

Data Structure & Analysis

Anuradha
BHATIA

Table of Contents

1. Introduction to Data Structure.....	1
2. Stacks.....	10
3. Queues.....	38
4. Link List.....	69
5. Sorting & Searching.....	92
6. Trees and Graph.....	115

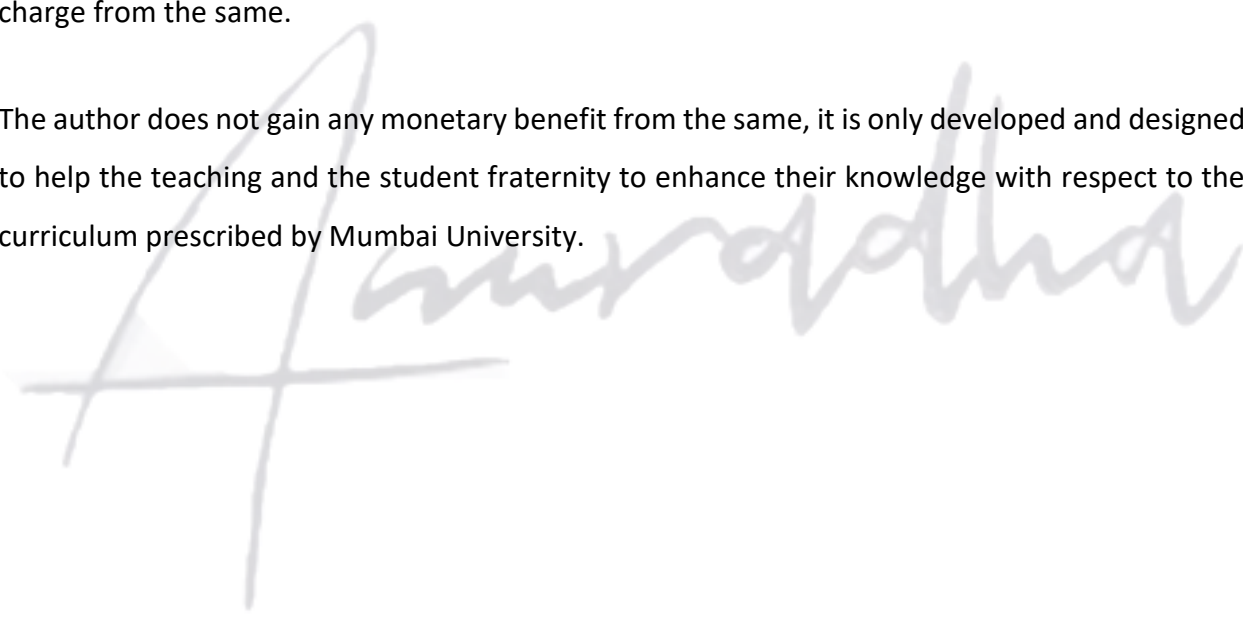
Auradha

Disclaimer

The content of the book is the copyright property of the author, to be used by the students for the reference for the subject “Data Structure and Analysis”, CPE8035 and BEITL802, Eighth Semester, for the Final Year Computer Engineering and Information Technology Mumbai University.

The complete set of the e-book is available on the author’s website <https://github.com/anuradhabhatia/Notes>, and students are allowed to download it free of charge from the same.

The author does not gain any monetary benefit from the same, it is only developed and designed to help the teaching and the student fraternity to enhance their knowledge with respect to the curriculum prescribed by Mumbai University.



1. Introduction to Data Structure

CONTENTS

- 1.1 Basic Terminology
 - 1. Elementary data structure organization
 - 2. Classification of data structure
- 1.2 Operations on data structures
- 1.3 Different Approaches to designing an algorithm
 - 1. Top-Down approach
 - 2. Bottom-up approach
- 1.4 Complexity
 - 1. Time complexity
 - 2. Space complexity
- 1.5 Big 'O' Notation

1.1 Basic Terminology

1. Elementary data structure organization

- i. Data can be organized in many ways and data structures is one of these ways.
- ii. It is used to represent data in the memory of the computer so that the processing of data can be done in easier way.
- iii. Data structures is the logical and mathematical model of a particular organization of data.

2. Classification of data structure

The data structures can be of the following types:

- i. Linear Data structures: In these data structures the elements form a sequence. Such as Arrays, Linked Lists, Stacks and Queues are linear data structures.
- ii. Non-Linear Data Structures: In these data structures the elements do not form a sequence. Such as Trees and Graphs are non-linear data structures.

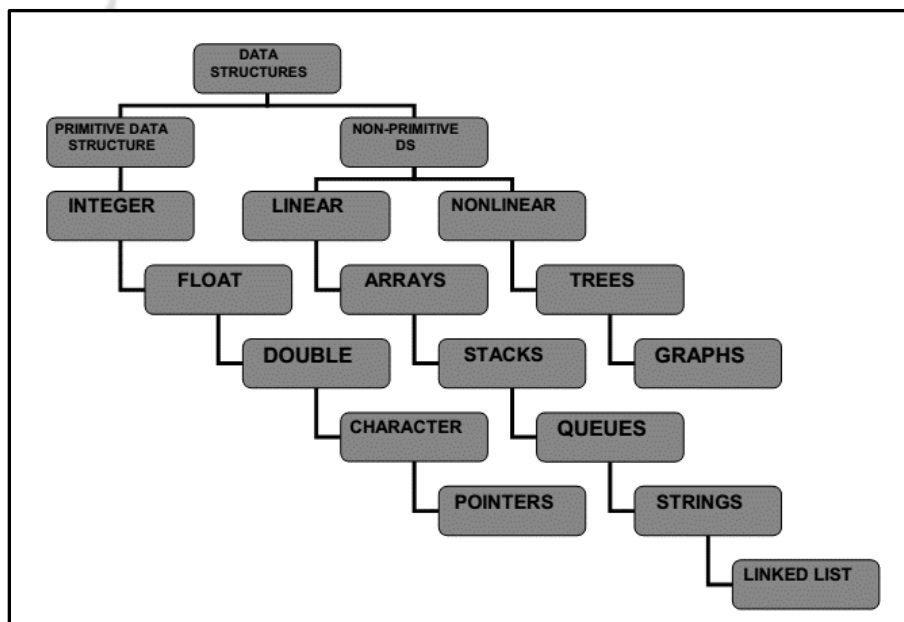


Figure 1.1

1.2 Operations on data structures

1. Traversing, Inserting, deleting

The following four operations play a major role in this text.

- i. **Traversing:** Accessing each record exactly once so that certain items in the record may be processed.
- ii. **Inserting:** adding a new record to the structure.
- iii. **Deleting:** Removing a record from the structure. Sometimes two or more of the operations may be used in a given situation; for example we may want to delete the record with a given key, which may mean we first need to search for the location of the record.

2. Searching, sorting, merging

- i. **Searching:** Finding the location of the record with a given key value, or finding the locations of all records, which satisfy one or more conditions.
- ii. **Sorting:** Arranging the records in some logical order (for example alphabetically according to some Name key, or in numerical order according to some Number key. such as account number.
- iii. **Merging:** Combining the records in two different sorted files into a single sorted file

1.3 Different Approaches to designing an algorithm

1. Top-Down approach

- i. This approach divides the problem in to manageable segments.
- ii. This technique generates diagrams that are used to design the system.
- iii. Frequently several alternate solutions to the programming problem are developed and compared during this phase.

2. Bottom-up approach

- i. The Bottom-up approach is an older approach which gives early emphasis on coding.
- ii. Since the programmer does not have a master plan for the project, the resulting program may have many error ridden segments.

1.4 Complexity

1. Time complexity

Time Complexity is divided in to THREE Types.

- i. **Best Case Time Complexity:** Efficiency of an algorithm for an input of size N for which the algorithm takes the smallest amount of time.
- ii. **Average Case Time Complexity:** It gives the expected value of $T(n)$. Average case is used when worst case and best case doesn't gives any necessary information about algorithm's behavior, then the algorithm's efficiency is calculated on Random input.
- iii. **Worst Case Time Complexity:** efficiency of an algorithm for an input of size N for which the algorithm takes the longest amount of time.

Time complexity is calculated on the basis of

- i. **Operation Count:** Find the basic operation $a = a * b$; this code takes 1 unit of time

```
For (i=0; i<n; i++)  
{  
    for (j=0; j<n; j++)  
        if (j==5)  
        {  
            printf("Internal Instruction");  
        }  
}
```

Figure 1.2

- ii. **Step Count**
Example

Statement	Step Count
<code>x = a + b;</code>	1
<code>for(i=1; i <= n; i++)</code>	n
<code>for(i=1; i <= n; i++)</code> <code>for(j=1; j <= n; j++)</code> <code>x = a + b;</code>	n ²

Table 1.1

Example

```
int sum (int a[], int n)
{
    int i , sum=0;
    sum-count ++;
    for (i=0; i<n; i++)
    {
        for-count ++;
        sum = sum + a[i];
    }
    for-count++;
    return-count ++;
    return sum;
}
```

Figure 1.3

The Step count for “sum” = 1

The step count for, “for statement” = n + 1

The step count for, “assignment” = n

The step count for, “return” = 1

Total steps = 2n + 3

2. Space complexity

- Specifies the amount of temporary storage required for running the algorithm.

- ii. We do not count storage required for input and output when we are working with the same type of problem and with different algorithms.
- iii. Space needed by algorithm consists of the following component.
 - a. The fixed static part: it is independent of characteristics ex: number of input/output. This part includes the instruction space (i.e. space for code). Space for simple variables, space for constants, and fixed size component variables. Let C_p be the space required for the code segment (i.e. static part)
 - b. The variable dynamic part: that consists of the space needed by component variables whose size is dependent on the particular problem instance at runtime. i.e. space needed by reference variables, and the recursion stack space. Let S_p be the space for the dynamic part.
 - c. Overall space required by program: $S(p) = C_p + S_p$

Finding the sum of three numbers:

```
#include<stdio.h>
Void main()
{
    int x, y, z, sum;
    Printf("Enter the three numbers");
    Scanf("%d, %d, %d", &x, &y, &z);
    Sum= x + y + z;
    Printf("the sum = %d", sum);
}
```

Figure 1.4

- iv. In the above program there are no instance characteristics and the space needed by x, y, z and sum is independent of instance characteristics. The space for each integer variable is 2. We have 4 integer variables and space needed by x, y, z and sum are $4 * 2 = 8$ bytes.

$$S(p) = C_p + S_p$$

$$S(p) = 8 + 0$$

$$S(p) = 8$$

1.5 Big 'O' Notation

Asymptotic Notation

The asymptotic behavior of a function is the study of how the value of a function varies for larger values of "n" where "n" is the size of the input. Using asymptotic behavior we can easily find the time efficiency of an algorithm.

Different asymptotic notations are:

- i. **Big O (big oh):** it is the formal method of expressing the upper bound of an algorithm's running time. It's a measure of longest amount of time it could possibly take for the algorithm to complete. Let $f(n)$ is the time efficiency of an algorithm. The function $f(n)$ is said to be Big-oh of $g(n)$, denoted by $f(n) \in O(g(n))$ OR $f(n) \approx O(g(n))$.

Such that there exist a +ve constant "c" and +ve integer n_0 satisfying the constraint.

$$f(n) \leq c * g(n) \text{ for all } n \geq n_0$$

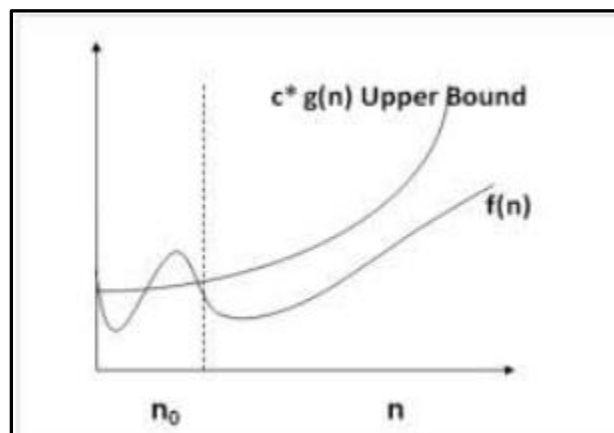


Figure 1.5

- ii. **Ω notation (Big-Omega):** Let $f(n)$ be the time complexity of an algorithm. The function $f(n)$ is said to be Big-Omega of $g(n)$ which is denoted by

$$f(n) \in \Omega(g(n)) \text{ OR } f(n) \approx \Omega(g(n))$$

such that there exist a +ve constant “ c ” and non-negative integer n_0 satisfying the constraint

$$f(n) \geq c * g(n) \text{ for all } n \geq n_0$$

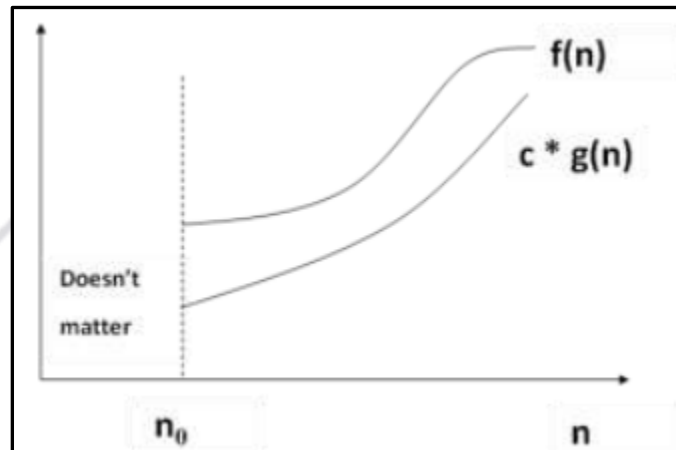


Figure 1.6

- iii. **Θ notation (Theta):** Let $f(n)$ be the time complexity of an algorithm. The function $f(n)$ is said to be Big-Theta of $g(n)$ which is denoted by
- $$f(n) \in \Theta(g(n)) \text{ OR } f(n) \approx \Theta(g(n))$$
- such that there exist a +ve constant “ c_1, c_2 ” and non-negative integer n_0 satisfying the constraint $c_2g(n) \leq f(n) \leq c_1g(n)$ for all $n \geq n_0$.

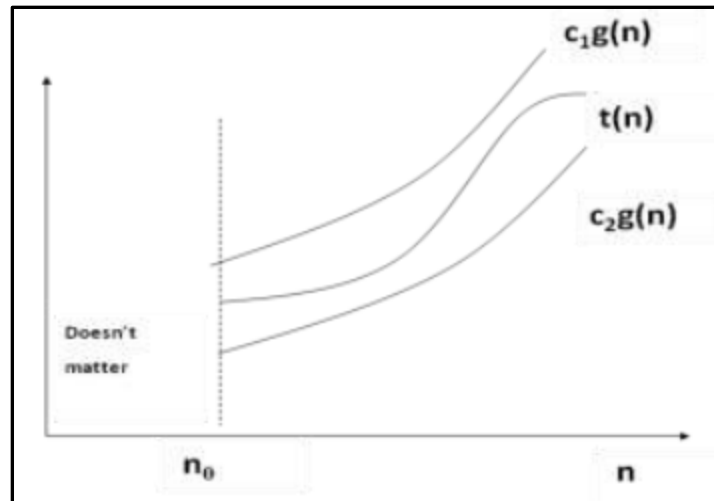


Figure 1.7

Anuradha

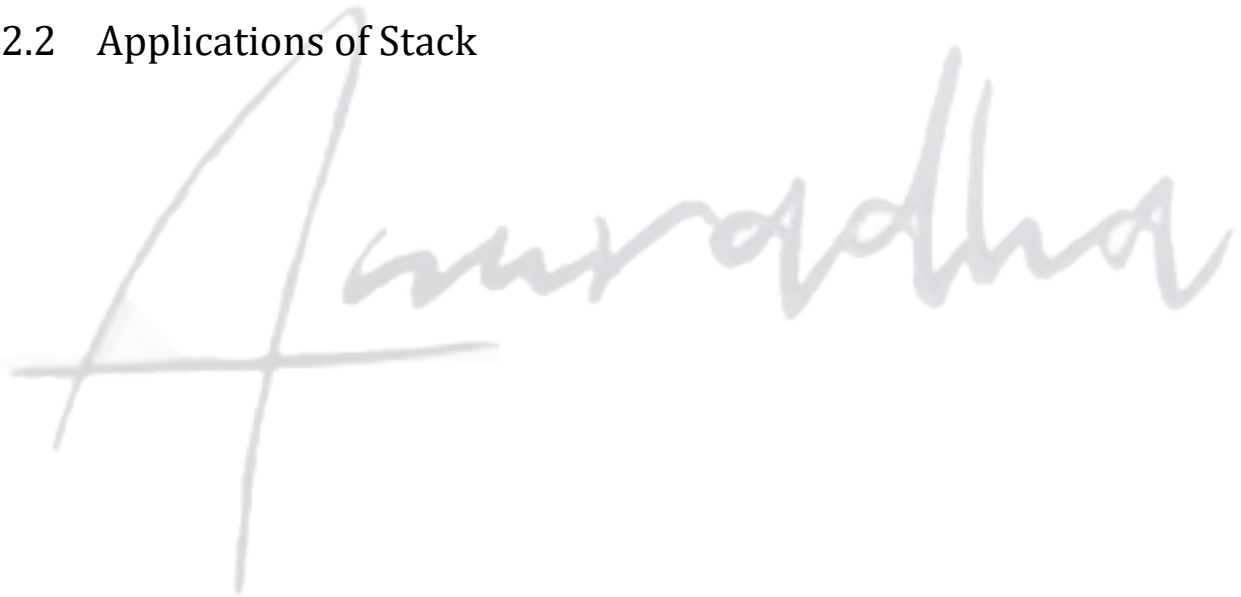
2. Stacks

CONTENTS

2.1 Introduction

1. Stack as an abstract data type
2. Representation of a Stack as an array

2.2 Applications of Stack



2.1 Introduction to Stacks

1. Stack as Abstract Data Type

- i. An abstract data type (ADT) consists of a data structure and a set of **primitive operations**. The main primitives of a stack are known as:
- ii. The stack abstract data type is defined by the following structure and operations.
- iii. A stack is structured, as described above, as an ordered collection of items where items are added to and removed from the end called the “top.”
- iv. Stacks are ordered LIFO.
- v. The stack operations are given below in the form of function are.
 - a. Stack () creates a new stack that is empty. It needs no parameters and returns an empty stack.
 - b. Push (item) adds a new item to the top of the stack. It needs the item and returns nothing.
 - c. Pop () removes the top item from the stack. It needs no parameters and returns the item. The stack is modified.

