmpty()
{
if(pstack==NULL)
return(1);
else
return(0);
}
```

PUSH function:

```
void PUSH(int item)
{
NodeType newnode;
newnode=(NodeType*)malloc(sizeof(NodeType));
newnode->info=item;
if(pstack==NULL)
{
pstack=newnode;
pstack->next=pstack;
}
else
{
newnode->next=pstack->next;
pstack->next=newnode;
}
}
```

POP function:

```
void POP()
{
NodeType *temp;
if(pstack==NULL)
{
printf("Stack underflow\n');
exit(1);
}
else if(pstack->next==pstack) //for only one node
{
printf("poped item=%d", pstack->info);
pstack=NULL;
}
else
{
temp=pstack->next;
pstack->next=temp->next;
printf("poped item=%d", temp->info);
free(temp);
}
}
```

Queue as a circular List:

It is easier to represent a queue as a circular list than as a linear list. As a linear list a queue is specified by two pointers, one to the front of the list and the other to its rear. However, by using a circular list, a queue may be specified by a single pointer q to that list. node(q) is the rear of the queue and the following node is its front.

Insertion function:

```
void insert(int item)
{
NodeType *nnode;
nnode=( NodeType *)malloc(sizeof(NodeType));
nnode->info=item;
if(pq==NULL)
pq=nnode;
else
{
nnode->next=pq->next;
pq->next=nnode;
pq=nnode;
}
}
```

Deletion *function:*

```
void delet(int item)
{
NodeType *temp;
if(pq==NULL)
{
printf("void deletion\n");
exit(1);
}
else if(pq->next==pq) //for only one node
{
printf("poped item=%d", pq->info);
pq=NULL;
}
else
{
temp=pq->next;
pq->next=temp->next;
printf("poped item=%d", temp->info);
free(temp);
}
}
```

Doubly Linked List:

A linked list in which all nodes are linked together by multiple numbers of links i.e. each node contains three fields (two pointer fields and one data field) rather than two fields is called doubly linked list. It provides bidirectional traversal.

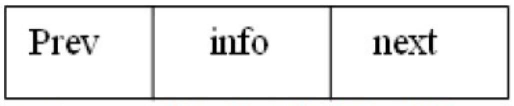


Fig: A node in doubly linked list

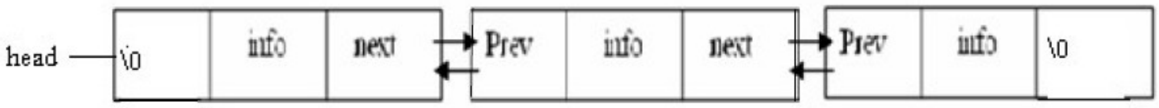


fig: A doubly linked list with three nodes

C representation of doubly linked list:

```
struct node
{
int info;
struct node *prev;
struct node *next;
};
typedef struct node NodeType;
NodeType *head=NULL:
```

Algorithms to insert a node in a doubly linked list:

Algorithm to insert a node at the beginning of a doubly linked list :

```
1.Allocate memory for the new node as,
        newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node
        set newnode->info=item
3. set newnode->prev=newnode->next=NULL
4. set newnode->next=head
5. set head->prev=newnode
6. set head=newnode
7. End
```

Algorithm to insert a node at the end of a doubly linked list :

```
1. Allocate memory for the new node as,
        newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node
        set newnode->info=item
3. set newnode->next=NULL
4. if head==NULL
        set newnode->prev=NULL;
        set head=newnode;
5. if head!=NULL
        set temp=head
        while(temp->next!=NULL)
        temp=temp->next;
        end while
        set temp->next=newnode;
        set newnode->prev=temp
6. End
```

Algorithm to delete a node from beginning of a doubly linked list:

```
1. if head==NULL then
        print "empty list" and exit
2. else
        set hold=head
        set head=head->next
        set head->prev=NULL;
        free(hold)
3. End
```

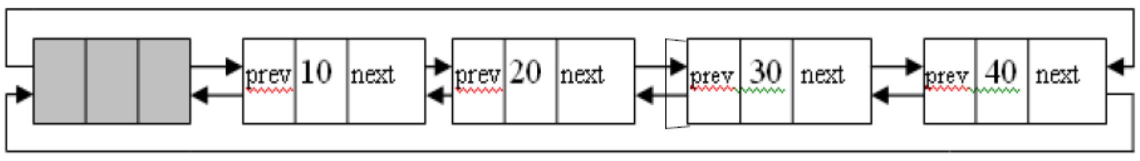
Algorithm to delete a node from end of a doubly linked list:

```
1. if head==NULL then
        print "empty list" and exit
2. else if(head->next==NULL) then
        set hold=head
        set head=NULL
        free(hold)
3. else
        set temp=head;
        while(temp->next->next !=NULL)
        temp=temp->next
        end while
        set hold=temp->next
        set temp->next=NULL
        free(hold)
4. End
```

Circular Doubly Linked List:

A circular doubly linked list is one which has the successor and predecessor pointer in circular manner.

It is a doubly linked list where the next link of last node points to the first node and previous link of first node points to last node of the list. The main objective of considering circular doubly linked list is to simplify the insertion and deletion operations performed on doubly linked list.



head node   Fig: A circular doubly linked list

C representation of doubly circular linked list:

```
struct node
{
int info;
struct node *prev;
struct node *next;
};
typedef struct node NodeType;
NodeType *head=NULL:
```

Algorithm to insert a node at the beginning of a circular doubly linked list:

```
1. Allocate memory for the new node as,
        newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node
        set newnode->info=item
3. set temp=head->next
4. set head->next=newnode
5. set newnode->prev=head
6. set newnode->next=temp
7. set temp->prev=newnode
8. End
```

Algorithm to insert a node at the end of a circular doubly linked list :

```
1. Allocate memory for the new node as,
        newnode=(NodeType*)malloc(sizeof(NodeType))
2. Assign value to info field of a new node
        set newnode->info=item
3. set temp=head->prev
4. set temp->next=newnode
5. set newnode->prev=temp
6. set newnode->next=head
```

Algorithm to delete a node from the beginning of a circular doubly linked list:

