

## Data Structures

### UNIT-I

#### Introduction to Data Structures *Data:*

*Data* is nothing but a piece of information. It can be a number, a string, or etc.

#### Types of Data:

**Atomic data** is the single entity, It cannot be sub divided. It is also known as scalar data For example, the integer 1234 (for example Roll Number) may be considered as a single integer value.

**Composite data.** Composite data can be sub divided into subfields that have meaning. It is also known as structured data. For example, a student's record consists of Roll\_Number, Name, Branch, Year, and so on.

#### Data Type

Data type is a term that specifies the type of data that a variable may store.

#### Built-in Data Types:

These are the fundamental data types. For example, int, float, and char are the built-in data types in C/C++ languages.

#### User-Designed data types:

The data types that are defined by the user are called *user-defined data types*. These types can be designed by using the built-in data types. Example: structures, unions, and classes.

#### Data Object

A *data object* is a set of elements (values). A data object has a data type to determine the type of values it may contain. It is a run-time instance of data structures. Data objects are of two types:

**Programmer defined data objects:** These are explicitly defined by the programmer at run-time Example: Variables, arrays, files, etc.

**System defined data objects:** These are automatically generated by the system when needed at run-time.

#### Abstract data type:

Data and the operations on the data are encapsulated and hidden from the user. An abstract data type is a data declaration packaged together with the operations that are meaningful for the data type. It includes the declaration of data, implementation of operations, and encapsulation of data and operations.

#### Data Structures

Data structures refer to data and representation of data objects within a program. A data structure is a collection of atomic and composite data types into a set with defined relationships.

#### Types of Data Structures:

The various types of data structures are as follows:

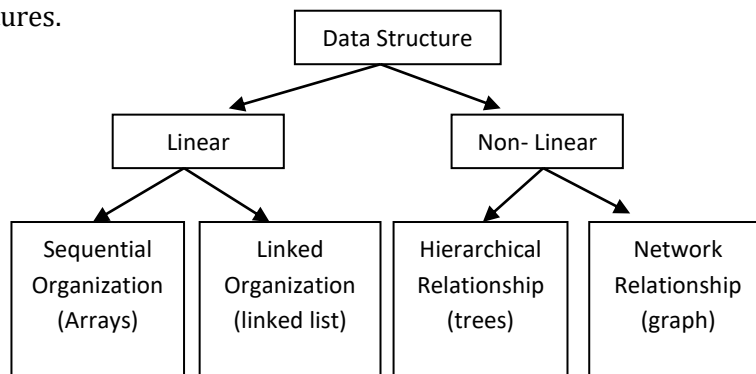
1. Primitive and non-primitive
2. Linear and non-linear
3. Static and dynamic
4. Persistent and ephemeral
5. Sequential and direct access

**Primitive data structures:** These are a set of primary or built-in data types. For example: The Data structures defined for integers and characters.

**Non-primitive data structures:** These are a set of derived elements. For example: Arrays, Class and structures.

#### Linear and Non-linear Data Structures:

A data structure that forms a sequence or a linear list is known as Linear Data Structures. In a linear data structure, every data element has a unique successor and predecessor These are represented in two ways either by using linked Lists or Arrays. A *data structure that does not form a sequence is known as Non-linear data structures*. These are used to represent the hierarchical or network relationships. In non-linear data structures, every data element may have more than one predecessor as well as successor. Trees and graphs are examples of non-linear data structures.



## Static and Dynamic Data Structures

A data structure that can be created during compilation time is known as Static data structure. An array (static array) is a static data structure. A data structure that is created at run-time is called *dynamic data structure*. A linked list is a dynamic data structure.

## Persistent and Ephemeral Data Structures

A data structure that supports operations on the most recent version as well as the previous version is known as a *persistent data structure*. A data structure that supports operations only on the most recent version is known as Ephemeral data structures.

## Sequential Access and Direct Access Data Structures

A data structure that follows a sequential order to access its elements is known as a Sequential access data structures. It means that to access the  $n$ th element, we must access the preceding ( $n - 1$ ) data elements. A linked list is a sequential access data structure. A data structure that allows to access any element directly is known as a direct access data structure. An array is an example of a direct access data structure.

## Algorithms

An algorithm is a step-by-step solution for any task. Each algorithm includes steps for

1. Input,
2. Processing, and
3. Output.

An algorithm is a *finite ordered* set of *unambiguous* and *effective* steps which, when followed, accomplish a particular task by accepting *zero or more input quantities* and generate *at least one output*. The following are the characteristics of algorithms:

- **Input** An algorithm is supplied with sufficient input.
- **Output** An algorithm must produce an output.
- **Unambiguous steps** Each step in the algorithm must be clear.
- **Finiteness** An algorithm must halt. It must have finite number of steps.
- **Effectiveness** Every instruction must be executed easily.

In brief, an algorithm is an ordered finite set of unambiguous and effective steps that produces a result and terminates.

## Algorithm Design Tools Pseudo code and Flowchart

Algorithms can be represented by using two popular tools:

1. Pseudo code
2. Flowchart

A pseudo code is an English-like presentation of the code required for an algorithm. It is partly English and partly computer language structure code. Pseudo code uses various notations:

- Algorithm header,
- Purpose,
- Conditions and Return statements,
- Variables,
- Statement numbers, and
- Sub algorithms.

## Algorithm header:

A *header* includes the name of the algorithm, the parameters, and the list of pre and post conditions. This information is important to know about the algorithm.