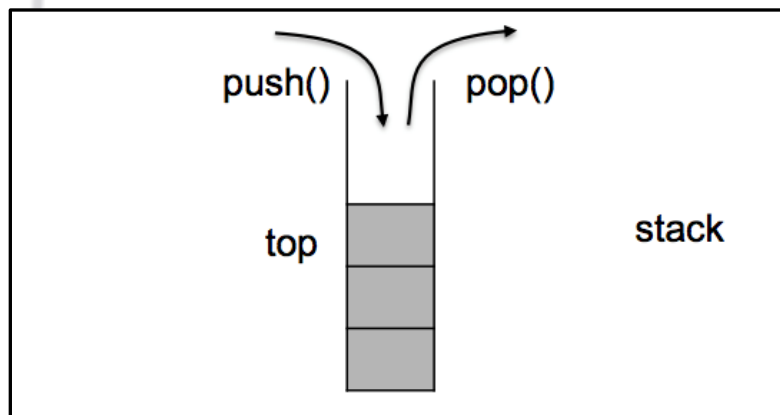


Figure 1: Stack Representation

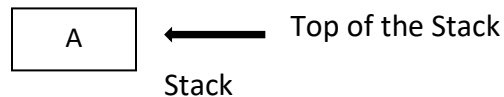
2. Representation of Stack through arrays

- i. Stack is represented using arrays as a simple array, with any variable name and the top of the stack is represented as top and initialized to -1 for the stack to be empty.
- ii. Order produced by a stack:
- iii. Stacks are linear data structures.
- iv. This means that their contexts are stored in what looks like a line (although vertically) as shown in Figure 1.
- v. This linear property, however, is not sufficient to discriminate a stack from other linear data structures.
- vi. For example, an array is a sort of linear data structure. However, you can access any element in an array--not true for a stack, since you can only deal with the element at its top.
- vii. One of the distinguishing characteristics of a stack, and the thing that makes it useful, is the order in which elements come out of a stack.

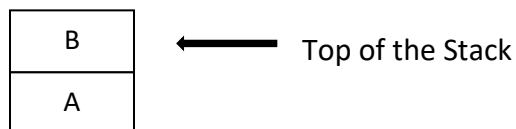
a) To start with stack empty:



b) Now, let's perform Push(stack, A), giving:

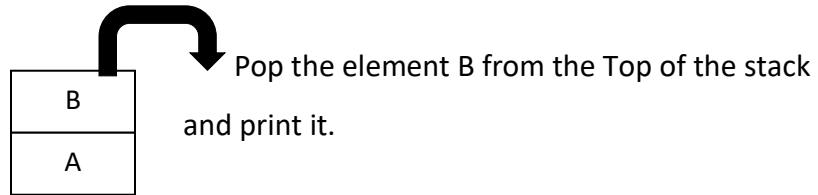


c) Again, another push operation, Push(stack, B), giving:

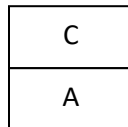


Stack

d) Now let's remove an item, letter = Pop(stack), giving:



e) And finally, one more addition, Push(stack, C), giving:



Stack

f) In the last step pop out all the elements from the stack. The final printed stack will be B, C, A

2.2 Application of Stacks

1. Reversing the list

- The application of the stack is it prints any given series of numbers whether sorted or unsorted in reverse order.
- As the stack follows Last In First Order (LIFO), it gives the series of number in reverse order.
- Given below is the program to print the stack.

WAP to PUSH and POP the elements on to the stack, without using functions

```
#include<stdio.h>
void main()
{
    int stack[5], top = -1;
    while (top <5)
    {
        top++;
        printf ("\nenter the element for the stack➡");
        scanf ("%d", &stack[top]);
    }
    while (top!=0)
    {
        printf("\nthe popped element is➡ %d", st[top]);
        top = top -1;
    }
}
```

OUTPUT:

```
enter the element for the stack➡10
enter the element for the stack➡20
enter the element for the stack➡30
enter the element for the stack➡40
enter the element for the stack➡50
the popped element is➡50
the popped element is➡40
the popped element is➡30
the popped element is➡20
the popped element is➡10
```

2. Polish Notation

- i. Infix, Postfix and Prefix notations are three different but equivalent ways of writing expressions known as Polish notation.
- ii. It is easiest to demonstrate the differences by looking at examples of operators that take two operands.

Infix notation: $X + Y$

Postfix notation (also known as "Reverse Polish notation"): $XY +$

Prefix notation (also known as "Polish notation"): $+XY$

- iii. Operators are written in-between their operands.
- iv. An expression such as $A * (B + C) / D$ is usually taken to mean by multiplication first, then followed by division and then $+$.

3. Conversion of Infix to Postfix Expression

- i. To convert an expression to its postfix notation signifies that operators have to be after the operands.
- ii. We parenthesize the expression following the BODMAS rule, i.e., exponent (^) having the highest priority, followed by multiplication (*) and division (/), and the least priority to plus (+) and (-).
- iii. If the two operators are with the same priority, i.e. (+) and (-), then we parenthesize them from left to right.
- iv. Then we move the operators to the closest closing brace for postfix notation as shown below.

Examples: $A * B + C / D \rightarrow ((A * B) + (C / D))$

Postfix Expression is: $AB*CD/+$

WAP to convert infix expression to postfix notation

```
#define SIZE 50      /* Size of Stack */
#include <ctype.h>
char s[SIZE];
int top=-1;    /* Global declarations */

push(char elem)
{
    /* Function for PUSH operation */
    s[++top]=elem;
}

char pop()
{
    /* Function for POP operation */
    return(s[top--]);
}

int pr(char elem)
{
    /* Function for precedence */
    switch(elem)
    {
        case '#': return 0;
        case '(': return 1;
        case '+':
        case '-': return 2;
        case '*':
        case '/': return 3;
    }
}

main()
{
    /* Main Program */
}
```

```
char infix[50],pofx[50],ch,elem;
int i=0,k=0;
printf("\n\nRead the Infix Expression ? ");
scanf("%s",infix);
push('#');
while( (ch=infix[i++]) != '\0')
{
    if( ch == '(') push(ch);
    else
        if(isalnum(ch)) pofx[k++]=ch;
    else
        if( ch == ')')
        {
            while( s[top] != '(')
                pofx[k++]=pop();
            elem=pop(); /* Remove ( */
        }
        else
        {
            /* Operator */
            while( pr(s[top]) >= pr(ch) )
                pofx[k++]=pop();
            push(ch);
        }
    }
while( s[top] != '#') /* Pop from stack till empty */
    pofx[k++]=pop();
pofx[k]='\0'; /* Make pofx as valid string */
printf("\n\nGiven Infix Expn: %s Postfix Expn: %s\n",infix,pofx);
}
```

OUTPUT

Read the Infix Expression? $A+B*C$

Given Infix Expn: $A+B*C$ Postfix Expn: $ABC*+$

4. Evaluating Postfix Expression

- i. To evaluate the given postfix expression, let's start with the example

$10\ 2\ 8\ *\ +\ 3\ -$

- ii. First, push (10) into the stack.



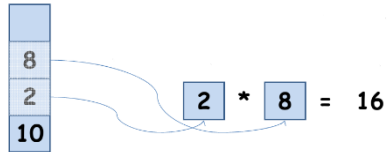
- iii. As the next element is also a number we push (2) on to the stack.



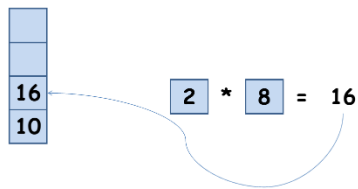
- iv. The next element is a number, so we again push (8) on to the stack.



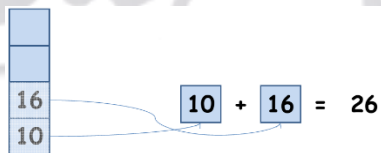
- v. The next element is an operator (*), so we pop out the top two number from the stack and perform * on the two numbers.



- vi. Then push the result on top of the stack.



- vii. The next element is a + operator, so we perform the addition on the next two elements on the stack.



- viii. Now we push (26) on to the stack.



- ix. The next element is 3, so we push (3) on to the stack.



- x. The last element is the operator –

$$26 - 3 = 23$$



WAP to convert the infix to postfix and evaluate the expression

```
#include<stdio.h>
#include <ctype.h>
#define SIZE 50      /* Size of Stack */
char s[SIZE];
int top = -1; /* Global declarations */

push(char elem) { /* Function for PUSH operation */
    s[++top] = elem;
}

char pop() { /* Function for POP operation */
    return (s[top--]);
}

int pr(char elem) { /* Function for precedence */
    switch (elem) {
        case '#':
            return 0;
        case '(':
            return 1;
        case '+':
        case '-':
            return 2;
        case '*':
        case '/':
            return 3;
    }
}

pushit(int ele){ /* Function for PUSH operation */
```



```
s[++top]=ele;
}

int popit(){          /* Function for POP operation */
    return(s[top--]);
}

main()
{ /* Main Program */
    char infx[50], pofx[50], ch, elem;
    int i = 0, k = 0, op1, op2, ele;
    printf("\n\nRead the Infix Expression  ");
    scanf("%s", infx);
    push('#');
    while ((ch = infx[i++]) != '\0') {
        if (ch == '(')
            push(ch);
        else if (isalnum(ch))
            pofx[k++] = ch;
        else if (ch == ')') {
            while (s[top] != '(')
                pofx[k++] = pop();
            elem = pop(); /* Remove ( */
        } else { /* Operator */
            while (pr(s[top]) >= pr(ch))
                pofx[k++] = pop();
            push(ch);
        }
    }
    while (s[top] != '#') /* Pop from stack till empty */
        pofx[k++] = pop();
}
```

```
pofx[k] = '\0'; /* Make pofx as valid string */
printf("\n\nGiven Infix Expn: %s Postfix Expn: %s\n", infix, pofx);

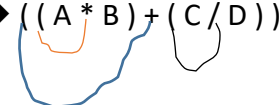
while( (ch=pofx[i++]) != '\0')
{
    if(isdigit(ch)) pushit(ch-'0'); /* Push the operand */
    else
    {
        /* Operator,pop two operands */
        op2=popit();
        op1=popit();
        switch(ch)
        {
            case '+':pushit(op1+op2);break;
            case '-':pushit(op1-op2);break;
            case '*':pushit(op1*op2);break;
            case '/':pushit(op1/op2);break;
        }
    }
}

printf("\n Given Postfix Expn: %s\n",pofx);
printf("\n Result after Evaluation: %d\n",s[top]);
}
```

5. Converting an Infix expression to Prefix Expression

i. To convert an infix expression to its prefix expression, the operators have to move to the beginning of the expression.

ii. Examples: $A * B + C / D \rightarrow ((A * B) + (C / D))$



iii. The prefix expression is $\rightarrow +*AB/CD$

WAP to convert the Infix expression to Prefix expression

```
#define SIZE 50          /* Size of Stack */
#include<string.h>
#include <ctype.h>
char s[SIZE];
int top=-1;    /* Global declarations */

push(char elem)
{
    /* Function for PUSH operation */
    s[++top]=elem;
}

char pop()
{
    /* Function for POP operation */
    return(s[top--]);
}

int pr(char elem)
{
    /* Function for precedence */
    switch(elem)
    {
        case '#': return 0;
        case ')': return 1;
```

```
case '+':
case '-': return 2;
case '*':
case '/': return 3;
}
}

main()
{
    /* Main Program */
    char infix[50],prfx[50],ch,elem;
    int i=0,k=0;
```

```
printf("\n\nRead the Infix Expression ? ");
scanf("%s",infix);
push('#');
strrev(infix);
while( (ch=infix[i++]) != '\0')
{
    if( ch == ')') push(ch);
    else
        if(isalnum(ch)) prfx[k++]=ch;
        else
            if( ch == '(')
            {
                while( s[top] != ')')
                    prfx[k++]=pop();
                elem=pop(); /* Remove ) */
            }
            else
```

```
        {    /* Operator */
            while( pr(s[top]) >= pr(ch) )
                prfx[k++]=pop();
            push(ch);
        }
    }
    while( s[top] != '#' )    /* Pop from stack till empty */
        prfx[k++]=pop();
    prfx[k]='\0';    /* Make prfx as valid string */
    strrev(prfx);
    strrev(infx);
    printf("\n\nGiven Infix Expn: %s Prefix Expn: %s\n",infx,prfx);
}
```

Example of conversions

Infix	Postfix	Prefix	Notes
A * B + C / D	A B * C D / +	+ * A B / C D	multiply A and B, divide C by D, add the results
A * (B + C) / D	A B C + * D /	/ * A + B C D	add B and C, multiply by A, divide by D
A * (B + C / D)	A B C D / + *	* A + B / C D	divide C by D, add B, multiply by A

6. On Stack conversion of infix to postfix expression

- The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.
- This is shown in a table with three columns.
- The first will show the symbol currently being read.
- The second will show what is on the stack and the third will show the current contents of the postfix string.
- The stack will be written from left to right with the 'bottom' of the stack to the left.

Example 1: $A * B + C$ becomes $A B * C +$

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	A		A
2	*	*	A
3	B	*	A B
4	+	+	A B * {pop and print the '*' before pushing the '+'}
5	C	+	A B * C
6			A B * C +

Example 2: $A + B * C$ becomes $A B C * +$

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING

1	A		A
2	+	+	A
3	B	+	A B
4	*	+ *	A B
5	C	+ *	A B C
6			A B C * +

Example 3: $A * (B + C)$ becomes $A B C + *$

A subexpression in parentheses must be done before the rest of the expression.

	current symbol	operator stack	postfix string
1	A		A
2	*	*	A
3	(* (A B
4	B	* (A B
5	+	* (+	A B
6	C	* (+	A B C
7)	*	A B C +
8			A B C + *

- Since expressions in parentheses must be done first, everything on the stack is saved and the left parenthesis is pushed to provide a marker.
- When the next operator is read, the stack is treated as though it were empty and the new operator (here the '+' sign) is pushed on. Then when the right parenthesis is read, the stack is popped until the corresponding left parenthesis is found. Since postfix expressions have no parentheses, the parentheses are not printed.

Example 4. $A - B + C$ becomes $A B - C +$

- i. When operators have the same precedence, we must consider association.
- ii. Left to right association means that the operator on the stack must be done first, while right to left association means the reverse.

	current symbol	operator stack	postfix string
1	A		A
2	-	-	A
3	B	-	A B
4	+	+	A B -
5	C	+	A B - C
6			A B - C +

- iii. In line 4, the '-' will be popped and printed before the '+' is pushed onto the stack.
- iv. Both operators have the same precedence level, so left to right association tells us to do the first one found before the second.

5. $A * B ^ C + D$ becomes $A B C ^ * D +$

- i. The exponentiation and the multiplication must be done before the addition.

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	A		A
2	*	*	A
3	B	*	A B
4	^	* ^	A B
5	C	* ^	A B C
6	+	+	A B C ^ *
7	D	+	A B C ^ * D
8			A B C ^ * D +

- ii. When the '+' is encountered in line 6, it is first compared to the '^' on top of the stack.
- iii. Since it has lower precedence, the '^' is popped and printed.
- iv. But instead of pushing the '+' sign onto the stack now, we must compare it with the new top of the stack, the '*'.
- v. Since the operator also has higher precedence than the '+', it also must be popped and printed. Now the stack is empty, so the '+' can be pushed onto the stack.

6. $A * (B + C * D) + E$ becomes $A B C D * + * E +$

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	A		A
2	*	*	A
3	(* (A
4	B	* (A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

A summary of the rules follows:

- i. Print operands as they arrive.
- ii. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
- iii. If the incoming symbol is a left parenthesis, push it on the stack.
- iv. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
- v. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
- vi. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
- vii. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
- viii. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

7. On stack conversion of Infix to Prefix

- i. First step is to reverse the expression before converting into its infix form.
- ii. The order in which the operators appear is not reversed. When the '+' is read, it has lower precedence than the '*', so the '*' must be printed first.
- iii. This is shown in a table with three columns.
- iv. The first will show the symbol currently being read.
- v. The second will show what is on the stack and the third will show the current contents of the postfix string.

- vi. The stack will be written from left to right with the 'bottom' of the stack to the left.