```
1. if head->next==NULL then
        print "empty list" and exit
2. else
        set temp=head->next;
        set head->next=temp->next
        set temp->next=head
        free(temp)
3. End
```

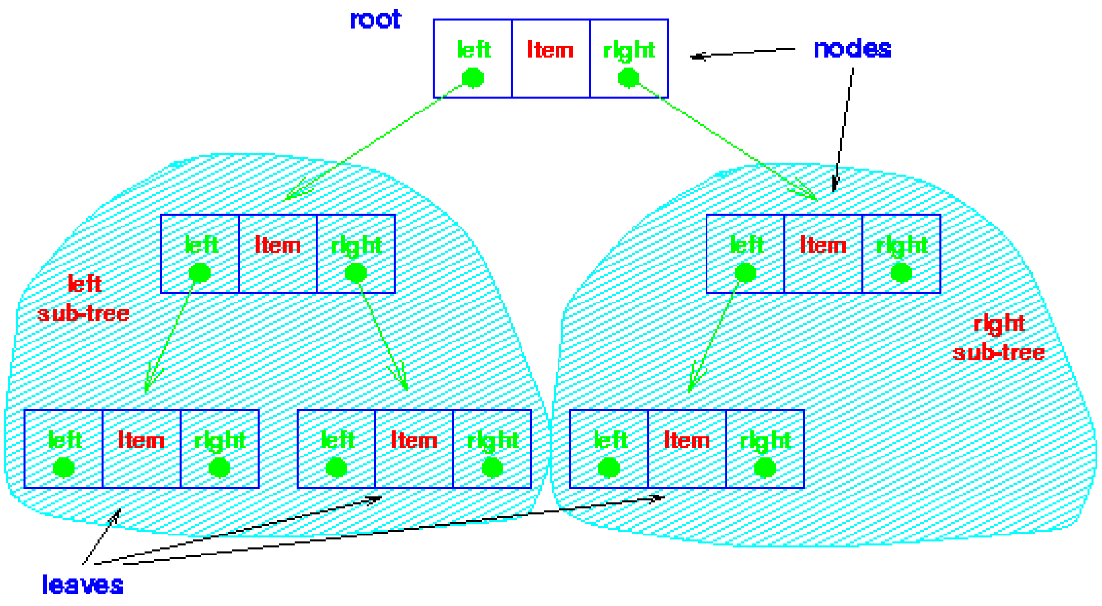
Algorithm to delete a node from the end of a circular doubly linked list:

```
1. if head->next==NULL then
        print "empty list" and exit
2. else
set temp=head->prev;
        set head->left=temp->left
        free(temp)
3. End
```

Tree

A tree is an abstract model of a hierarchical structure that consists of nodes with a parent-child relationship.

• Tree is a sequence of nodes.
• There is a starting node known as root node.
• Every node other than the root has a parent node.
• Nodes may have any number of children.



Characteristics of trees:

✔ Non-linear data structure
✔ combines advantages of an ordered array
✔ searching as fast as in ordered array
✔ insertion and deletion as fast as in linked list

Application:

✔ Directory structure of a file store
✔ Structure of arithmetic expressions
✔ Hierarchy of an organization

Some key terms:

Degree of a node:

The degree of a node is the number of children of that node. In the tree below, degree of node A is 3.

Degree of a Tree:

The degree of a tree is the maximum degree of nodes in a given tree. In the tree below, the node A has maximum degree, thus the degree of the tree is 3.

Path:

It is the sequence of consecutive edges from source node to destination node. There is a single unique path from the root to any node.

Height of a node:

The height of a node is the maximum path length from that node to a leaf node. A leaf node has a height of 0.

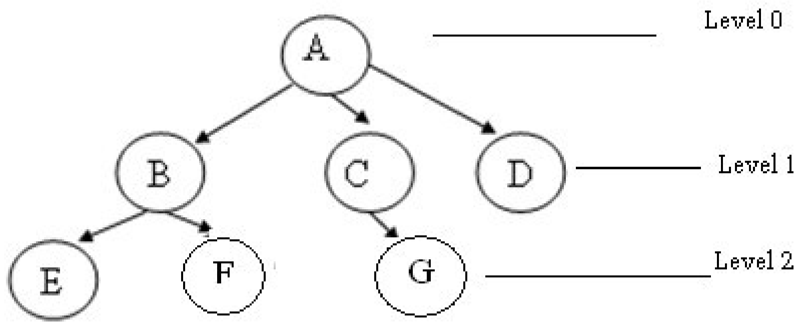
Height of a tree:

The height of a tree is the height of the root.

Depth of a node:
Depth of a node is the path length from the root to that node. The root node has a depth of 0.

Depth of a tree:
Depth of a tree is the maximum level of any leaf in the tree. This is equal to the longest path from the root to any leaf.

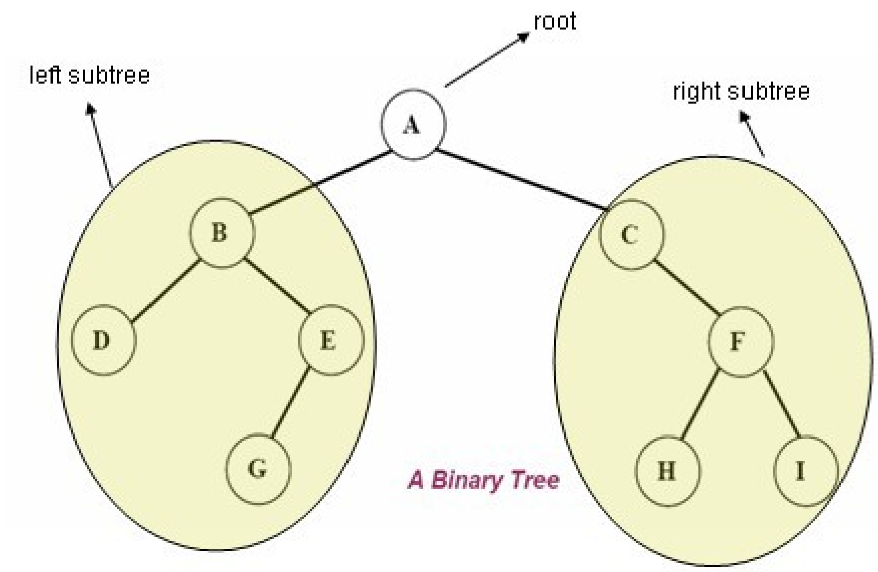
Level of a node:
The level of a node is 0, if it is root; otherwise it is one more than its parent.
Illustration:



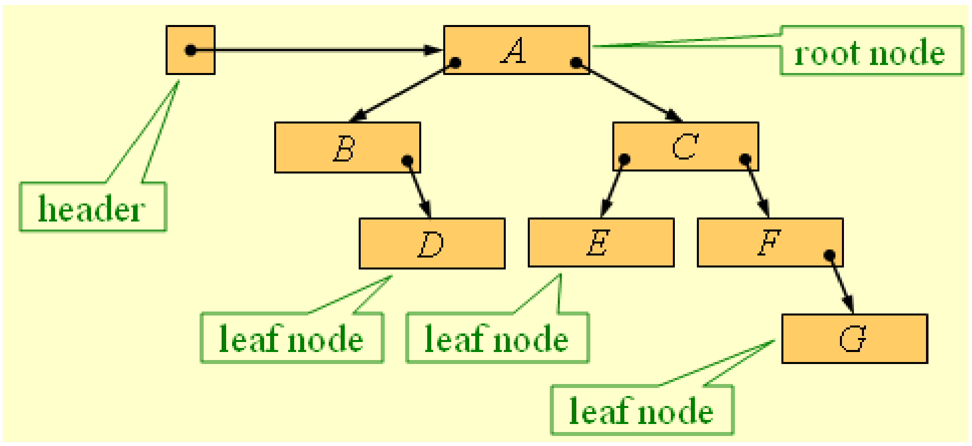
✔ A is the root node
✔ B is the parent of E and F
✔ D is the sibling of B and C
✔ E and F are children of B
✔ E, F, G, D are external nodes or leaves
✔ A, B, C are internal nodes
✔ Depth of F is 2
✔ the height of tree is 2
✔ the degree of node A is 3
✔ the degree of tree is 3

Binary Trees

A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the ***root*** of the tree. The other two subsets are themselves binary trees called the ***left*** and ***right sub-trees*** of the original tree. A left or right sub tree can be empty. Each element of a binary tree is called a ***node*** of the tree. The following figure shows a binary tree with 9 nodes where A is the root.



A **binary tree** consists of a **header**, plus a number of **nodes** connected by **links** in a hierarchical data structure:



Binary tree properties:

✔ If a binary tree contains m nodes at level l, it contains at most **2m** nodes at level **l+1**.

✔ Since a binary tree can contain at most 1 node at level 0 (the rot), it contains at most **2l** nodes at level **l.**

Types of binary tree

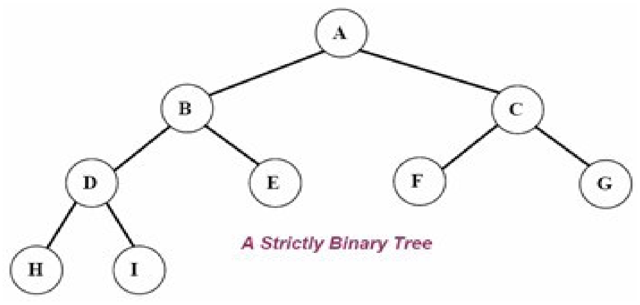
✔ Complete binary tree

✔ Strictly binary tree

✔ Almost complete binary tree

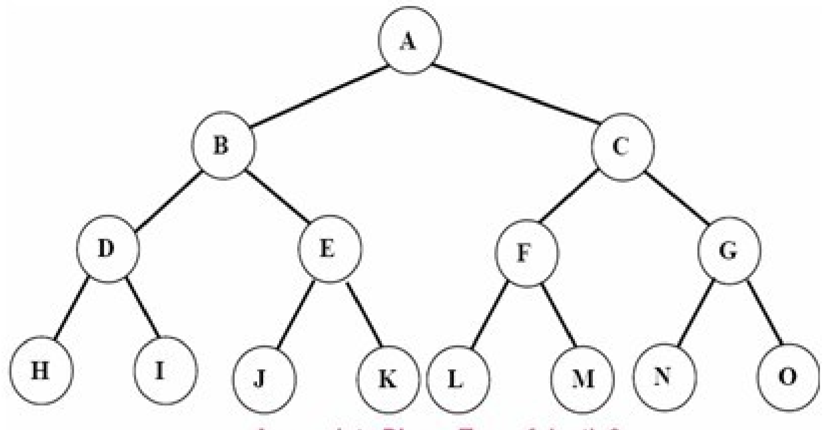
Strictly binary tree:

If every non-leaf node in a binary tree has nonempty left and right sub-trees, then such a tree is called a ***strictly binary tree***.



Complete binary tree:

A ***complete binary tree*** of depth d is called strictly binary tree if all of whose leaves are at level **d**. A complete binary tree with depth **d** has **2^d** leaves and **2^d -1** non-leaf nodes(internal)



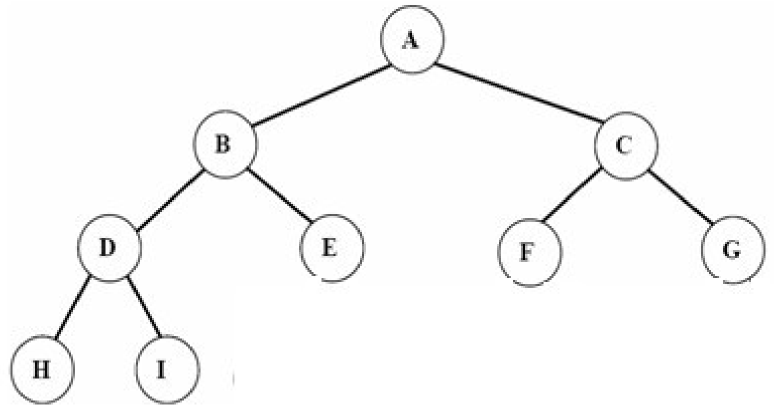
A complete Binary Tree of depth 3

Almost complete binary tree:

A binary tree of depth **d** is an almost complete binary tree if:

✔ Any node **nd** at level less than **d-1** has two sons.

✔ For any nose **nd** in the tree with a right descendant at level **d, nd** must have a left son and every left descendant of **nd** is either a leaf at level **d** or has two sons.



Operations on Binary tree:

✔ **father(n,T):**Return the parent node of the node n in tree T. If n is the root, NULL is returned.

✔ **LeftChild(n,T):**Return the left child of node n in tree T. Return NULL if n does not have a left child.

✔ **RightChild(n,T)**:Return the right child of node n in tree T. Return NULL if n does not have a right child.

✔ **Info(n,T):** Return information stored in node n of tree T (i.e. content of a node).

✔ **Sibling(n,T):** return the sibling node of node **n** in tree T. Return NULL if n has no sibling.

✔ **Root(T):** Return root node of a tree if and only if the tree is nonempty.

✔ **Size(T):** Return the number of nodes in tree T

✔ **MakeEmpty(T):** Create an empty tree T

✔ **SetLeft(S,T):** Attach the tree S as the left sub-tree of tree T

✔ **SetRight(S,T):** Attach the tree S as the right sub-tree of tree T.

✔ **Preorder(T):** Traverses all the nodes of tree T in preorder.

✔ **postorder(T):** Traverses all the nodes of tree T in postorder
✔ **Inorder(T):** Traverses all the nodes of tree T in inorder.

C representation for Binary tree:

```
struct bnode
{
int info;
struct bnode *left;
struct bnode *right;
};
struct bnode *root=NULL;
```

Tree traversal

The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner.
There are three popular methods of traversal
✔ Pre-order traversal
✔ In-order traversal
✔ Post-order traversal

Pre-order traversal:

The preorder traversal of a nonempty binary tree is defined as follows:
✔ Visit the root node
✔ Traverse the left sub-tree in preorder
✔ Traverse the right sub-tree in preorder

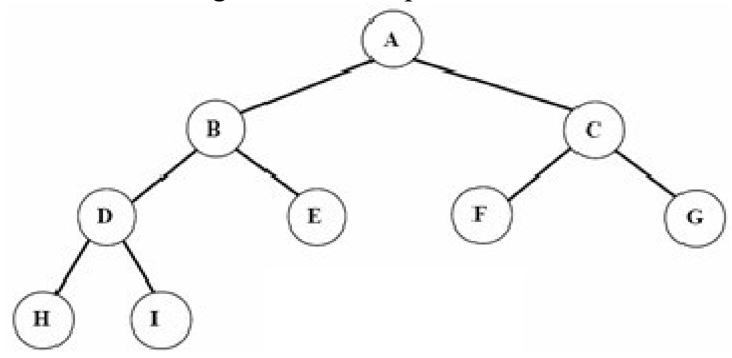


Fig: Binary tree
The preorder traversal output of the given tree is: A B D H I E C F G
The preorder is also known as depth first order.

C function for preorder traversing:

```
void preorder(struct bnode *root)
{
if(root!=NULL)
{
printf("%c", root->info);
preorder(root->left);
preorder(root->right);
}
}
```

In-order traversal:

The inorder traversal of a nonempty binary tree is defined as follows:

✔ Traverse the left sub-tree in inorder

✔ Visit the root node

✔ Traverse the right sub-tree in inorder

The inorder traversal output of the given tree is: H D I B E A F C G

C function for inorder traversing:

```
void inorder(struct bnode *root)
{
if(root!=NULL)
{
inorder(root->left);
printf("%c", root->info);
inorder(root->right);
}
}
```

Post-order traversal:

The post-order traversal of a nonempty binary tree is defined as follows:

✔ Traverse the left sub-tree in post-order

✔ Traverse the right sub-tree in post-order

✔ Visit the root node

The post-order traversal output of the given tree is: H I D E B F G C A

C function for post-order traversing:

```
void post-order(struct bnode *root)
{
if(root!=NULL)
{
post-order(root->left);
post-order(root->right);
printf("%c", root->info);
}
}
```

Binary search tree(BST):

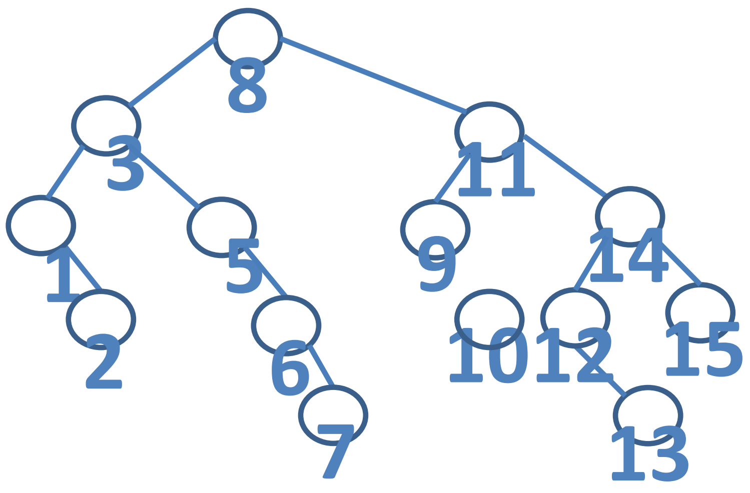
A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) ans satisfies the following conditions:

✔ All keys in the left sub-tree o the root are smaller than the key in the root node

✔ All keys in the right sub-tree of the root are greater than the key in the root node

✔ The left and right sub-trees of the root are again binary search trees

Given the following sequence of numbers,
8,3,1,2,5,6,7,11,9,10,14,12,13,15
The following binary search tree can be constructed:



Operations on Binary search tree(BST):

Following operations can be done in BST:

✔ **Search(k, T):** Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.

✔ **Insert(k, T):** Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.

✔ **Delete(k, T):**Delete a node with value k in the info field from the tree T such that the property of BST is maintained.

✔ **FindMin(T), FindMax(T):** Find minimum and maximum element from the given nonempty BST.

Searching through the BST:

•***Problem: Search for a given target value in a BST.***

•***Idea: Compare the target value with the element in the root node.***

✔ If the target value is **equal**, the search is successful.

✔ If target value is **less**, search the left subtree.

✔ If target value is **greater**, search the right subtree.

✔ If the subtree is **empty**, the search is unsuccessful.

BST search algorithm :

To find which if any node of a BST contains an element equal to *target*:

```
1. Set curr to the BST's root.
2. Repeat:
2.1. If curr is null:
2.1.1. Terminate with answer none.
2.2. Otherwise, if target is equal to curr's element:
2.2.1. Terminate with answer curr.
2.3. Otherwise, if target is less than curr's element:
2.3.1. Set curr to curr's left child.
2.4. Otherwise, if target is greater than curr's element:
2.4.1. Set curr to curr's right child.
3. end
```

C function for BST searching:

```
void BinSearch(struct bnode *root , int key)
{
```

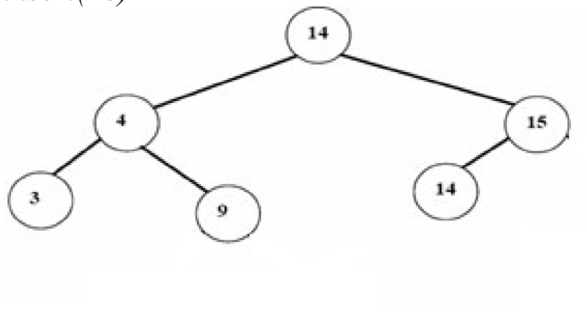
```
if(root == NULL)
{
printf("The number does not exist");
exit(1);
}else if (key == root->info)
{
printf("The searched item is found"):
}else if(key < root->info)
return BinSearch(root->left, key);
else
return BinSearch(root->right, key);
}
```

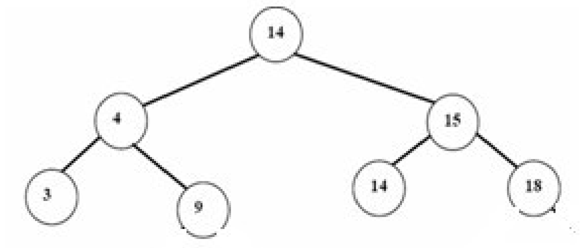
Insertion of a node in BST:

To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this a search is carried out. If the search is unsuccessful, then item is inserted.

•***Idea***: To insert a new element into a BST, proceed as if searching for that element. If the element is not already present, the search will lead to a null link. Replace that null link by a link to a leaf node containing the new element.

insert(18)





BST insertion algorithm :

To insert the element *elem* into a BST:
1. Set *parent* to null, and set *curr* to the BST's root.
2. Repeat:
2.1. If *curr* is null:
2.1.1. Replace the null link from which *curr* was taken
(either the BST's root or *parent*'s left child or *parent*'s right child) by a
link to a newly-created leaf node with element *elem*.
2.1.2. Terminate.
2.2. Otherwise, if *elem* is equal to *curr*'s element:
2.2.1. Terminate.
2.3. Otherwise, if *elem* is less than *curr*'s element:
2.3.1. Set *parent* to *curr*, and set *curr* to *curr*'s left child.
2.4. Otherwise, if *elem* is greater than *curr*'s element:
2.4.1. Set *parent* to *curr*, and set *curr* to *curr*'s right child.
3.End

C function for BST insertion:

```
void insert(struct bnode *root, int item)
{
```

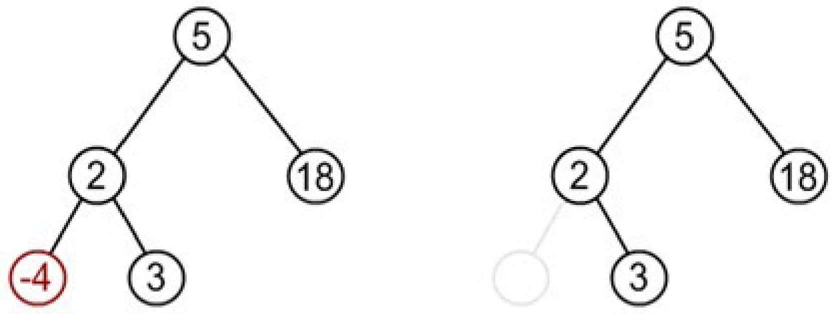
```
if(root=NULL)
{
root=(struct bnode*)malloc (sizeof(struct bnode));
root->left=root->right=NULL;
root->info=item;
}
else
{
if(item<root->info)
root->left=insert(root->left, item);
else
root->right=insert(root->right, item);
}
}
```

Deleting a node from the BST:

While deleting a node from BST, there may be three cases:

1. The node to be deleted may be a leaf node:

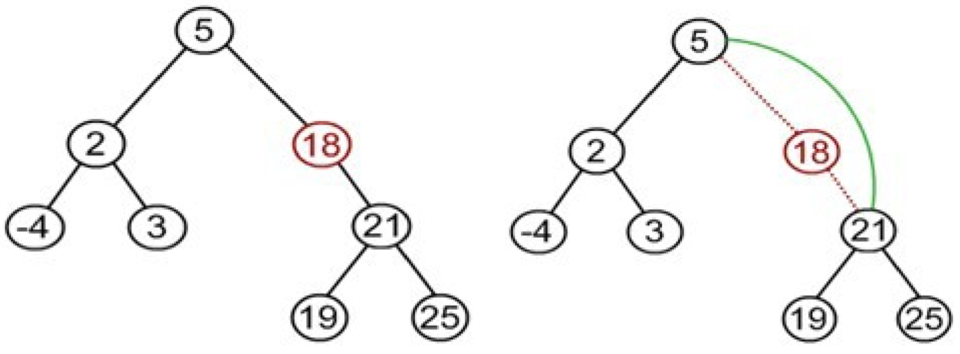
In this case simply delete a node and set null pointer to its parents those side at which this deleted node exist.

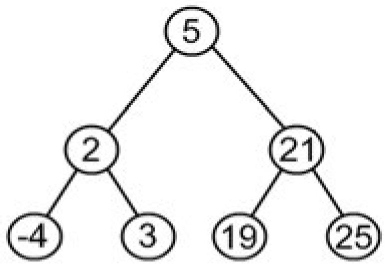


2. **The node to be deleted has one child:**

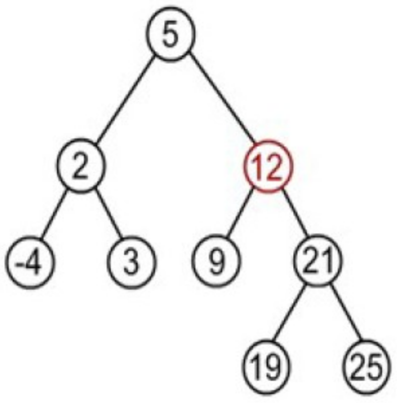
In this case the child of the node to be deleted is appended to its parent node.

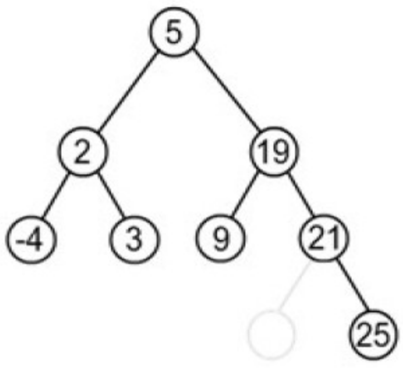
Suppose node to be deleted is 18





3. the node to be deleted has two children:
In this case node to be deleted is replaced by its in-order successor node.
OR
If the node to be deleted is either replaced by its right sub-trees leftmost node or its left sub-trees rightmost node.
Suppose node to deleted is 12
Find minimum element in the right sub-tree of the node to be removed. In current example it is19.





General algorithm to delete a node from a BST:

1. start
2. if a node to be deleted is a leaf nod at left side then simply delete and set null pointer to it's parent's left pointer.
3. If a node to be deleted is a leaf node at right side then simply delete and set null pointer to it's parent's right pointer
4. if a node to be deleted has on child then connect it's child pointer with it's parent pointer and delete it from the tree
5. if a node to be deleted has two children then replace the node being deleted either by
a. right most node of it's left sub-tree or
b. left most node of it's right sub-tree.
6. End

The deleteBST function:

```
struct bnode *delete(struct bnode *root, int item)
{
struct bnode *temp;
if(root==NULL)
{
printf("Empty tree");
return;
} else if(item<root->info)
root->left=delete(root->left, item);
else if(item>root->info)
root->right=delete(root->right, item);
else if(root->left!=NULL &&root->right!=NULL) //node has two child
{
temp=find_min(root->right);
root->info=temp->info;
root->right=delete(root->right, root->info);
}
else
{
temp=root;
if(root->left==NULL)
root=root->right;
else if(root->right==NULL)
root=root->left;
free(temp);
}
return(temp);
} /*********find minimum element function*********/
struct bnode *find_min(struct bnode *root)
{
if(root==NULL)
return0;
else if(root->left==NULL)
return root;
else
return(find_min(root->left));
}
```

Sorting

Sorting is among the most basic problems in algorithm design. We are given a sequence of items, each associated with a given key value. The problem is to permute the items so that they are in increasing (or decreasing) order by key. Sorting is important because it is often the first step in more complex algorithms. Sorting algorithms are usually divided into two classes, **internal sorting algorithms**, which assume that data is stored in an array in main memory, and **external sorting algorithm**, which assume that data is stored on disk or some other device that is best accessed sequentially. We will consider internal sorting at first. Sorting algorithms often have additional properties that are of interest, depending on the application. Here are two important properties.

In-place: The algorithm uses no additional array storage, and hence (other than perhaps the system's recursion stack) it is possible to sort very large lists without the need to allocate additional working storage.

Stable: A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key, remain sorted on the first key.

In brief the sorting is a process of arranging the items in a list in some order that is either ascending or descending order.

Let a[n] be an array of n elements a0,a1,a2,a3........,an-1 in memory. The sorting of the array a[n] means arranging the content of a[n] in either increasing or decreasing order.
i.e. a0<=a1<=a2<=a3<.=.......<=an-1
consider a list of values: 2 ,4 ,6 ,8 ,9 ,1 ,22 ,4 ,77 ,8 ,9
After sorting the values: 1, 2, 4, 4, 6, 8, 8,9 , 9 , 22, 77

Bubble Sort

The basic idea of this sort is to pass through the array sequentially several times. Each pass consists of comparing each element in the array with its successor (for example a[i] with a[i+1]) and interchanging the two elements if they are not in the proper order. For example, consider the following array:

Initially:

25	57	48	37	12	92	86	33
0	1	2	3	4	5	6	7

After Pass 1:

25	48	37	12	57	86	33	92
0	1	2	3	4	5	6	7

After Pass 2:

25	37	12	48	57	33	86	92
0	1	2	3	4	5	6	7

After Pass 3:

25	12	37	48	33	57	86	92
0	1	2	3	4	5	6	7

After Pass 4:

12	25	37	33	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 5:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 6:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

After Pass 7:

12	25	33	37	48	57	86	92
0	1	2	3	4	5	6	7

Here, we notice that after each pass, an element is placed in its proper order and is not considered in succeeding passes. Furthermore, we need n – 1 passes to sort n elements.

Algorithm