Example: $A*(B+C*D)+E$

	CURRENT SYMBOL	OPERATOR STACK	POSTFIX STRING
1	A		A
2	*	*	A
3	(* (A
4	B	* (A B
5	+	* (+	A B
6	C	* (+	A B C
7	*	* (+ *	A B C
8	D	* (+ *	A B C D
9)	*	A B C D * +
10	+	+	A B C D * + *
11	E	+	A B C D * + * E
12			A B C D * + * E +

WAP to convert infix to prefix expression

```
# include <stdio.h>
# include <string.h>
# define MAX 20
void infixtoprefix(char infix[20],char prefix[20]);
void reverse(char array[30]);
char pop();
void push(char symbol);
int isOperator(char symbol);
int prcd(symbol);
int top=-1;
char stack[MAX];
```

```
main() {
    char infix[20],prefix[20],temp;
    printf("Enter infix operation: ");
    gets(infix);
    infixtoprefix(infix,prefix);
    reverse(prefix);
    puts((prefix));
}
//-----
void infixtoprefix(char infix[20],char prefix[20]) {
    int i,j=0;
    char symbol;
    stack[++top]='#';
    reverse(infix);
    for (i=0;i<strlen(infix);i++) {
        symbol=infix[i];
        if (isOperator(symbol)==0) {
            prefix[j]=symbol;
            j++;
        } else {
            if (symbol=='(') {
                push(symbol);
            } else if(symbol == '(') {
                while (stack[top]!='(') {
                    prefix[j]=pop();
                    j++;
                }
                pop();
            } else {
                if (prcd(stack[top])<=prcd(symbol)) {
```

```
        push(symbol);
    } else {
        while(prcd(stack[top])>=prcd(symbol)) {
            prefix[j]=pop();
            j++;
        }
        push(symbol);
    }
    //end for else
}
}
//end for else
}
//end for for
while (stack[top]!='#') {
    prefix[j]=pop();
    j++;
}
prefix[j]='\0';
}
////-----
void reverse(char array[30]) // for reverse of the given expression {
    int i,j;
    char temp[100];
    for (i=strlen(array)-1,j=0;i+1!=0;--i,++j) {
        temp[j]=array[i];
    }
    temp[j]='\0';
    strcpy(array,temp);
    return array;
}
```

```
}  
//-----  
char pop() {  
    char a;  
    a=stack[top];  
    top--;  
    return a;  
}  
//-----  
void push(char symbol) {  
    top++;  
    stack[top]=symbol;  
}  
//-----  
int prcd(symbol) // returns the value that helps in the precedence {  
    switch(symbol) {  
        case '+':  
            case '-':  
                return 2;  
            break;  
        case '*':  
            case '/':  
                return 4;  
            break;  
        case '$':  
            case '^':  
                return 6;  
            break;  
        case '#':  
            case '(':
```

```
        case ')':
            return 1;
        break;
    }
}
//-----
int isOperator(char symbol) {
    switch(symbol) {
        case '+':
            case '-':
            case '*':
            case '/':
            case '^':
            case '$':
            case '&':
            case '(':
            case ')':
                return 1;
            break;
        default:
            return 0;
        // returns 0 if the symbol is other than given above
    }
}
```


3. Queues

CONTENTS

3.1 Introduction

1. Queues as an abstract data type
2. Representation of a Queue as an array

3.2 Types of Queue

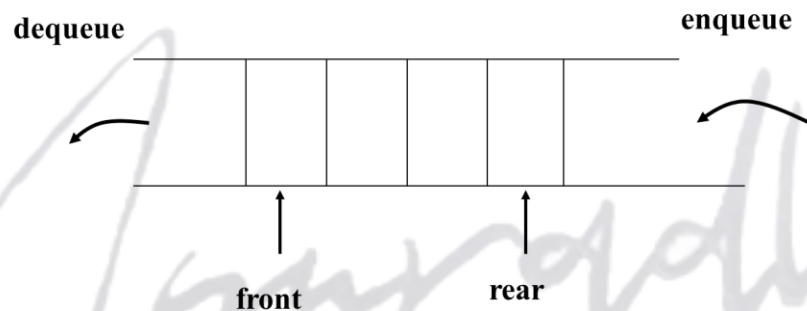
1. Circular Queue
2. Double Ended Queue
3. Priority Queue

3.3 Applications of Queue

3.1 Introduction to Queues

1. Queue as Abstract Data Type

- i. A queue is a list from which items are deleted from one end (front) and into which items are inserted at the other end (rear, or back)
- ii. It is like line of people waiting to purchase tickets:
- iii. Queue is referred to as a first-in-first-out (FIFO) data structure.
- iv. The first item inserted into a queue is the first item to leave

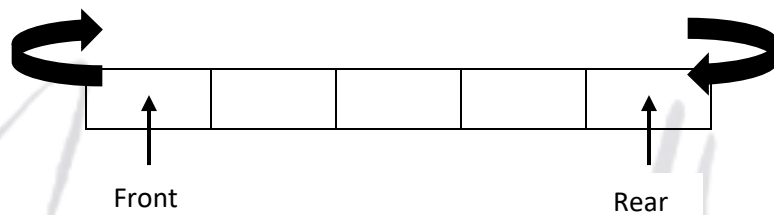


2. Operations on Queue

- i. ***createQueue()*** :Create an empty queue
- ii. ***destroyQueue()***: Destroy a queue
- iii. ***isEmpty():Boolean***: Determine whether a queue is empty
- iv. ***enqueue(in newItem:QueueItemType) throw QueueException***: Inserts a new item at the end of the queue (at the **rear** of the queue)
- v. ***dequeue() throw QueueException***
dequeue(out queueFront:QueueItemType) throw QueueException
Removes (and returns) the element at the **front** of the queue
Remove the item that was added earliest
- vi. ***getFront(out queueFront:QueueItemType) throw QueueException***:
Retrieve the item that was added earliest (without removing).

3. Representation of a Queue as an array

- i. With queues two integer indexes are required, one giving the position of the front element of the queue in the array, the other giving the position of the back element, or rather in a similar way to stacks, the position after the back element.
- ii. We add the elements from rear, and remove the elements from the front as shown in the Figure.



- iii. In queue as we delete one data element, the front is incremented by one.
- iv. We delete the elements from the end, known as the rear of the queue.
- v. The **disadvantage** of using the queue is as we delete the elements, the front moves ahead, even though the queue is empty we still cannot add the elements in the queue.

WAP to perform add and delete elements from the queue.

```
#include<stdio.h>
#include<conio.h>
void addq(int);
void delq();
int qu[5],front = -1, rear = -1, item;
void addq(int item)
{
    if ((front == rear) && (front == -1))
    {
```

```
printf("Queue is empty");
front++;
rear++;
qu[rear]= item;
}
else
{
    rear++;

qu[rear] = item;
}
}
void delq()
{
    printf(" The removed item is ➡ %d", qu[front]);
    front= front + 1;
}
void main()
{
    int ch = 1,ans =0,ele,temp;
    clrscr();
    do
    {
        printf("\n 1: Add the element in the queue");
        printf("\n 2: Delete the element from the queue");
        printf("\n 3: Display the queue");
        printf("\n 4: Exit");
        printf("\n Enter your choice ➡");
        scanf("%d",&ch);
        switch(ch)
```

```
{
    case 1: { if (rear<=5)
        {
            printf("Enter the element to be added ➡");
            scanf("%d",&ele);
            addq(ele);
            break;
        }
        else
        {
            printf("Queue Full");
            break;
        }
    }

    case 2: {
        delq();
        break;
    }

    case 3: { temp = front;
        while (front <= rear)
        {
            printf("\n The element of the queue are ➡ %d", qu[front]);
            front++;
        }
        front = temp;
        break;
    }
}
```

```
    }  
    case 4: {  
        exit();  
        break;  
    }  
}  
printf(" \n Do you want to continue(1 to continue , 0 to exit) ➔");  
scanf("%d", &ans);  
}while(ans != 0);  
}
```

OUTPUT

```
1: Add the element in the queue  
2: Delete the element from the queue  
3: Display the queue  
4: Exit  
Enter your choice ➔1  
Enter the element to be added ➔10  
Queue is empty  
Do you want to continue (1 to continue, 0 to exit) ➔1
```

1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice →1
Enter the element to be added →20
Do you want to continue (1 to continue, 0 to exit) →1

1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice →1
Enter the element to be added →30
Do you want to continue (1 to continue, 0 to exit) →1

1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue
4: Exit
Enter your choice →1
Enter the element to be added →40
Do you want to continue (1 to continue, 0 to exit) →1

1: Add the element in the queue
2: Delete the element from the queue
3: Display the queue

4: Exit

Enter your choice →3

The element of the queue are →10

The element of the queue are → 20

The element of the queue are → 30

The element of the queue are → 40

Do you want to continue (1 to continue, 0 to exit) →1

1: Add the element in the queue

2: Delete the element from the queue

3: Display the queue

4: Exit

Enter your choice →2

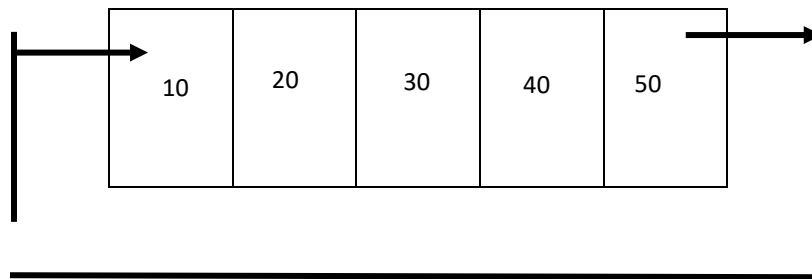
The removed item is →10

Do you want to continue (1 to continue, 0 to exit) →0

3.2 Types of Queue

1. Circular Queue

- i. Circular queue is a linear data structure. It follows FIFO principle.
- ii. In circular queue the last node is connected back to the first node to make a circle.
- iii. Circular linked list follow the First In First Out principle
- iv. Elements are added at the rear end and the elements are deleted at front end of the queue
- v. Both the front and the rear pointers points to the beginning of the array.
- vi. It is also called as “Ring buffer”.
- vii. Items can inserted and deleted from a queue in $O(1)$ time.



viii. Circular Queue can be created in three ways they are

a. Using arrays

- In arrays the range of a subscript is 0 to $n-1$ where n is the maximum size.
- To make the array as a circular array by making the subscript 0 as the next address of the subscript $n-1$ by using the formula $\text{subscript} = (\text{subscript} + 1) \% \text{maximum size}$.
- In circular queue the front and rear pointer are updated by using the above formula.
- The function below explains to add an element in the circular queue using arrays.

Step 1. Start

Step 2. if $(\text{front} == (\text{rear} + 1) \% \text{max})$

Print error "circular queue overflow "

Step 3. Else

{ $\text{rear} = (\text{rear} + 1) \% \text{max}$

$\text{Q}[\text{rear}] = \text{element};$

If $(\text{front} == -1)$ $\text{f} = 0;$

}

Step 4. Stop

- The function below explains to delete an element from circular queue using arrays.

Step 1. Start

Step 2. if ((front == rear) && (rear == -1))

Print error "circular queue underflow "

step 3. else

{ element = Q[front]

If (front == rear) front=rear = -1

Else

Front = (front + 1) % max

Step 4. Stop

WAP to insert, delete and display the elements of the circular queue.

```
#include<stdio.h>
#define SIZE 5
void insert();
void delet();
void display();
int queue[SIZE], rear=-1, front=-1, item;
main()
{
    int ch;
    do
    {
        printf("\n\n1. \tInsert\n2. \tDelete\n3. \tDisplay\n4. \tExit\n");
        printf("\nEnter your choice: ");
        scanf("%d", &ch);
        switch(ch)
        {
            case 1:
                insert();
                break;
            case 2:
                delet();
                break;
            case 3:
                display();
                break;
            case 4:
                exit(0);
            default:
```

```
        printf("\n\nInvalid choice. Please try again... \n");
    }
    } while(1);
    getch();
}

void insert()
{
    if((front==0 && rear==SIZE-1) || (front==rear+1))
        printf("\n\nQueue is full.");

    else
    {
        printf("\n\nEnter ITEM: ");
        scanf("%d", &item);
        if(rear == -1)
        {
            rear = 0;
            front = 0;
        }
        else if(rear == SIZE-1)
            rear = 0;
        else
            rear++;
        queue[rear] = item;
        printf("\n\nItem inserted: %d\n", item);
    }
}

void delet()
{
    if(front == -1)
```

```
        printf("\n\nQueue is empty. \n");
    else
    {
        item = queue[front];
        if(front == rear)
        {
            front = -1;
            rear = -1;
        }

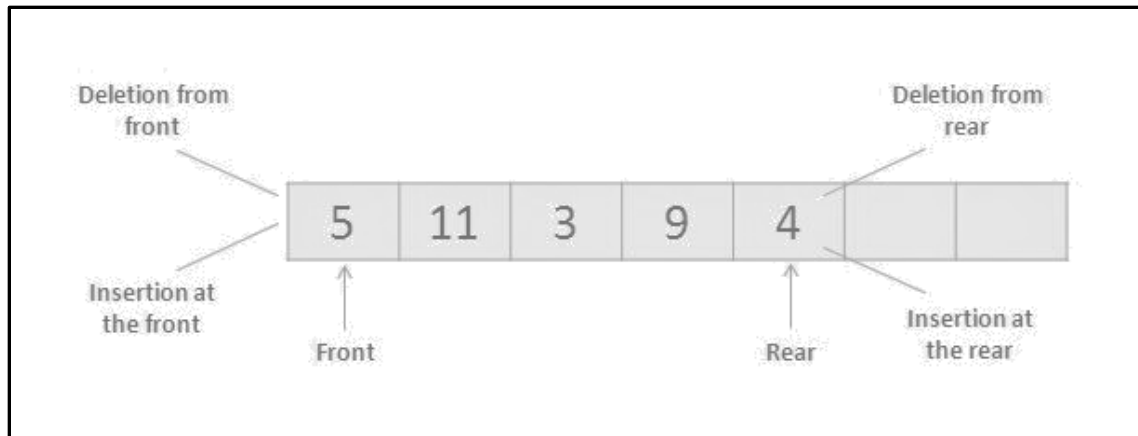
        else if(front == SIZE-1)
            front = 0;
        else
            front++;
        printf("\n\nITEM deleted: %d", item);
    }
}

void display()
{
    int i;
    if((front == -1) || (front==rear+1))
        printf("\n\nQueue is empty. \n");
    else
    { printf("\n\n");
      for(i=front; i<=rear; i++)
          printf("\t%d",queue[i]);
    }
}
```

2. Double Ended Queue

- i. Dequeue is also called as double ended queue and it allows user to perform insertion and deletion from the front and rear position.
- ii. And it can be easily implemented using doubly linked list.
- iii. On creating dequeue, we need to add two special nodes at the ends of the doubly linked list (head and tail).
- iv. And these two nodes needs to be linked between each other (initially).





Algorithm – Insertion at rear end

Step -1 : [Check for overflow]

if(rear==MAX)

Output "Queue is Overflow"

return;

Step-2 : [Insert element]

else

rear=rear+1;

q[rear]=no;

[Set rear and front pointer]

if rear=0

rear=1;

if front=0

front=1;

Step-3 : return

Algorithm: Insertion at front end

Step-1 : [Check for the front position]

if(front<=1)

Ourput "Cannot add value at front end"

return;

Step-2 : [Insert at front]

else

front=front-1;

q[front]=no;

Step-3 : Return

Algorithm: Deletion from front end

```
Step-1 [ Check for front pointer]
    if front=0
        Output " Queue is Underflow"
        return;
Step-2 [Perform deletion]
    else
        no=q[front];
        Output "Deleted element is",no
    [Set front and rear pointer]
    if front=rear
        front=0;
        rear=0;
    else
        front=front+1;
Step-3 : Return
```

Algorithm: deletion from rear end

Step-1 : [Check for the rear pointer]

if rear=0

Output "Cannot delete value at rear end"

return;

Step-2: [perform deletion]

else

no=q[rear];

[Check for the front and rear pointer]

if front= rear

front=0;

rear=0;

else

rear=rear-1;

Output "Deleted element is",no

Step-3 : Return

WAP to add, delete, and display the elements in a double ended queue.

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

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

```
#include<stdlib.h>
```

```
#define MAX 10
```

```
int q[MAX],front=0,rear=0;
```

```
void add_rear();
```

```
void add_front();
```

```
void delete_rear();
```

```
void delete_front();
```

```
void display();
```

```
void main()
{
    int ch;
    clrscr();
    do
    {
        clrscr();
        printf("\n DQueue Menu");
        printf("\n-----");
        printf("\n 1. Add Rear");
        printf("\n 2. Add Front");
        printf("\n 3. Delete Rear");
        printf("\n 4. Delete Front");
        printf("\n 5. Display");
        printf("\n 6. Exit");
        printf("\n Enter your choice:-");
        scanf("%d",&ch);

        switch(ch)
        {
            case 1:
            {
                add_rear();
                printf("\n Queue after insert at rear");
                display();
                break;
            }
            case 2:
            {
                add_front();
```

```
        printf("\n Queue after insert at front");
        display();
        break;
    }
    case 3:
    {
        delete_rear();
        printf("\n Queue after delete at rear");
        display();
        break;
    }
    case 4:
    {
        delete_front();
        printf("\n Queue after delete at front");
        display();

        break;
    }
    case 5:
    {
        display();
        break;
    }
```

case 6:

```
        {
            exit(0);
            break;
        }
    default:
        {
            printf("\n Wrong Choice\n");
        }
    }
} while(ch!=6);
}
void add_rear()
{
    int no;
    printf("\n Enter value to insert : ");
    scanf("%d",&no);
    if(rear==MAX)
    {
        printf("\n Queue is Overflow");
        return;
    }
    else
    {
        rear++;
        q[rear]=no;
        if(rear==0)
            rear=1;
        if(front==0)
            front=1;
    }
}
```

```
    }  
}  
void add_front()  
{  
    int no;  
    printf("\n Enter value to insert:-");  
    scanf("%d",&no);  
    if(front<=1)  
    {  
        printf("\n Cannot add value at front end");  
        return;  
    }  
    else  
    {  
        front--;  
        q[front]=no;  
    }  
}  
void delete_front()  
{  
    int no;  
    if(front==0)  
    {  
        printf("\n Queue is Underflow\n");  
        return;  
    }  
    else  
    {  
        no=q[front];  
        printf("\n Deleted element is %d\n",no);  
    }  
}
```

```
        if(front==rear)
        {
            front=0;
            rear=0;
        }
        else
        {
            front++;
        }
    }
}

void delete_rear()
{
    int no;
    if(rear==0)
    {
        printf("\n Cannot delete value at rear end\n");
        return;
    }
    else
    {
        no=q[rear];
        if(front==rear)
        {
            front=0;
            rear=0;
        }
        else
        {
            rear--;
        }
    }
}
```

```
        printf("\n Deleted element is %d\n",no);
    }
}
void display()
{
    int i;
    if(front==0)
    {
        printf("\n Queue is Underflow\n");
        return;
    }
    else
    {
        printf("\n Output");
        for(i=front;i<=rear;i++)
        {
            printf("\n %d",q[i]);
        }
    }
}
```

3. Priority Queue

- i. A priority queue is a collection in which items can be added at any time, but the only item that can be removed is the one with the highest priority.
- ii. Examples of the priority queue are operating system scheduler, patients waiting in the operation theater.
- iii. Operations
 - add(x): add item x.
 - remove: remove the highest priority item.

- peek: return the highest priority (without removing it).
- size: return the number of items in the priority queue.
- isEmpty: return whether the priority queue has no items.

WAP to insert and delete elements from priority queue using arrays

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

#define MAX 5

void insert_by_priority(int);
void delete_by_priority(int);
void create();
void check(int);
void display_pqueue();

int pri_que[MAX];
int front, rear;

void main()
{
    int n, ch;
```

```
printf("\n1 - Insert an element into queue");
printf("\n2 - Delete an element from queue");
printf("\n3 - Display queue elements");
printf("\n4 - Exit");

create();
while (1)
{
    printf("\nEnter your choice : ");
    scanf("%d", &ch);

    switch (ch)
    {
        case 1:
            printf("\nEnter value to be inserted : ");
            scanf("%d",&n);
            insert_by_priority(n);
            break;
        case 2:
            printf("\nEnter value to delete : ");
            scanf("%d",&n);
            delete_by_priority(n);
            break;
        case 3:
            display_pqueue();
            break;
        case 4:
            exit(0);
        default:
```

```
        printf("\nChoice is incorrect, Enter a correct choice");
    }
}

/* Function to create an empty priority queue */
void create()
{
    front = rear = -1;
}

/* Function to insert value into priority queue */
void insert_by_priority(int data)
{
    if (rear >= MAX - 1)
    {
        printf("\nQueue overflow no more elements can be inserted");
        return;
    }
    if ((front == -1) && (rear == -1))
    {
        front++;
        rear++;
        pri_que[rear] = data;
        return;
    }
    else
        check(data);
    rear++;
}
```

```
/* Function to check priority and place element */
void check(int data)
{
    int i,j;

    for (i = 0; i <= rear; i++)
    {
        if (data >= pri_que[i])
        {
            for (j = rear + 1; j > i; j--)
            {
                pri_que[j] = pri_que[j - 1];
            }
            pri_que[i] = data;
            return;
        }
    }
    pri_que[i] = data;
}

/* Function to delete an element from queue */
void delete_by_priority(int data)
{
    int i;

    if ((front== -1) && (rear== -1))
    {
        printf("\nQueue is empty no elements to delete");
    }
}
```

```
        return;
    }

    for (i = 0; i <= rear; i++)
    {
        if (data == pri_que[i])
        {
            for (; i < rear; i++)
            {
                pri_que[i] = pri_que[i + 1];
            }

            pri_que[i] = -99;
            rear--;

            if (rear == -1)
                front = -1;
            return;
        }
    }

    printf("\n%d not found in queue to delete", data);
}

/* Function to display queue elements */
void display_pqueue()
{
    if ((front == -1) && (rear == -1))
    {
        printf("\nQueue is empty");
    }
}
```

```
        return;
    }

    for (; front <= rear; front++)
    {
        printf(" %d ", pri_que[front]);
    }

    front = 0;
}
```

1 - Insert an element into queue
2 - Delete an element from queue
3 - Display queue elements
4 - Exit

Enter your choice : 1

Enter value to be inserted : 20

Enter your choice : 1

Enter value to be inserted : 45

Enter your choice : 1

Enter value to be inserted : 89

Enter your choice : 3

89 45 20

Enter your choice : 1

Enter value to be inserted : 56

Enter your choice : 3

89 56 45 20

Enter your choice : 2

Enter value to delete : 45

Enter your choice : 3

89 56 20

Enter your choice : 4

3.3 Applications

- i. When a resource is shared among multiple consumers. Examples include CPU scheduling, Disk Scheduling.
- ii. When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes. Examples
- iii. Include IO Buffers, pipes, file IO, etc.

4. Link List

CONTENTS

4.1 Introduction

1. Terminologies: node, Address, Pointer, Information, Next, Null Pointer, Empty list etc.

4.2 Type of lists

1. Linear list
2. Circular list
3. Doubly list

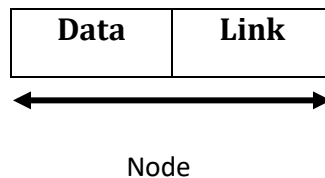
4.3 Operations on a singly linked list (only algorithm)

1. Traversing a singly linked list
2. Searching a linked list
3. Inserting a new node in a linked list
4. Deleting a node from a linked list

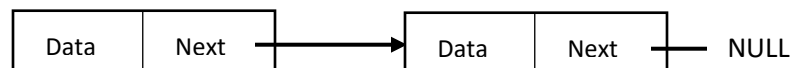
4.1 Introduction to Link List

1. Terminologies

- a. **Node:** The structure combination of data and its link to the next structure is called as node.



- b. **Address:** Stores the address of the next node of the link list.
- c. **Pointer:** A data type used to point to the address of the next node.
- d. **Information:** The data stored in the data field of the link list structure.
- e. **Next:** The link which points to the next node of the link list
- f. **Null Pointer:** Signifies the end of link list, when the next or the link pointer points to NULL
- g. **Empty list:** Signifies no data in the link list.



4.2 Type of lists

1. Linear list

- i. Linked lists stores data with structures, create a new place to store data.
- ii. It is written using a struct definition that contains variables holding information about something and that has a pointer to a struct of its same type.

- iii. Each of these individual struct in the list is commonly known as a node or element of the list.

No	Element	Explanation
1	Node	Linked list is collection of number of nodes
2	Address Field in Node	Address field in node is used to keep address of next node
3	Data Field in Node	Data field in node is used to hold data inside linked

Code to create a node in the beginning

```
#include <stdlib.h>
struct node {
    int data;
    struct node *next;
};

int main()
{
    struct node *root;
    root = (struct node *) malloc( sizeof(struct node) );
    root->next = NULL;
    root->data = 18;
}
```

To create a node and add it at the end.

```
#include <stdio.h>
#include <stdlib.h>
struct node {
    int data;
    struct node *next;
};
int main()
{
    struct node *root;
    struct node *temp;
    root = malloc( sizeof(struct node) );
    root->next = NULL;
    root->x = 18;
    temp = root;
    if ( temp != NULL )
    {
        while ( temp->next != NULL )
        {
            temp = temp->next;
        }
    }
    temp->next = malloc( sizeof(struct node) );
    temp = temp->next;
    if ( temp == 0 )
    {
        printf( "Out of memory" );
        return 0;
    }
}
```

```
}  
/* initialize the new memory */  
temp->next = NULL;  
temp->data = 22;  
return 0;  
}
```

To print the node

```
temp = root;  
if ( temp != 0 ) { /* Makes sure there is a place to start */  
    while ( temp->next != NULL )  
    {  
        printf( "%d\n", temp->data);  
        temp = temp->next;  
    }  
    printf( "%d\n", temp->x );  
}
```

Advantages

1. The advantage of a singly linked list is its ability to expand to accept virtually unlimited number of nodes in a fragmented memory environment. Linked lists are a dynamic data structure, allocating the needed memory while the program is running.
2. Insertion and deletion node operations are easily implemented in a linked list.
3. Linear data structures such as stacks and queues are easily executed with a linked list.

4. They can reduce access time and may expand in real time without memory overhead.

Disadvantages

1. The disadvantage is its speed. Operations in a singly-linked list are slow as it uses sequential search to locate a node.
2. They have a tendency to use more memory due to pointers requiring extra storage space.
3. Nodes in a linked list must be read in order from the beginning as linked lists are inherently sequential access.
4. Nodes are stored in contiguously, greatly increasing the time required to access individual elements within the list.
5. Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a back pointer.

WAP to add, delete, display and search the element in the link list

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
    int data;
    struct Node *next;
}node;
void insert(node *pointer, int data)
{
    while(pointer->next!=NULL)
    {
        pointer = pointer -> next;
    }
```

```
    pointer->next = (node *)malloc(sizeof(node));
    pointer = pointer->next;
    pointer->data = data;
    pointer->next = NULL;
}

int search(node *pointer, int key)
{
    pointer = pointer -> next; //First node is dummy node.
    while(pointer!=NULL)
    {
        if(pointer->data == key) //key is found.
        {
            return 1;
        }
        pointer = pointer -> next; //Search in the next node.
    }
    return 0;
}

void delete(node *pointer, int data)
{
    while(pointer->next!=NULL && (pointer->next)->data != data)
    {
        pointer = pointer -> next;
    }
    if(pointer->next==NULL)
    {
        printf("Element %d is not present in the list\n",data);
        return;
    }
    node *temp;
```

```
        temp = pointer -> next;
        pointer->next = temp->next;
        free(temp);
        return;
    }
void print(node *pointer)
{
    if(pointer==NULL)
    {
        return;
    }
    printf("%d ",pointer->data);
    print(pointer->next);
}
int main()
{
    node *start,*temp;
    start = (node *)malloc(sizeof(node));
    temp = start;
    temp -> next = NULL;
    printf("1. Insert\n");
    printf("2. Delete\n");
    printf("3. Print\n");
    printf("4. Search\n");
```

```
while(1)
{
    int query;
    scanf("%d",&query);
    if(query==1)
    {
        int data;
        scanf("%d",&data);
        insert(start,data);
    }
    else if(query==2)
    {
        int data;
        scanf("%d",&data);
        delete(start,data);
    }
    else if(query==3)
    {
        printf("The list is ");
        print(start->next);
        printf("\n");
    }
    else if(query==4)
    {
        int data;
        scanf("%d",&data);
        int status = seach(start,data);
        if(status)
        {
            printf("Element Found\n");
        }
    }
}
```

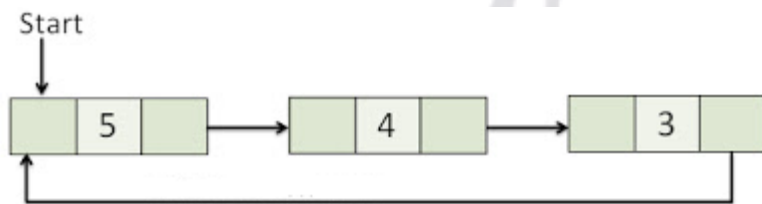


```
        }  
        else  
        {  
            printf("Element Not Found\n");  
        }  
    }  
}
```

Anuradha

2. Circular list

- a. Circular Linked List is divided into 2 Categories.
 - Singly Circular Linked List
 - Doubly Circular Linked List
- b. In Circular Linked List Address field of Last node contain address of “First Node”.
- c. In short First Node and Last Nodes are adjacent.
- d. Linked List is made circular by linking first and last node, so it looks like circular chain [shown in following diagram].
- e. Two way access is possible only if we are using “Doubly Circular Linked List” Sequential movement is possible, No direct access is allowed.



WAP to add, delete and display circular link list

```
#include<stdio.h>
#include<stdlib.h>
typedef struct Node
{
    int data;
    struct Node *next;
}node;
void insert(node *pointer, int data)
{
    node *start = pointer;
    while(pointer->next!=start)
```

```
{
    pointer = pointer -> next;
}
pointer->next = (node *)malloc(sizeof(node));
pointer = pointer->next;
pointer->data = data;
pointer->next = start;
}

int find(node *pointer, int key)
{
    node *start = pointer;
    pointer = pointer -> next; //First node is dummy node.
    while(pointer!=start)
    {
        if(pointer->data == key) //key is found.
        {
            return 1;
        }
        pointer = pointer -> next; //Search in the next node.
    }
    /*Key is not found */
    return 0;
}

void delete(node *pointer, int data)
{
    node *start = pointer;
    while(pointer->next!=start && (pointer->next)->data != data)
    {
        pointer = pointer -> next;
    }
}
```

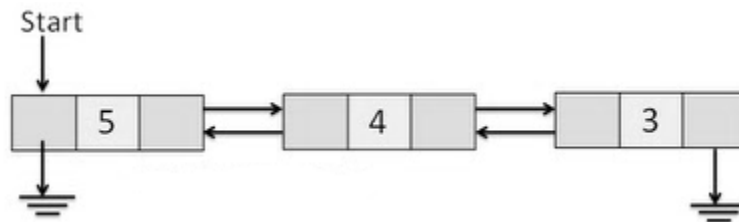
```
        if(pointer->next==start)
        {
            printf("Element %d is not present in the list\n",data);
            return;
        }
        node *temp;
        temp = pointer -> next;
        pointer->next = temp->next;
        free(temp);
        return;
    }
    void print(node *start,node *pointer)
    {
        if(pointer==start)
        {
            return;
        }
        printf("%d ",pointer->data);
        print(start,pointer->next);
    }
    int main()
    {
        node *start,*temp;
        start = (node *)malloc(sizeof(node));
        temp = start;
        temp -> next = start;
        printf("1. Insert\n");
        printf("2. Delete\n");
        printf("3. Print\n");
        printf("4. Find\n");
```

```
while(1)
{
    int query;
    scanf("%d",&query);
    if(query==1)
    {
        int data;
        scanf("%d",&data);
        insert(start,data);
    }
    else if(query==2)
    {
        int data;
        scanf("%d",&data);
        delete(start,data);
    }
    else if(query==3)
    {
        printf("The list is ");
        print(start,start->next);
        printf("\n");
    }
    else if(query==4)
    {
        int data;
        scanf("%d",&data);
        int status = find(start,data);
        if(status)
        {
```

```
        printf("Element Found\n");
    }
    else
    {
        printf("Element Not Found\n");
    }
}
}
```

3. Doubly list

- In Doubly Linked List, each node contain two address fields.
- One address field for storing address of next node to be followed and second address field contain address of previous node linked to it.
- So Two way access is possible i.e we can start accessing nodes from start as well as from last.
- Like Singly Linked List also only Linear but Bidirectional Sequential movement is possible.
- Elements are accessed sequentially, no direct access is allowed.