```
BubbleSort(A, n)
{
   for(i = 0; i <n-1; i++)
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
            {
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
            }
        }
}
```

Time Complexity:

Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:

Time complexity = (n-1) + (n-2) + (n-3) + …………………………. +2 +1

= $O(n^2)$

There is no best-case linear time complexity for this algorithm.
Space Complexity:
Since no extra space besides 3 variables is needed for sorting, Space complexity = O(n)

Selection Sort

Idea: Find the least (or greatest) value in the array, swap it into the leftmost (or rightmost) component (where it belongs), and then forget the leftmost component. Do this repeatedly. Let a[n] be a linear array of n elements. The selection sort works as follows:
pass 1: Find the location 'loc' of the smallest element into the list of n elements a[0], a[1], a[2], a[3], ……...,a[n-1] and then interchange a[loc] and a[0].
Pass 2: Find the location 'loc' of the smallest element into the sub-list of n-1 elements a[1], a[2], a[3], ……...,a[n-1] and then interchange a[loc] and a[1] such that a[0], a[1] are sorted.
…...................... and so on.
Then we will get the sorted list a[0]<=a[1]<= a[2]<=a[3]<= ……..<=a[n-1].
Algorithm:

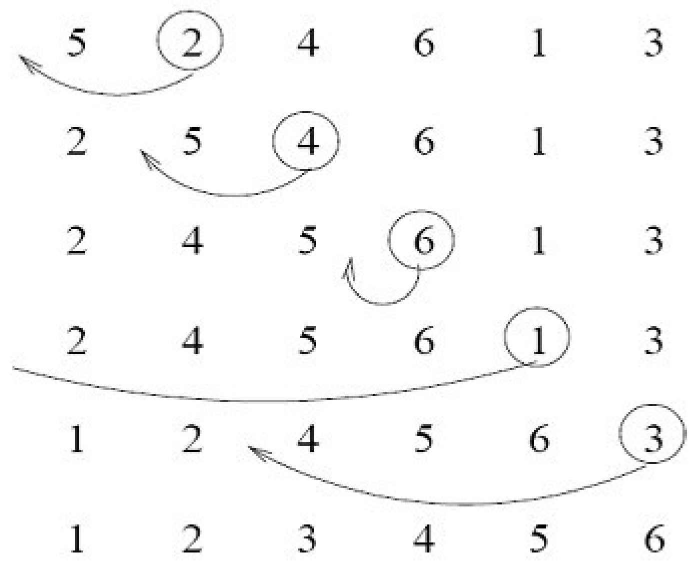
```
SelectionSort(A)
{
        for( i = 0;i < n -1;i++)
        {
                least=A[i];
                p=i;
                for (j = i + 1;j < n ;j++)
                {
                        if (A[j] < least)
                                least= A[j]; p=j;
                }
                swap(A[i],A[p]);
        }
}
```

Time Complexity:
Inner loop executes for (n-1) times when i=0, (n-2) times when i=1 and so on:
Time complexity = (n-1) + (n-2) + (n-3) + ………………………… +2 +1= $O(n^2)$
There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.
Space Complexity:
Since no extra space besides 5 variables is needed for sorting, Space complexity = O(n)

Insertion Sort

Idea: like sorting a hand of playing cards start with an empty left hand and the cards facing down on the table. Remove one card at a time from the table, and insert it into the correct position in the left hand. Compare it with each of the cards already in the hand, from right to left. The cards held in the left hand are sorted. Suppose an array a[n] with n elements. The insertion sort works as follows:
pass 1: a[0] by itself is trivially sorted.
Pass 2: a[1] is inserted either before or after a[0] so that a[0], a[1] is sorted.
Pass 3: a[2] is inserted into its proper place in a[0],a[1] that is before a[0], between a[0] and a[1], or after a[1] so that a[0],a[1],a[2] is sorted.
…..
pass N: a[n-1] is inserted into its proper place in a[0],a[1],a[2],........,a[n-2] so that a[0],a[1],a[2],............,a[n-1] is sorted with n elements.

Example:



Algorithm:

```
InsertionSort(A)
     {
          for (i=1;i<n;i++)
          {
               key = A[ i]
               for(j=i; j>0 && A[j] >key; j--)
               {
                    A[j + 1] = A[j]
               }
               A[j + 1] = key

          }
     }
```

Time Complexity:

Worst Case Analysis:

Array elements are in reverse sorted order

Inner loop executes for 1 times when i=1, 2 times when i=2... and n-1 times when i=n-1:

Time complexity = 1 + 2 + 3 + ... +(n-2) +(n-1)

= $O(n^2)$

Best case Analysis:

Array elements are already sorted

Inner loop executes for 1 times when i=1, 1 times when i=2... and 1 times when i=n-1:

Time complexity = 1 + 2 + 3 + ... +1 +1

= O(n)

Space Complexity:

Since no extra space besides 5 variables is needed for sorting

Space complexity = O(n)

Quick Sort

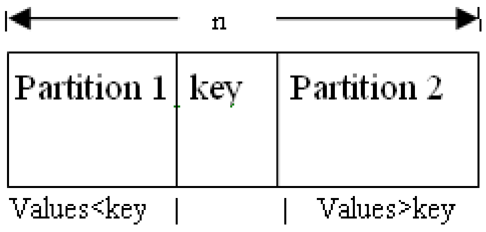
Quick sort developed by C.A.R Hoare is an unstable sorting. In practice this is the fastest sorting method. It possesses very good average case complexity among all the sorting algorithms. This algorithm is based on the divide and conquer paradigm. The main idea behind this sorting is partitioning of the elements.

Steps for Quick Sort:

Divide: partition the array into two nonempty sub arrays.

Conquer: two sub arrays are sorted recursively.

Combine: two sub arrays are already sorted in place so no need to combine.



Example: a[]={5, 3, 2, 6, 4, 1, 3, 7}

```
5    3    2    6    4    1    3    7
x                                  y

5    3    2    6    4    1    3    7
               x              y    {swap x & y}

5    3    2    3    4    1    6    7
                         y    x    {swap y and pivot}

1    3    2    3    4    5    6    7
                         p
 (1   3    2    3    4)   5   (5    7)
```

and continue this process for each sub-arrays and finally we get a sorted array.

Algorithm:

```
QuickSort(A,l,r)
{
        if(l<r)
        {
        p = Partition(A,l,r);
        QuickSort(A,l,p-1);
        QuickSort(A,p+1,r);
        }
}
Partition(A,l,r)
{
x =l;
y =r ;
p = A[l];
while(x<y)
{
while(A[x] <= p)
        x++;
while(A[y] >=p)
        y--;
if(x<y)
        swap(A[x],A[y]);
}
A[l] = A[y];
A[y] = p;
return y; //return position of pivot
}
```

Time Complexity:

Best Case:

Divides the array into two partitions of equal size, therefore T(n) = 2T(n/2) + O(n) , Solving this recurrence we get, T(n)=O(nlogn)

Worst case:

when one partition contains the n-1 elements and another partition contains only one element. Therefore its recurrence relation is:

T(n) = T(n-1) + O(n), Solving this recurrence we get

$T(n)=O(n^2)$

Average case:

Good and bad splits are randomly distributed across throughout the tree

T1(n)= 2T'(n/2) + O(n) Balanced

T'(n)= T(n –1) + O(n) Unbalanced

Solving:

$B(n)= 2(B(n/2 -1) + \Theta(n/2)) + \Theta(n)$

$= 2B(n/2 -1) + \Theta(n)$

= O(nlogn)

=>T(n)=O(nlogn)

Merge Sort

To sort an array A[l . . r]:

• **Divide**– Divide the n-element sequence to be sorted into two sub-sequences of n/2 elements

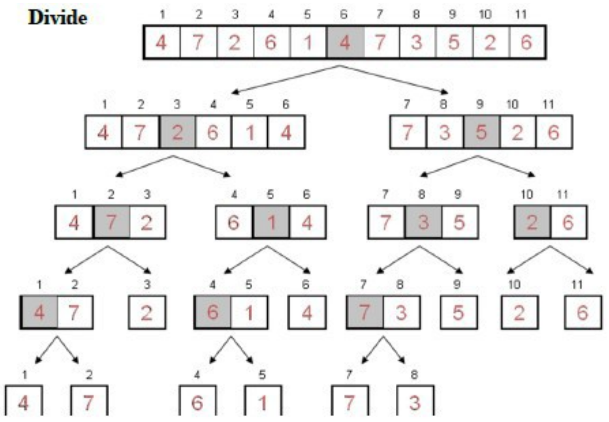
• **Conquer**– Sort the sub-sequences recursively using merge sort. When the size of the sequencesis 1 there is nothing more to do

•**Combine**

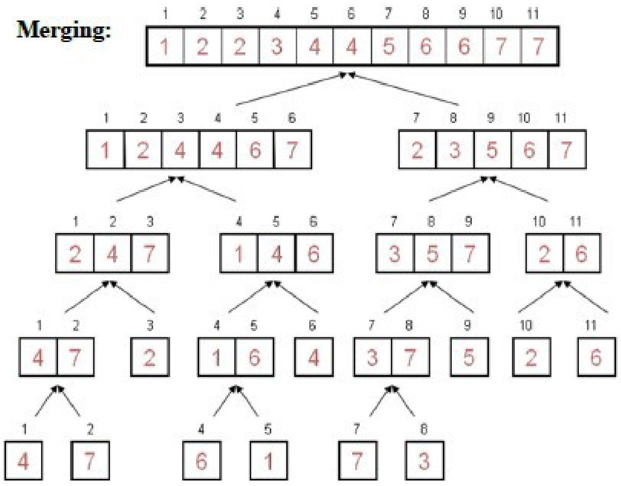
Merge the two sorted sub-sequences

Example: a[]={4, 7, 2, 6, 1, 4, 7, 3, 5, 2, 6}

Divide



Merging:



Algorithm:

```
        MergeSort(A, 1, r)
        {
                If ( 1 < r)
                {                                       //Check for base case
                  m = ⌊(1 + r)/2⌋                      //Divide
                        MergeSort(A, 1, m)              //Conquer
                        MergeSort(A, m + 1, r)          //Conquer
                        Merge(A, 1, m+1, r)             //Combine
                }
        }
Merge(A,B,l,m,r)
{
        x=l, y=m;
        k=l;
        while(x<m && y<r)
        {
                if(A[x] < A[y])
                {
                        B[k]= A[x];
                        k++; x++;
                }
                else
                {
                        B[k] = A[y];
                        k++; y++;
                }
        }
        while(x<m)
        {
                A[k] = A[x];
                k++; x++;
        }
        while(y<r)
        {
                A[k] = A[y];
                k++; y++;
        }
        for(i=l;i<= r; i++)
        {
                A[i] = B[i]
        }
}
```

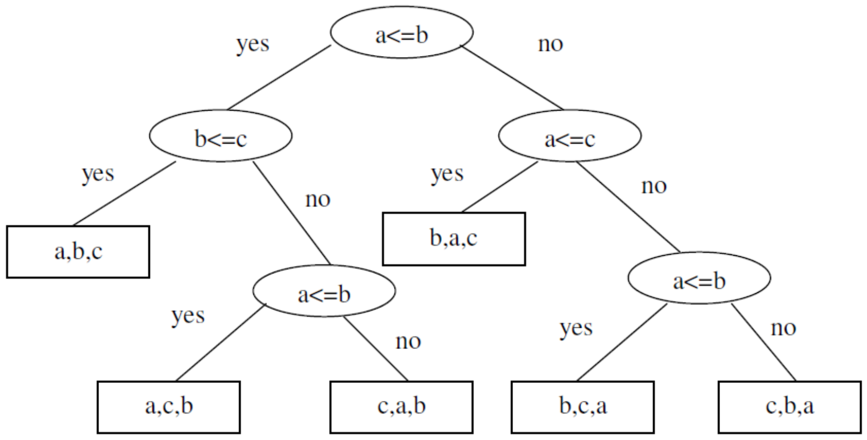
Time Complexity:
Recurrence Relation for Merge sort:
T(n) = 1 if n=1
T(n) = 2 T(n/2) + O(n) if n>1
Solving this recurrence we get
T(n) = O(nlogn)
Space Complexity:
It uses one extra array and some extra variables during sorting, therefore
Space Complexity= 2n + c = O(n)

Sorting Comparisons

Sort	Worst Case	Average Case	Best Case	Comments
Insertion Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n)$	
Selection Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	(*Unstable)
Bubble Sort	$\Theta(n^2)$	$\Theta(n^2)$	$\Theta(n^2)$	
Merge Sort	$\Theta(nlogn)$	$\Theta(nlogn)$	$\Theta(nlogn)$	Requires Memory
Heap Sort	$\Theta(nlogn)$	$\Theta(nlogn)$	$\Theta(nlogn)$	*Large constants
Quick Sort	$\Theta(n^2)$	$\Theta(nlogn)$	$\Theta(nlogn)$	*Small constants

Lower Bound for Simple Sorting

Comparison sorts algorithms compare pair of data for relation greater than or smaller than. They work in same manner for every kind of data. Consider all the possible comparison runs of sorting algorithm on input size n. When we unwind loops and recursive calls and focus on the point where there is comparison, we get binary decision tree. The tree describes all the possibilities for the algorithm execution on an input of n elements.



At the leaves, we have original input in the sorted order. By comparison we can generate all the possible permutations of the inputs are possible so there must be at least n! leaves in the tree. A complete tree of height h has 2^h leaves, so our decision tree must have h>= log(n!), but logn! is Ω(nlogn). Thus, the number of comparison in the worst case is Ω(nlogn). So we can say that runtime complexity of any comparison sort is Ω(nlogn).