4.3 Operations on a singly linked list (only algorithm)

1. Create a node

```
void creat()
{
    char ch;
    do
    {
        struct node *new_node,*current;
        new_node=(struct node *)malloc(sizeof(struct node));

        printf("\nEnter the data : ");
        scanf("%d",&new_node->data);
        new_node->next=NULL;

        if(start==NULL)
        {
            start=new_node;
            current=new_node;
        }
        else
        {
            current->next=new_node;
            current=new_node;
        }
        printf("\nDo you want to creat another : ");
        ch=getche();
    }
```

```
}while(ch!='n');  
}
```

Traversing a singly linked list

```
struct node *temp; //Declare temp  
temp = start;  
while(temp!=NULL)  
{  
    printf("%d",temp->data);  
    temp=temp->next;  
}
```

Searching a linked list

```
int search(int num)  
{  
    int flag = 0;  
    struct node *temp;  
  
    temp = start;  
    while(temp!=NULL)  
    {  
        if(temp->data == num)  
            return(1); //Found  
  
        temp = temp->next;  
    }  
  
    if(flag == 0)  
        return(0); // Not found  
}
```


Inserting a new node in a linked list at the Beginning

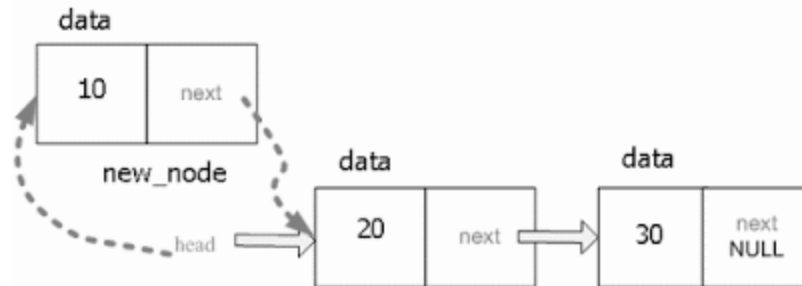
```
void insert_at_beg()
{
    struct node *new_node,*current;

    new_node=(struct node *)malloc(sizeof(struct node));

    if(new_node == NULL)
        printf("\nFailed to Allocate Memory");

    printf("\nEnter the data : ");
    scanf("%d",&new_node->data);
    new_node->next=NULL;

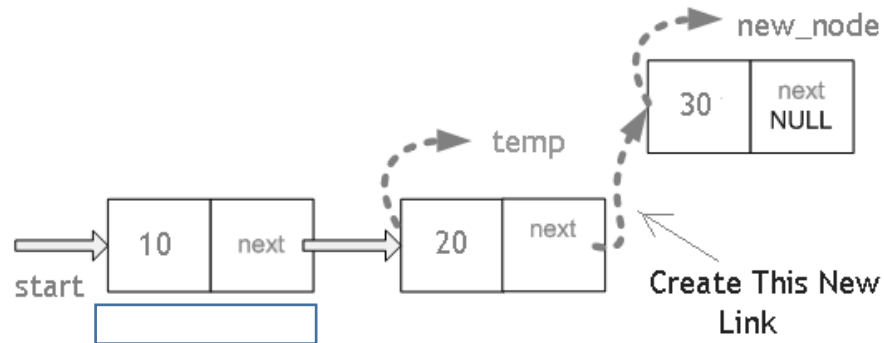
    if(start==NULL)
    {
        start=new_node;
        current=new_node;
    }
    else
    {
        new_node->next=start;
        start=new_node;
    }
}
```



Inserting a new node in a linked list at the End

```
void insert_at_end()
{
    struct node *new_node,*current;
    new_node=(struct node *)malloc(sizeof(struct node));
    if(new_node == NULL)
        printf("nFailed to Allocate Memory");
    printf("nEnter the data : ");
    scanf("%d",&new_node->data);
    new_node->next=NULL;
    if(start==NULL)
    {
        start=new_node;
        current=new_node;
    }
    else
    {
        temp = start;
        while(temp->next!=NULL)
        {
            temp = temp->next;
        }
        temp->next = new_node;
    }
}
```

```
}  
}
```



Inserting a new node in a linked list at the Middle

```
void insert_mid()  
{  
    int pos,i;  
    struct node *new_node,*current,*temp,*temp1;  
    new_node=(struct node *)malloc(sizeof(struct node));  
    printf("\nEnter the data : ");  
    scanf("%d",&new_node->data);  
    new_node->next=NULL;  
    st :  
    printf("\nEnter the position : ");  
    scanf("%d",&pos);  
    if(pos>=(length()+1))  
    {  
        printf("\nError : pos > length ");  
        goto st;  
    }  
  
    if(start==NULL)  
    {
```

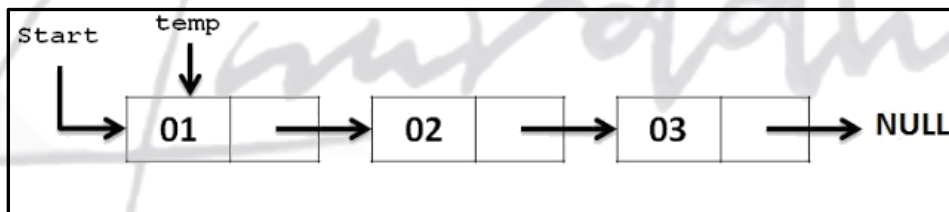
```
start=new_node;
current=new_node;
}
else
{
temp = start;
for(i=1;i< pos-1;i++)
{
temp = temp->next;
}
temp1=temp->next;
temp->next = new_node;
new_node->next=temp1;
}
}
```

Deleting a node from Beginning of a linked list

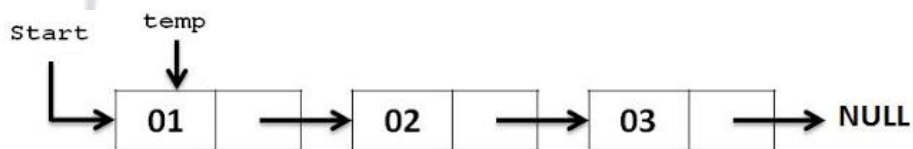
```
void del_beg()
{
    struct node *temp;

    temp = start;
    start = start->next;

    free(temp);
    printf("\nThe Element deleted Successfully ");
}
```

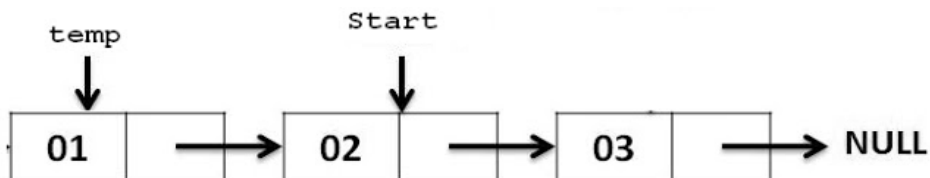


Step 1: Store Current Start in Another Temporary Pointer temp = start



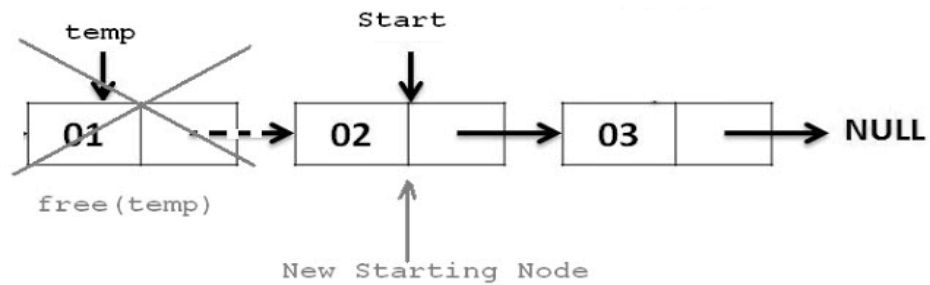
Step 2: Move start pointer one position ahead

start = start -> next



Step 3: Delete temp i.e Previous Starting Node as we have Updated Version of Start Pointer.

`free (temp)`



5. Sorting & Searching

CONTENTS

5.1 Sorting Techniques

1. Introduction
2. Selection sort
3. Insertion sort
4. Bubble sort
5. Merge sort
6. Radix sort
7. Shell sort
8. Quick sort

5.2 Searching

1. Linear search
2. Binary search

5.1 Sorting Techniques

1. Introduction

- i. Sorting is a process that organizes a collection of data into either ascending or descending order.
- ii. An internal sort requires that the collection of data fit entirely in the computer's main memory.
- iii. We use an external sort when the collection of data cannot fit in the computer's main memory all at once but must reside in secondary storage such as on a disk.
- iv. We will analyze only internal sorting algorithms.
- v. Any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted.

2. Selection Sort

- i. The list is divided into two sub lists, sorted and unsorted, which are divided by an imaginary wall.
- ii. We find the smallest i.e the minimum element from the unsorted sub list and swap it with the element at the beginning of the unsorted data.
- iii. After each selection and swapping, the imaginary wall between the two sub lists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.
- iv. Each time we move one element from the unsorted sub list to the sorted sub list, we say that we have completed a sort pass.
- v. A list of n elements requires $n-1$ passes to completely rearrange the data.

23	78	45	8	32	56	Original List
8	78	45	23	32	56	After Pass 1
8	23	45	78	32	56	After Pass 2
8	23	32	78	45	56	After Pass 3
8	23	32	45	78	56	After Pass 4
8	23	32	45	56	78	After Pass 5

Figure 1: Selection Sort

Advantages

- The main advantage of the selection sort is that it performs well on a small list.
- It is an in-place sorting algorithm, no additional temporary storage is required beyond what is needed to hold the original list.

Disadvantages

- The primary disadvantage of the selection sort is its poor efficiency when dealing with a huge list of items.
- Its performance is easily influenced by the initial ordering of the items before the sorting process.

WAP to sort the elements using selection sort

```
#include <stdio.h>
void main()
{
    int array[100], n, i, j, min, swap;
```

```
printf("Enter number of elements\n");
scanf("%d", &n);

printf("Enter %d integers\n", n);

for ( i = 0 ; i < n ; i++ )
    scanf("%d", &array[i]);
    i=0
while(i<n)
{
min = a[i];
for(j=i+1; j < 5; j++)
{
if(min > a[j])
{
min = a[j];
temp = a[i];
a[i] = a[j];
a[j] = temp;
}
}
i++;
}

printf("Sorted list in ascending order:\n");
for ( i = 0 ; i < n ; i++ )
    printf("%d\n", array[i]);
}
```

3. Insertion Sort

- i. It always maintains two zones in the array to be sorted: sorted and unsorted.
- ii. At the beginning the sorted zone consist of one element (the first element of array that we are sorting).
- iii. On each step the algorithms expand the sorted zone by one element inserting the first element from the unsorted zone in the proper place in the sorted zone and shifting all larger elements one slot down.

Advantages

- i. The insertion sorts repeatedly scans the list of items, each time inserting the item in the unordered sequence into its correct position.
- ii. The main advantage of the insertion sort is its simplicity. It also exhibits a good performance when dealing with a small list. The insertion sort is an in-place sorting algorithm so the space requirement is minimal.

Disadvantage

- i. The disadvantage of the insertion sort is that it does not perform as well as other, better sorting algorithms.
- ii. With n -squared steps required for every n element to be sorted, the insertion sort does not deal well with a huge list.
- iii. Therefore, the insertion sort is particularly useful only when sorting a list of few items.

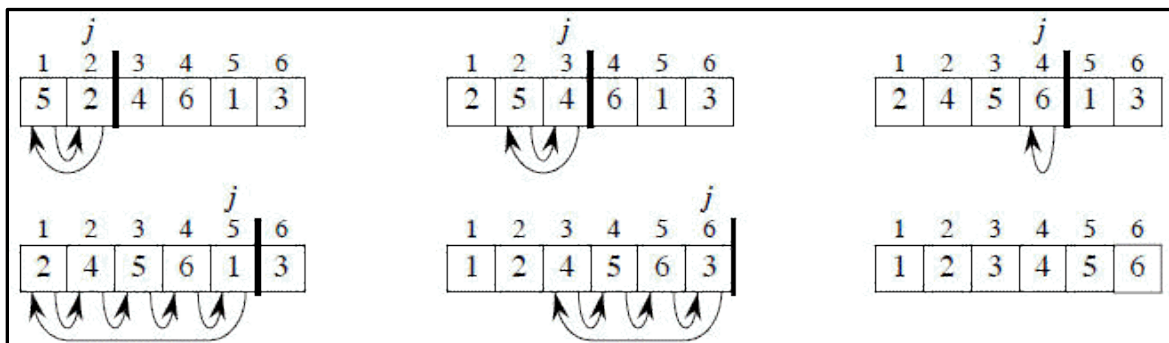


Figure 2: Insertion Sort

WAP to sort the elements using insertion sort.

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int i, j, temp,a[10];
    printf("enter the elements ➡");
    for(i = 0; i <= 10; i++)
    {
        scanf("%d", &a[i]);
    }
    for (i = 0; i <= 10; i++)
    {
        for(j = i +1; j <= 10; j++)
        {
            if(a[i] > a[j])
            {
                temp = a[i];
                while( j != i)
                {
                    a[j] = a[j-1];
                    j--;
                }
                a[i] = temp;
            }
        }
    }
    printf (" the sorted array is ➡");
    for( i = 0; i <=10; i++)
```

```
printf("%d \n", a[i]);  
}
```

4. Bubble Sort

- i. Bubble sort algorithms cycle through a list, analyzing pairs of elements from left to right, or beginning to end.
- ii. If the leftmost element in the pair is less than the rightmost element, the pair will remain in that order. If the rightmost element is less than the leftmost element, then the two elements will be switched.
- iii. This cycle repeats from beginning to end until a pass in which no switch occurs.

Advantages

- i. The bubble sort requires very little memory other than that which the array or list itself occupies.
- ii. The bubble sort is comprised of relatively few lines of code.
- iii. With a best-case running time of $O(n)$, the bubble sort is good for testing whether or not a list is sorted or not. Other sorting methods often cycle through their whole sorting sequence, which often have running times of $O(n^2)$ or $O(n \log n)$ for this task.
- iv. The same applies for data sets that have only a few items that need to be swapped a few times.

Disadvantages

- i. The main disadvantage of the bubble sort method is the time it requires.
- ii. With a running time of $O(n^2)$, it is highly inefficient for large data sets.

WAP to sort the elements in the array using bubble sort.

```
#include <stdio.h>

int main()
{
    int array[100], n, i, j, temp;

    printf("Enter number of elements\n");
    scanf("%d", &n);

    printf("Enter %d integers\n", n);

    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);

    for (i = 0; i < ( n - 1 ); i++)
    {
        for (j = i+1; j < n; j++)
        {
            if (array[i] > array[j]) /* For decreasing order use < */
            {
                temp    = array[i];
                array[i] = array[j];
                array[j] = temp;
            }
        }
    }

    printf("Sorted list in ascending order:\n");
```

```
for ( c = 0 ; c < n ; c++ )  
    printf("%d\n", array[c]);  
  
return 0;  
}
```

5. Merge Sort

- i. It follows the fundamental of dividing the elements and sorting, as sorting is easier in smaller bits of elements.
- ii. If the sub list is 1 in length, then the sub list is sorted.
- iii. If the list is more than one in length, then divide the unsorted list into roughly two parts.
- iv. Keep dividing the sub lists until each one is only 1 item in length.
- v. Now merge the sub-lists back into a list twice their size, at the same time sorting each items into order.
- vi. Keep merging the sub list until the list is complete once again.

Advantage

- i. Good for sorting slow-access data.
- ii. It is excellent for sorting data that are normally accessed sequentially.

Disadvantage

- i. If the list is N long, then it takes $2*N$ memory space to sort the elements.
- ii. For large data its time and space complexity is of the order of $n \log n$.

WAP to sort the elements in the array using merge sort.

```
#include<stdio.h>
#define MAX 50

void mergeSort(int arr[],int low,int mid,int high);
void partition(int arr[],int low,int high);

int main(){

    int merge[MAX],i,n;

    printf("Enter the total number of elements: ");
    scanf("%d",&n);

    printf("Enter the elements which to be sort: ");
    for(i=0;i<n;i++){
        scanf("%d",&merge[i]);
    }
    partition(merge,0,n-1);
```

```
    printf("After merge sorting elements are: ");
    for(i=0;i<n;i++){
        printf("%d ",merge[i]);
    }

    return 0;
}
```



```
void partition(int arr[],int low,int high){

    int mid;

    if(low<high){
        mid=(low+high)/2;
        partition(arr,low,mid);
        partition(arr,mid+1,high);
        mergeSort(arr,low,mid,high);
    }
}

void mergeSort(int arr[],int low,int mid,int high){

    int i,m,k,l,temp[MAX];

    l=low;
    i=low;
    m=mid+1;

    while((l<=mid)&&(m<=high)){

        if(arr[l]<=arr[m]){
            temp[i]=arr[l];
            l++;
        }
        else{
            temp[i]=arr[m];
            m++;
        }
    }
}
```

```
    }  
    i++;  
}  
if(l>mid){  
    for(k=m;k<=high;k++){  
        temp[i]=arr[k];  
        i++;  
    }  
}  
else{  
    for(k=l;k<=mid;k++){  
        temp[i]=arr[k];  
        i++;  
    }  
}  
  
for(k=low;k<=high;k++){  
    arr[k]=temp[k];  
}  
}
```

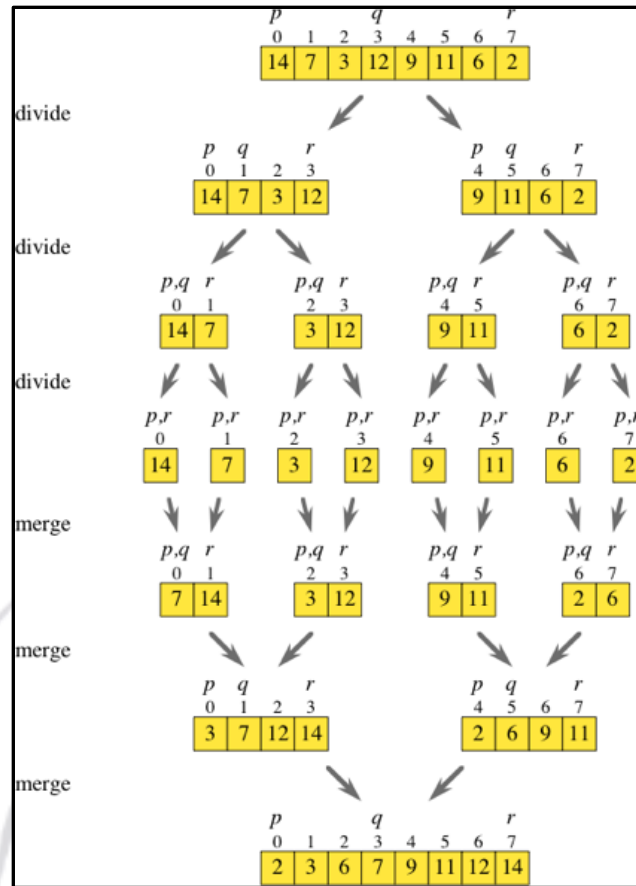


Figure 3: Merge Sort

6. Radix Sort

- i. The array for the bucket sort is divided in to 10 buckets from 0 to 9.
- ii. This sort is commonly also known as bucket sort.
- iii. The numbers are sorted on the LSB (Least Significant Bit) of any number for the first pass.
- iv. The graphical representation of the same is as shown below

Bucket	0	1	2	3	4	5	6	7	8	9
Content		1 81	–	–	64 4	25	36 16	–	–	9 49

Figure 4: Radix Sort Pass 1

Pass 1: 1, 81, 64, 4, 25, 36, 16, 9, and 49.

- v. This sort is implemented for the good functionality for a three bit number.
- vi. After the Pass 1 print the list and continue with the same on the middle bit and put in the respective buckets.

Bucket	0	1	2	3	4	5	6	7	8	9
Content	01 04 09	16	25	36	49		64		81	

Figure 5: Radix Sort Pass 2

- vii. Then print the pass 2.

Pass2: 01, 04, 09, 16, 25, 36, 49, 64, 81

7. Shell Sort

- i. The shell sort, sometimes called the “diminishing increment sort,” improves on the insertion sort by breaking the original list into a number of smaller sub lists, each of which is sorted using an insertion sort.
- ii. The unique way that these sub lists are chosen is the key to the shell sort.
- iii. Instead of breaking the list into subsists of contiguous items, the shell sort uses an increment i , sometimes called the gap, to create a sub list by choosing all items that are i items apart.
- iv. This can be seen in Figure 6. This list has nine items.
- v. If we use an increment of three, there are three sub lists, each of which can be sorted by an insertion sort.
- vi. After completing these sorts, we get the list shown in Figure 7.
- vii. Although this list is not completely sorted, something very interesting has happened. By sorting the sub lists, we have moved the items closer to where they actually belong.

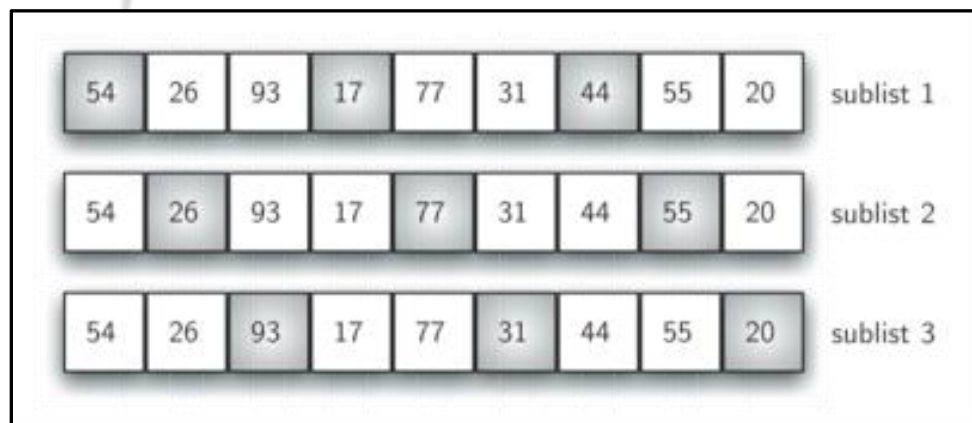


Figure 6: Shell sort with increment of 3

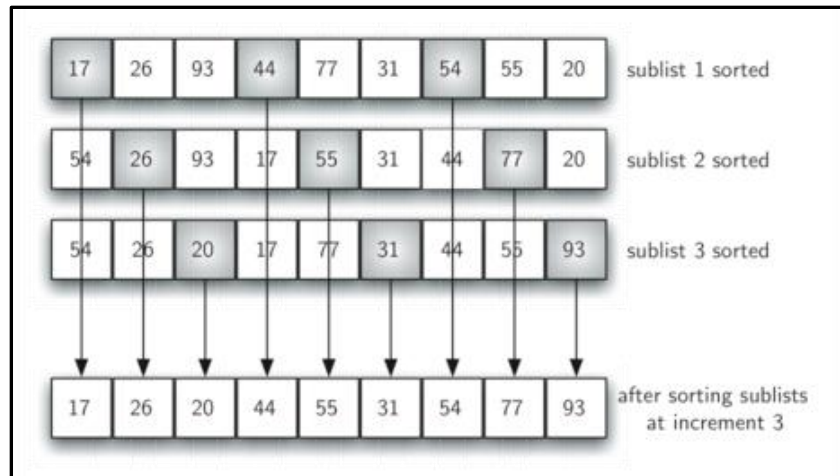


Figure 7: Shell Sort after sorting each sub list

- viii. Figure 8 shows a final insertion sort using an increment of one; in other words, a standard insertion sort.
- ix. Note that by performing the earlier sub list sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order.
- x. For this case, we need only four more shifts to complete the process.

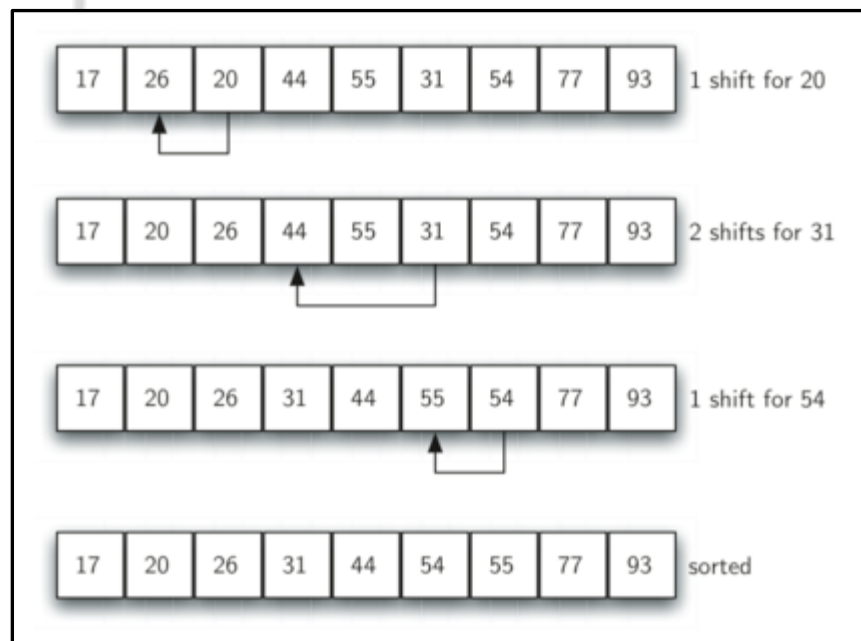


Figure 8: sorted list after increment of 1

8. Quick Sort

- i. The quicksort algorithm partitions an array of data into items that are less than the pivot and those that are greater than or equal to the pivot item.
- ii. The first step is to create the partitions, in which a random pivot item is created, and the partition algorithm is applied to the array of items.
- iii. This initial step is the most difficult part of the Quicksort algorithm.

Advantages

- i. The efficient average case compared to any aforementioned sort algorithm, as well as the elegant recursive definition, and the popularity due to its high efficiency.
- ii. The quick sort produces the most effective and widely used method of sorting a list of any item size.

Disadvantages

- i. The difficulty of implementing the partitioning algorithm and the average efficiency for the worst case scenario, which is not offset by the difficult implementation.
- ii. Quicksort works by "partitioning" the array to be sorted, then recursively sorting each partition. Here is the pseudocode.

```
procedure QuickSort (array A, int L, int N)
if L < N
M := Partition (A, L, N)
QuickSort (A, L, M - 1)
QuickSort (A, M + 1, N)
```

```
endif  
endprocedure
```

```
Int function Partition (array A, int L, int N)  
select M, where  $L \leq M \leq N$   
reorder  $A(L) \dots A(N)$  so that  $I < M$  implies  $A(I) \leq A(M)$ , and  $I > M$   
implies  $A(I) \geq A(M)$   
return M  
endfunction
```

Name	Average	Worst	Space	Stable	Method
Bubble sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(1)$	No	Selection
Insertion sort	$O(n^2)$	$O(n^2)$	$O(1)$	Yes	Insertion
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n)$	Yes	Merging
Quicksort	$O(n \log n)$	$O(n^2)$	$O(1)$	No	Partitioning
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(1)$	No	Selection

Table 1: Time and Space complexity

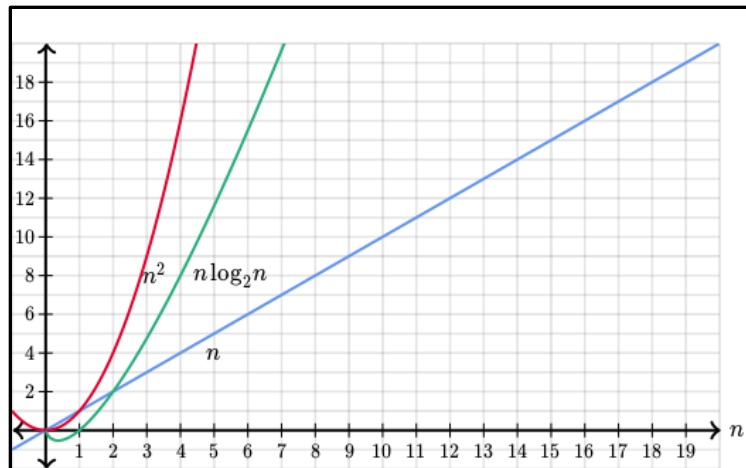


Figure 9: Graphical representation of complexity

5.2 Searching

1. Linear Search

- This is the simplest searching technique available.
- To search whether an element is present or absent in the array, we compare it with each and every other element in the array.
- If the element is found after comparison then it is printed as element found, else element not found or absent in the array.

Advantage

- This is the simplest searching algorithm
- The space complexity is very less.
- Efficient for small data, if the dataset created is small.

Disadvantage

- The time complexity increases.
- The number of comparisons become large as n , for the large set of array.

WAP to check whether an element is found in the array or not.

```
#include<stdio.h>

int main(){
    int a[10],i,n,m,c=0;
    printf("Enter the size of an array: ");
    scanf("%d",&n);
    printf("Enter the elements of the array: ");
    for(i=0;i<=n-1;i++){
        scanf("%d",&a[i]);
    }
    printf("Enter the number to be search: ");
    scanf("%d",&m);
    for(i=0;i<=n-1;i++){
        if(a[i]==m){
            c=1;
            break;
        }
    }
    if(c==0)
        printf("The number is not in the list");
    else
        printf("The number is found");

    return 0;
}
```

WAP to find the element present in the array and also print its location.

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

```
int main()
{
    int array[100], search, c, n;
    printf("Enter the number of elements in array\n");
    scanf("%d",&n);
    printf("Enter %d integer(s)\n", n);
    for (c = 0; c < n; c++)
        scanf("%d", &array[c]);
    printf("Enter the number to search\n");
    scanf("%d", &search);
    for (c = 0; c < n; c++)
    {
        if (array[c] == search) /* if required element found */
        {
            printf("%d is present at location %d.\n", search, c+1);
            break;
        }
    }
    if (c == n)
        printf("%d is not present in array.\n", search);

    return 0;
}
```

2. Binary Search

- i. A binary search divides a range of values into halves, and continues to narrow down the field of search until the unknown value is found.
- ii. It is the classic example of a "divide and conquer" algorithm.

WAP to search the element in the array using binary search

```
#include <stdio.h>
void main ()
{
    int c, first, last, middle, n, search, array[100];
    printf("Enter number of elements\n");
    scanf("%d",&n);
    printf("Enter %d integers\n", n);
    for (c = 0; c < n; c++)
        scanf("%d",&array[c]);
    printf("Enter value to find\n");
    scanf("%d", &search);
    first = 0;
    last = n - 1;
    middle = (first+last)/2;
    while (first <= last) {
        if (array[middle] < search)
            first = middle + 1;
        else if (array[middle] == search) {
            printf("%d found at location %d.\n", search, middle+1);
            break;
        }
        else
            last = middle - 1;
    }
```

```
        middle = (first + last)/2;
    }
    if (first > last)
        printf("Not found! %d is not present in the list.\n", search);
}
```



6. Trees and Graph

CONTENTS

6.1 Introduction to Trees,

1. Definitions & Tree terminologies,
2. Binary tree representation,
3. Operations on binary tree,
4. Traversal of binary trees,
5. Binary search tree,
6. Threaded Binary tree,
7. Expression tree,
8. Application of Trees

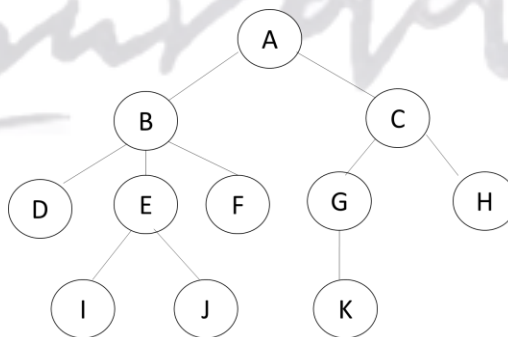
6.2 Introduction to Graph

1. Introduction Graph Terminologies,
2. Graph Representation
3. Type of graphs
4. Graph traversal
 - i. Depth first search (DFS)
 - ii. Breadth First search (BFS)
5. Minimum Spanning Tree - Prim's & Kruskal's
6. Shortest Path Algorithm – Dijkstra's Algorithm.
7. Applications of graph

6.1 Introduction to Trees

1. Definitions & Tree terminologies

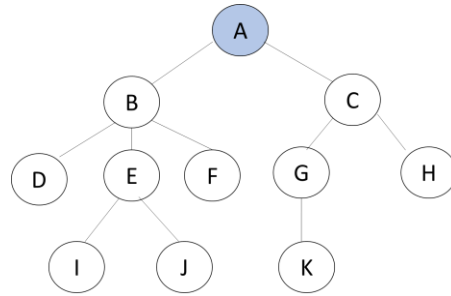
- i. Tree data structure is a collection of data (Node) which is organized in hierarchical structure.
- ii. In a tree data structure, every individual element is called as Node.
- iii. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.
- iv. In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.
- v. Tree with 11 nodes and 10 edges.
- vi. The tree with 'n' nodes will always have 'n-1' edges. Every individual element is called as a 'NODE'



Terminologies are

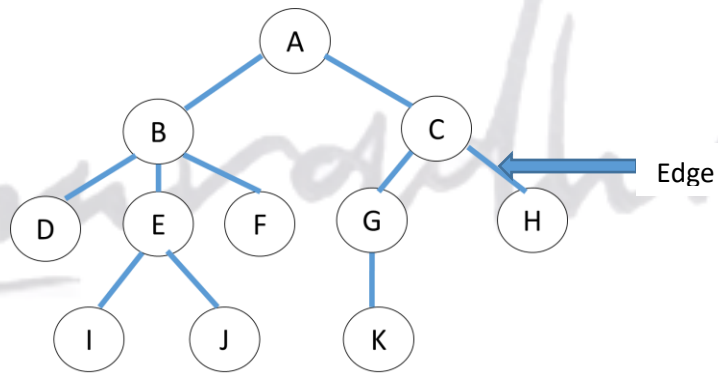
i. Root

- a. In a tree data structure, the first node is called as Root Node.
- b. Every tree must have root node.
- c. The root node is the origin of a tree.
- d. There are never multiple root nodes in a tree.
- e. 'A' is the Root Node.



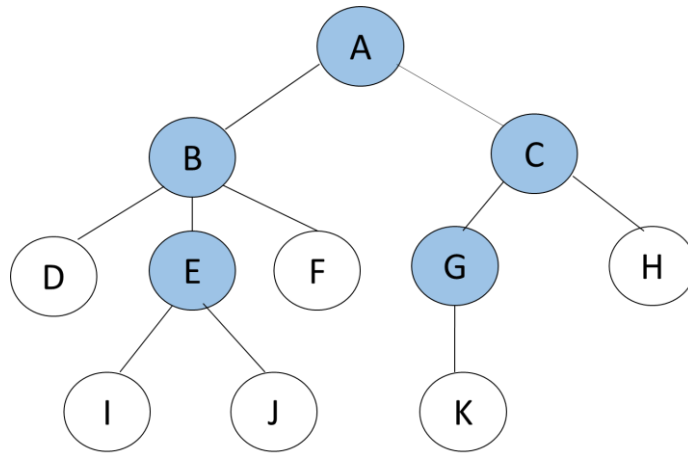
ii. Edge

- The connecting link between any two nodes is called as EDGE.
- In a tree with 'N' number of nodes there will be a maximum of 'N-1' number of edges.
- Blue highlighted lines are the edges.



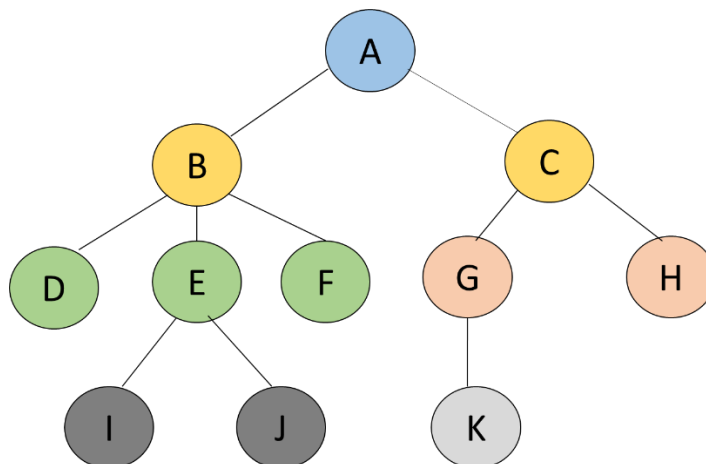
iii. Parent

- In a tree, the node which is predecessor of any node is called as PARENT NODE.
- The node which has branch from it to any other node is called as parent node.
- Parent node can also be defined as "The node which has child / children".
- "A", "B", "C", "E", "G" are parent nodes.



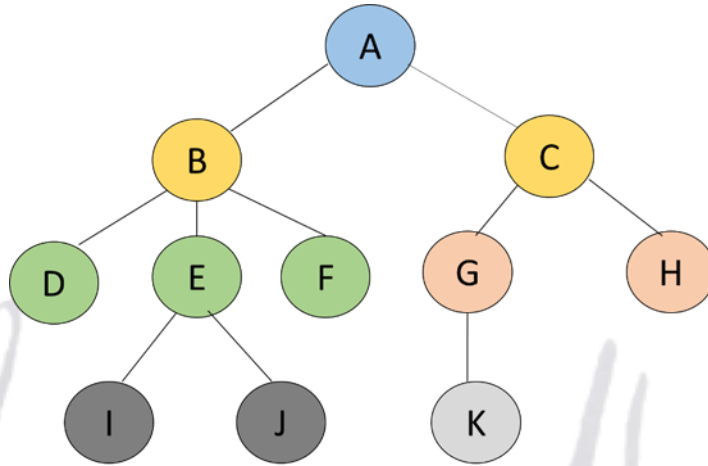
iv. Child

- In a tree, the node which is descendant of any node is called as CHILD Node.
- Node which has a link from its parent node is called as child node, "B", and "C" are child of "A".
- A parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.
- The child of each node are highlighted with different color.



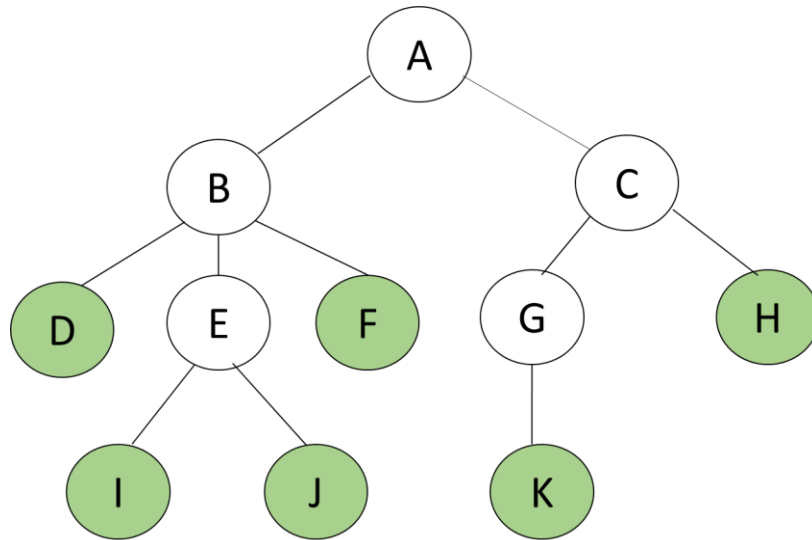
v. Siblings

- a. Nodes which belong to same Parent are called as SIBLINGS.
- b. "B" and "C" are siblings. "D", "E", and "F" are siblings.



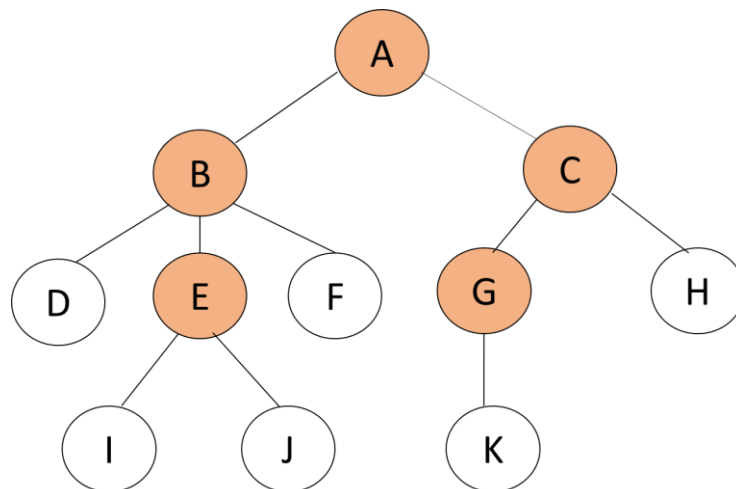
vi. Leaf

- a. The node which does not have a child is called as LEAF Node, i.e a leaf is a node with no child.
- b. The leaf nodes are also called as External Nodes.
- c. External node is also a node with no child. I
- d. A leaf node is also called as 'Terminal' node.
- e. "D", "F", "H", "I", "J", and "K" are leaf nodes.



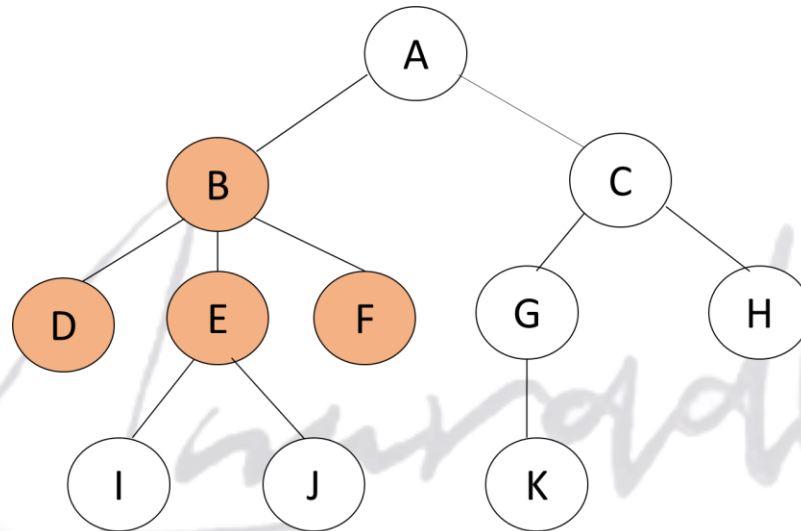
vii. Internal Nodes

- a. The node which has at least one child is called as INTERNAL Node, an internal node is a node with at least one child.
- b. Nodes other than leaf nodes are called as Internal Nodes.
- c. The root node is also said to be Internal Node if the tree has more than one node.
- d. Internal nodes are also called as 'Non-Terminal' nodes.
- e. "A", "B", "C", "E", and "G" are internal nodes.



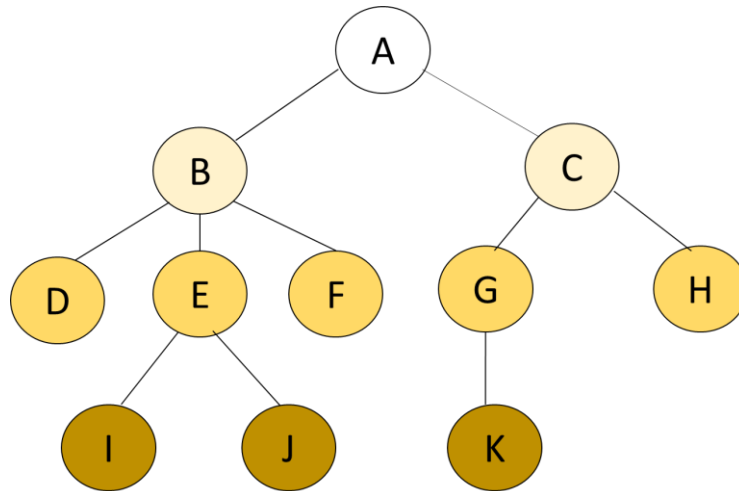
viii. Degree

- a. The total number of children of a node is called as DEGREE of that Node.
- b. The highest degree of a node among all the nodes in a tree is called as 'Degree of Tree'.
- c. "B" has a degree 3, as it has "D", "E" and "F" as its children.



ix. Level

- a. The root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2.
- b. Each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level.
- c. "A" is at level 0, "B" and "C" is at level 1.



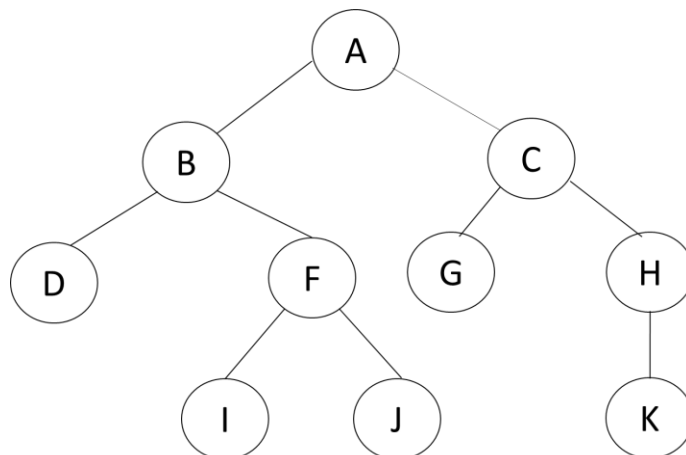
2. Application of Trees

- a. Router algorithm.
- b. Social networking based application.
- c. File system, Directory implementation.
- d. Maps typically uses tree data structure.

3. Binary Tree

- a. In a tree, every node can have any number of children.
- b. Binary tree is a tree in which every node can have a maximum of 2 children, left child and right child.

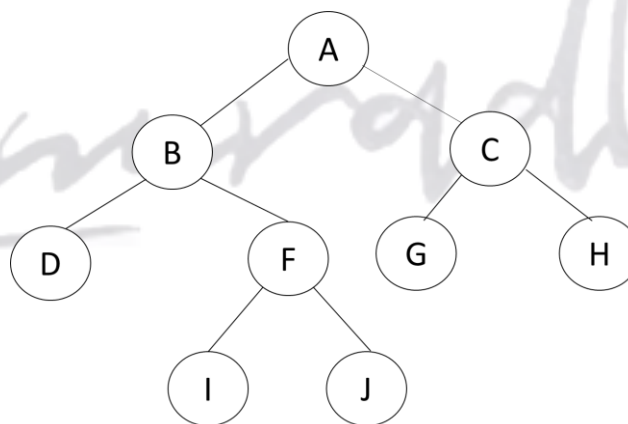
Example



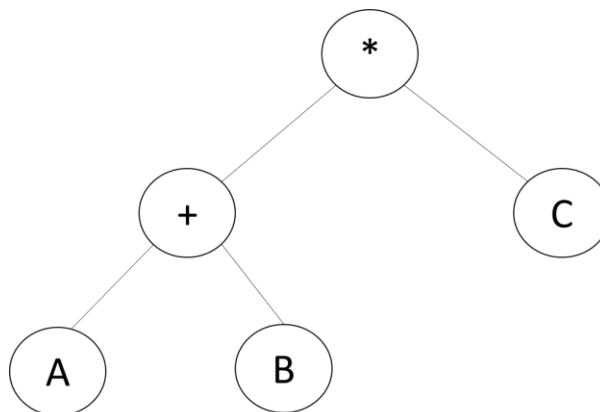
There are different types of binary trees and they are

i. **Strictly Binary Tree**

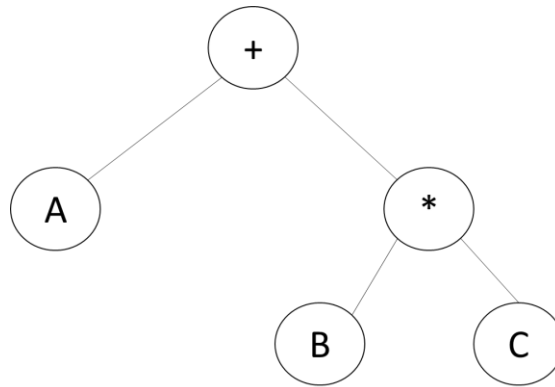
- a. Every node can have a maximum of two children.
- b. In a strictly binary tree, every node **should have** exactly two children or none.
- c. That means every internal node must have exactly two children.
- d. A strictly Binary Tree can be defined as A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree
- e. Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree



- f. Strictly binary tree data structure is used to represent mathematical expressions as $(A+B) * C$

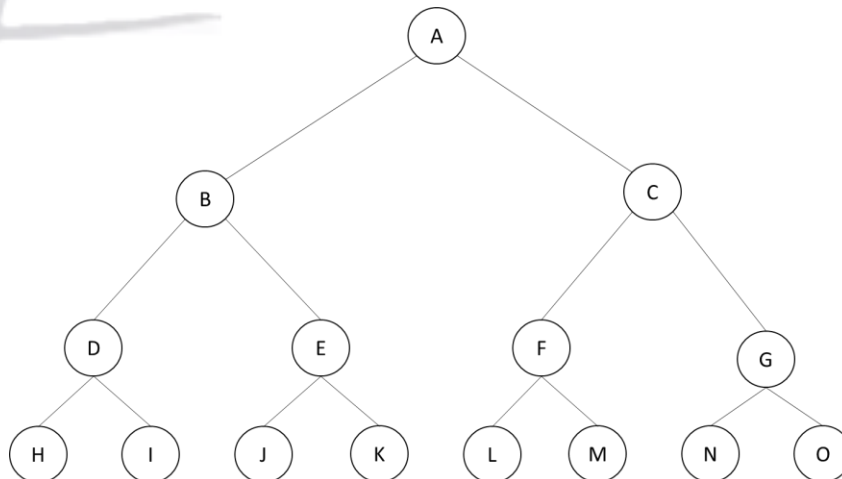


g. $A+B*C$



ii. **Complete Binary Tree**

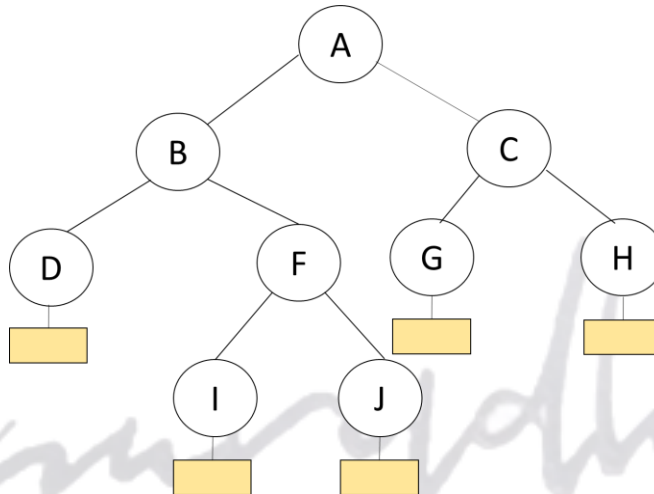
- a. Every node can have a maximum of two children.
- b. In a complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2 level number of nodes.
- c. For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.



- d. Complete binary tree is also called as Perfect Binary Tree

iii. Extended Binary Tree

- a. A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.
- b. The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



4. Operations on binary tree

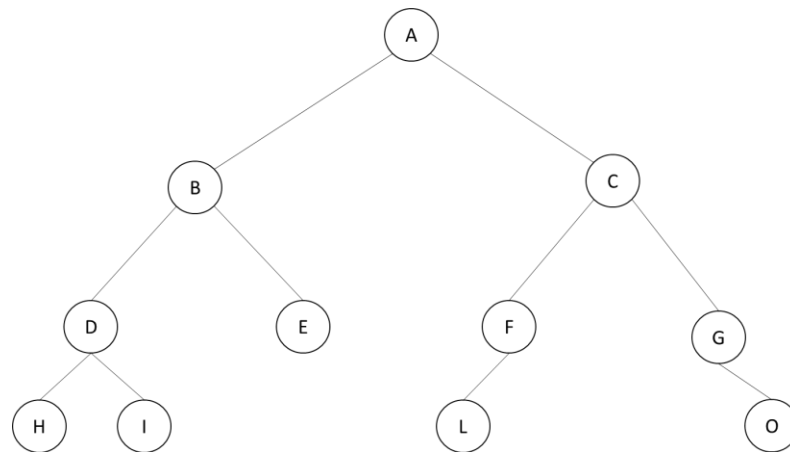
- i. Insert – Inserts an element in a tree/create a tree.
- ii. Search – Searches an element in a tree.
- iii. Preorder Traversal – Traverses a tree in a pre-order manner.
- iv. Inorder Traversal – Traverses a tree in an in-order manner.
- v. Postorder Traversal – Traverses a tree in a post-order manner.

5. Traversal of binary trees

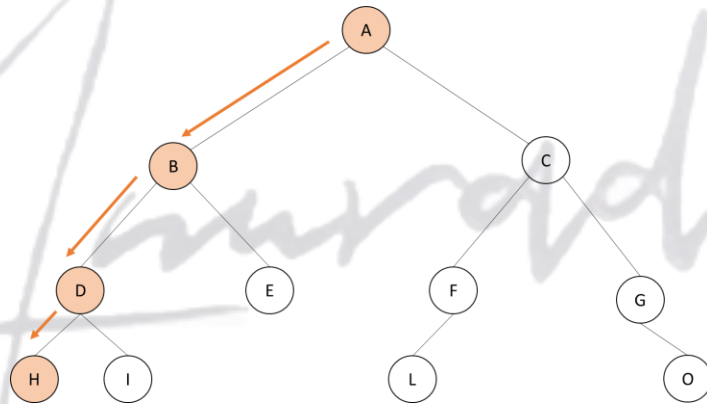
There are three types of binary tree traversals.

- i. In - Order Traversal
- ii. Pre - Order Traversal
- iii. Post - Order Traversal

Consider the following binary tree...

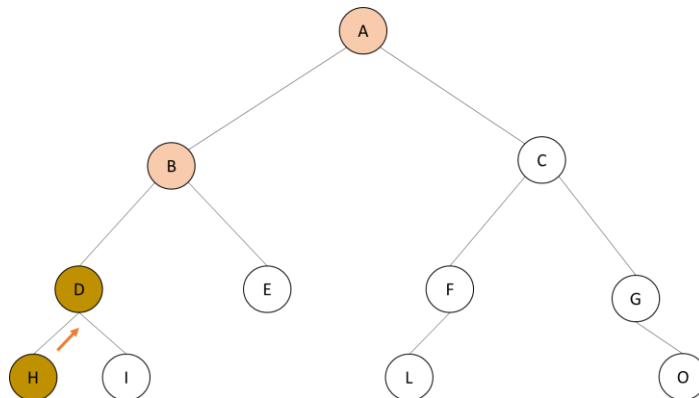


i. In - Order Traversal - LeftChild ➡ Data ➡ RightChild (LDR)

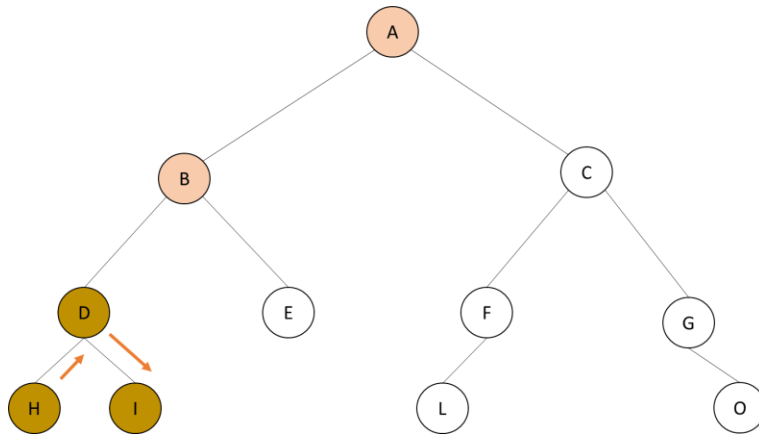


Step 1: Traverse the tree till we reach the Left most child and then print it. So H will be printed.

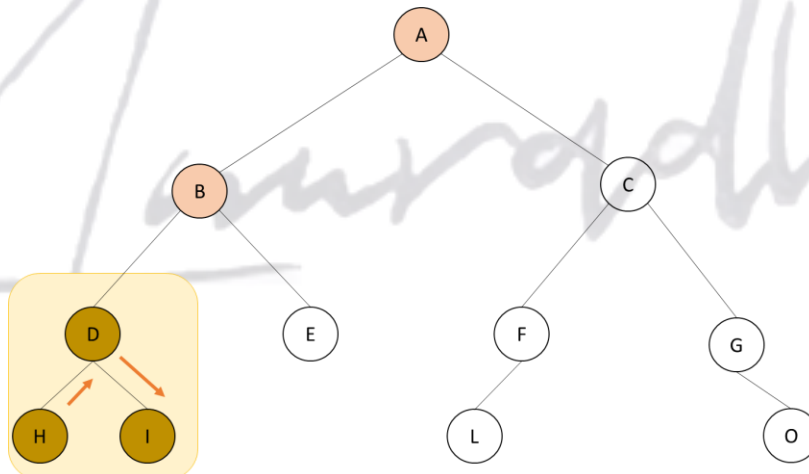
Step 2: Move up towards the parent of H, and print the data D.



Step 3: Now, traverse towards the Right child and print it. The order will be now H,D,I.

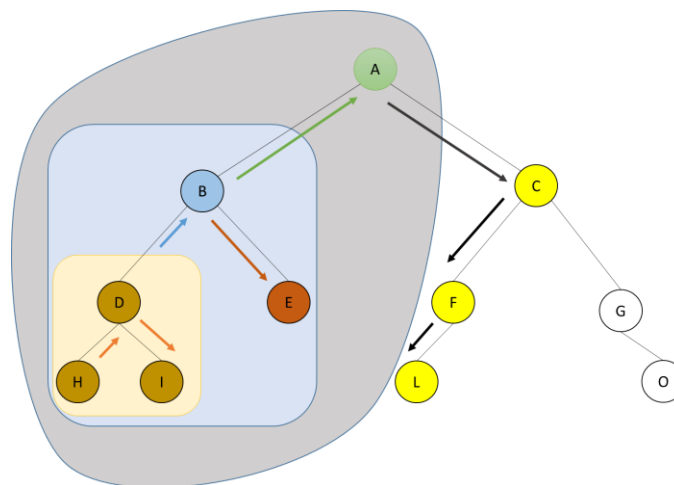
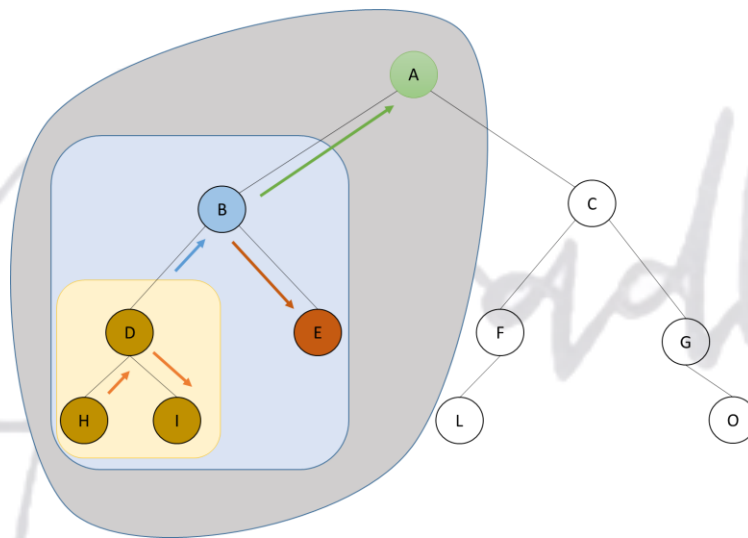
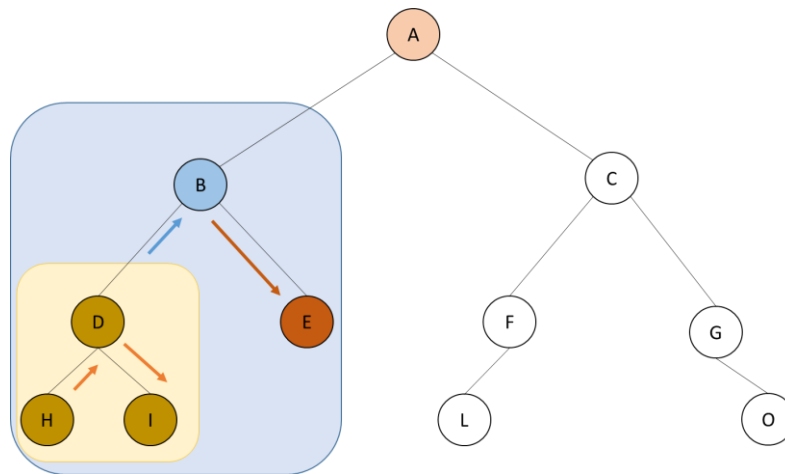


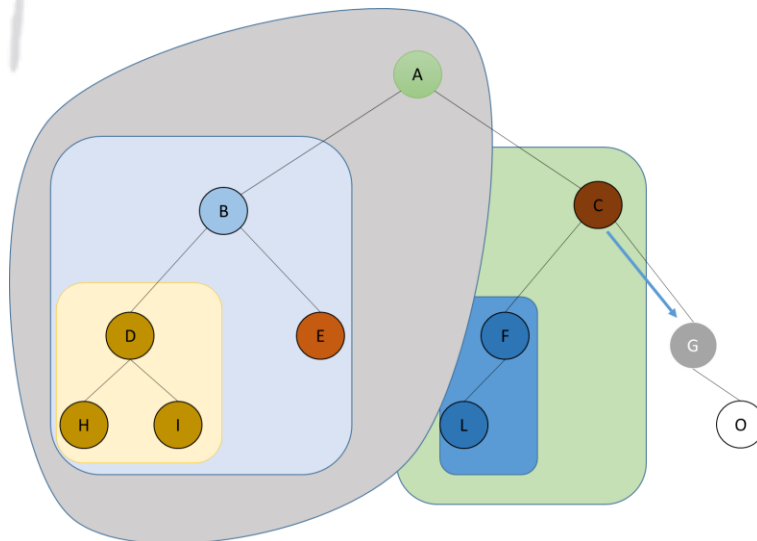
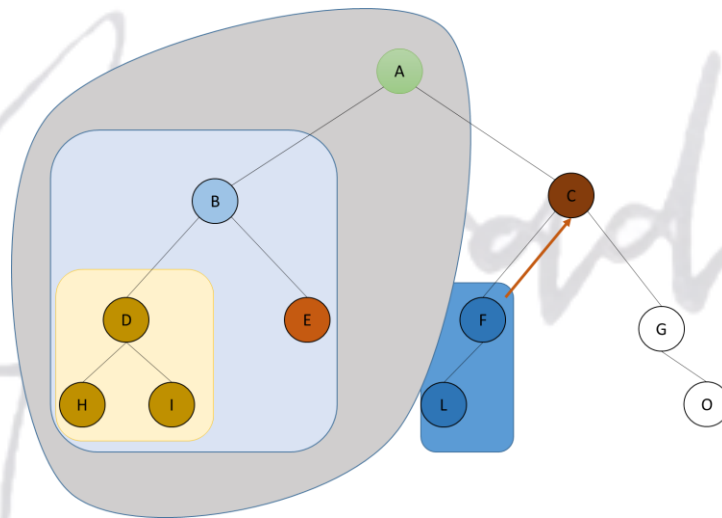
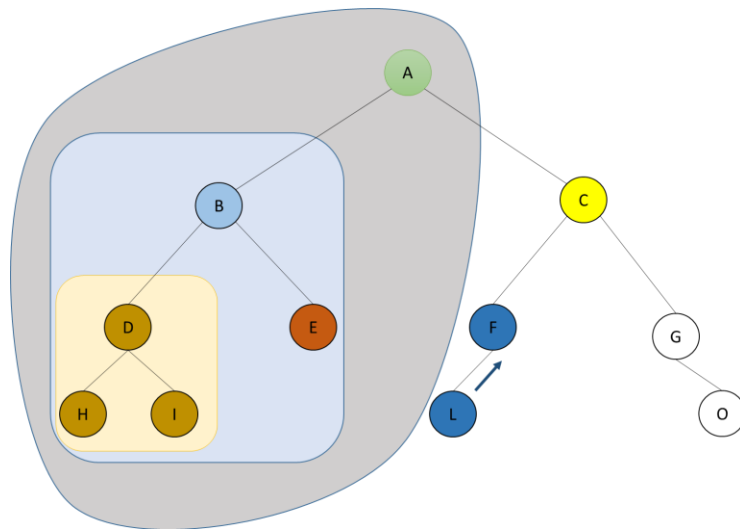
Step 4: The left subtree of the parent D is complete as shown below.

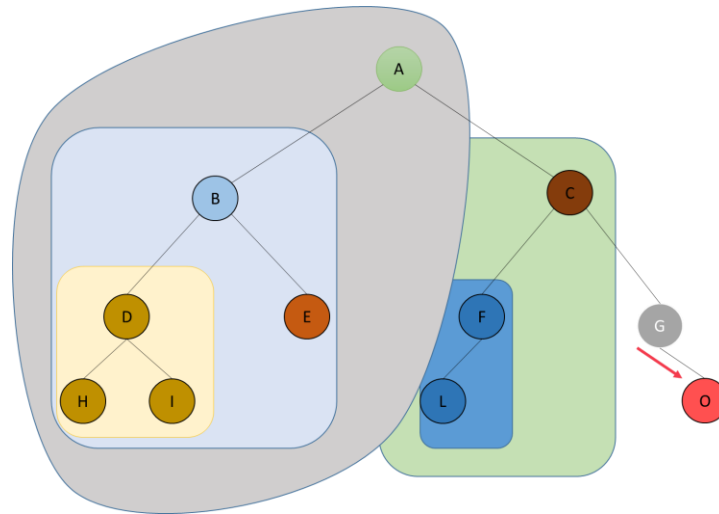


Step 5: Now repeat the steps for the Parent B.

Step 6: The traversal paths are shown below pictorially.



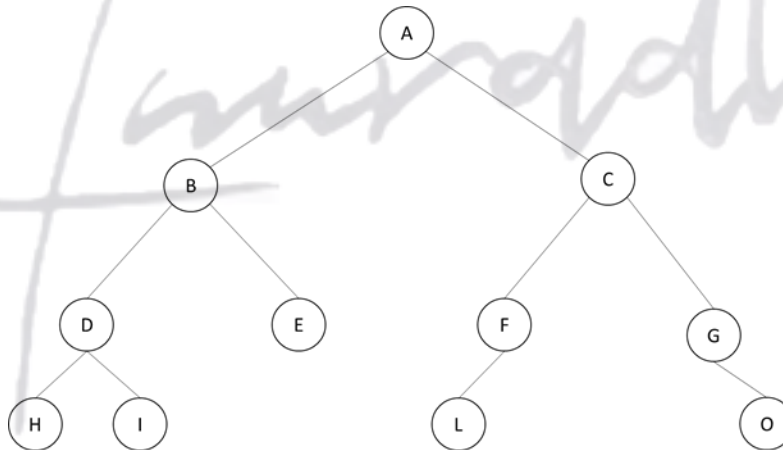




Step 7: The In Order traversal will be printed as

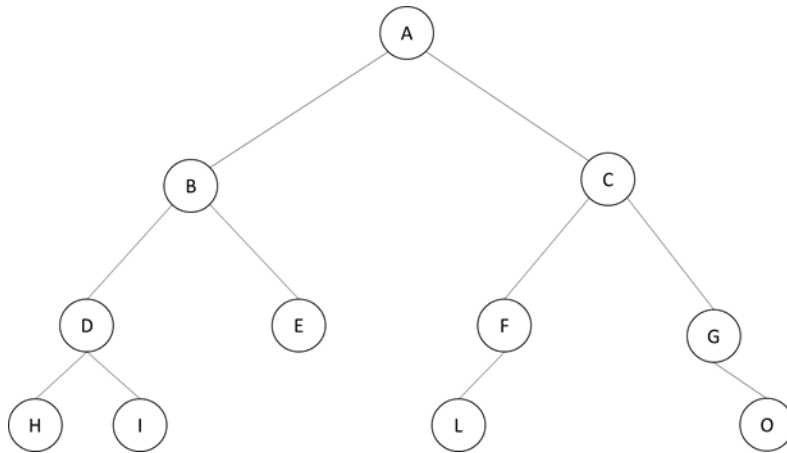
H, D, I, B, E, A, L, F, C, G, O.

ii. Pre - Order Traversal - Data \rightarrow LeftChild \rightarrow RightChild (DLR)



Similarly the preorder is performed as Inoreder. **A, B, D, H, I, E, C, F, L, G, O.**

iii. Post - Order Traversal - LeftChild \rightarrow RightChild \rightarrow Data (LRD)



The post order is : H, I, D, E, B, L, F, O, G, C, A.

Program to Create Binary Tree and display using In-Order Traversal.

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

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

```
struct Node{  
    int data;  
    struct Node *left;  
    struct Node *right;  
};
```

```
struct Node *root = NULL;
```

```
int count = 0;
```

```
struct Node* insert(struct Node*, int);
```

```
void display(struct Node*);
```

```
void main(){
```

```
    int choice, value;
```

```
    clrscr();
```

```
printf("\n----- Binary Tree ----- \n");
while(1){
    printf("\n***** MENU ***** \n");
    printf("1. Insert\n2. Display\n3. Exit");
    printf("\nEnter your choice: ");
    scanf("%d",&choice);
    switch(choice){
        case 1: printf("\nEnter the value to be insert: ");
                scanf("%d", &value);
                root = insert(root,value);
                break;
        case 2: display(root); break;
        case 3: exit(0);
        default: printf("\nPlease select correct operations!!! \n");
    }
}

struct Node* insert(struct Node *root,int value){
    struct Node *newNode;
    newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = value;
    if(root == NULL){
        newNode->left = newNode->right = NULL;
        root = newNode;
        count++;
    }
    else{
```

```
        if(count%2 != 0)
            root->left = insert(root->left,value);
        else
            root->right = insert(root->right,value);
    }
    return root;
}

// display is performed by using Inorder Traversal
void display(struct Node *root)
{
    if(root != NULL){
        display(root->left);
        printf("%d\t",root->data);
        display(root->right);
    }
}
```

6. Binary search tree

Binary Search Tree is a node-based binary tree data structure which has the following properties:

- i. The left subtree of a node contains only nodes with keys lesser than the node's key.
- ii. The right subtree of a node contains only nodes with keys greater than the node's key.
- iii. The left and right subtree each must also be a binary search tree.