*Algorithm Add*

*Pre A and B to be added*

*Post Sum of A and B*

*Return None*

Step 1. START

Step 2. PRINT "ENTER TWO NUMBERS"

Step 3. INPUT A, B

Step 4.  $C \leftarrow A + B$

Step 5. PRINT C

Step 6. STOP

**Purpose:** The *purpose* is a brief description about what the algorithm does. It should be as brief as possible,

## Condition and Return Statements

**The pre condition** states the pre-requirements for the parameters.

**The post condition** identifies any action taken.

## Statement Numbers

The statements in an algorithm are numbered sequentially. Numbering helps identify the statements uniquely.

## Variables

Variables are needed in algorithms. We need not define every variable used in the algorithm. The use of meaningful variable names is appreciated

## Statement Constructs

There are three statement constructs used for developing an algorithm.

1. sequence
2. decision
3. repetition

### Sub-algorithms

Modular design of a problem breaks an algorithm into smaller units called *sub-algorithms*.

### Flowchart

- These are a graphical representation of the algorithm.
- They use symbols and language to represent sequence, decision, and repetition actions.

### Analysis of algorithms

There are several ways to organize data and write algorithms. We can analyse the algorithms to choose the best one. The Performance of an algorithm is measured by using the following parameters:

1. *Programmer's time complexity (It is rarely used)*
2. *Time complexity*
3. *Space complexity*

### Complexity of Algorithms:

Algorithms are measured in terms of time and space complexity.

- **Time complexity:** It is a measure of how much time is required to execute an algorithm for a given number of inputs.
- **Space complexity:** It is a measure of the amount of storage (memory) is required by the algorithm.

### Space Complexity:

Space complexity is the amount of computer memory required as a function of the input size. Space complexity measurement can be performed at two different times:

1. Compile time
2. Run time

### Compile Time Space Complexity

*Compile time space complexity* is defined as the storage requirement of a program at compile time. This amount of storage can be determined by summing up the storage size of each variable using declaration statements. For example, the space complexity of a non recursive factorial function of number  $n$  depends on the number  $n$  itself.

**Space complexity = Space needed at compile time**

### Run-time Space Complexity

The Run-time space complexity can be used to determine the space complexity of recursive functions and dynamic data structures.

**Space complexity = program space + Data space + Stack space**

- **Program space** This is the memory occupied by the program.
- **Data space** This is the memory occupied by constants and variables.
- **Stack space** This is the stack memory needed to save the function's run-time environment.

### Time Complexity

Time complexity  $T(P)$  is the time taken by a program  $P$ . It is the sum of its compile and execution times. This is system-dependent. *Best, Worst, and Average Cases*

- The minimum number of steps taken on any instance of size  $n$  is referred to as the **best case** complexity
- The maximum number of steps taken on any instance of size  $n$  is referred to as the **worst case** complexity.
- The average number of steps taken on any instance of size  $n$  is referred to as **average case** complexity.

### Computing Time Complexity of an Algorithm

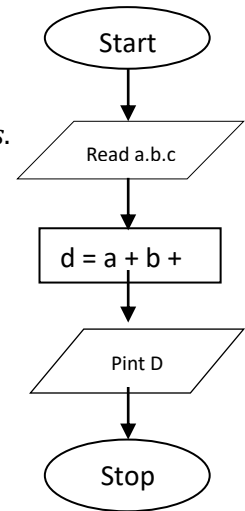
The total time taken by the algorithm is calculated using the sum of the time taken by each of its executable statements. The time required by each statement depends on the following:

1. The time required for executing it once
  2. The number of times the statement is executed
- The product of these two parameters gives the time required for that particular statement.
  - Compute the execution time of all executable statements.
  - The sum of all the execution times is the total time required for that algorithm or program.

### Big-O Notation

Big O notation is a symbolism used to describe the asymptotic behaviour of functions. Basically, it tells you how fast a function grows or declines. For example, if an algorithm is quadratic, we would say its efficiency is  $O(n^2)$  or on the order of  $n$  squared. The big-O notation can be derived from  $f(n)$  using the following steps:

1. In each term, set the coefficient of the term to 1.
  2. Keep the largest term in the function and discard the others.
- The terms are ranked from the lowest to the highest as follows:



$$\log_2 n \dots n \dots n \log_2 n \dots n^2 \dots n^3 \dots n^k \dots 2^n \dots n!$$

For example,

1. To calculate the big-O notation for

$$f(n) = n \times \frac{(n+1)}{2} = \frac{1}{2} n^2 + \frac{1}{2} n$$

we first remove all coefficients. This gives us  $n^2+n$  which, after removing the smaller factors, gives us  $n^2$  which, in big-O notation, is stated as  $O(f(n)) = O(n^2)$

2. To consider another example, let us look at the polynomial expression

$$f(n) = a_j n^k + a_{j-1} n^{k-1} + \dots + a_2 n^2 + a_1 n + a_0$$

We first eliminate all the coefficients as follows:

$$f(n) = n^k + n^{k-1} + \dots + n^2 + n + 1$$

The largest term in the expression is the first one, so we can say that the order of this polynomial expression is

$$O(f(n)) = O(n^k)$$

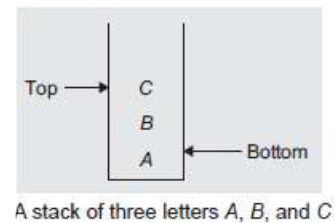
## LINEAR DATA STRUCTURE USING ARRAYS

Data can be organized in a linear or non-linear form.

In *linear* (or sequential) *organization*, all the elements of the data can be arranged in a particular sequence. Example: Arrays and linked lists. An *array* is a finite ordered collection of homogeneous data elements that provides direct access to any of its elements. Arrays are of different forms:

### STACKS

A stack is defined as a restricted list where all insertions and deletions are made only at one end, the top. There are two basic operations push and pop that can be performed on a stack; insertion of an element in the stack is called push and deletion of an element from the stack is called pop. The following figure shows a stack  $S = (A, B, C)$ , where A is the bottommost element and C is the topmost element:



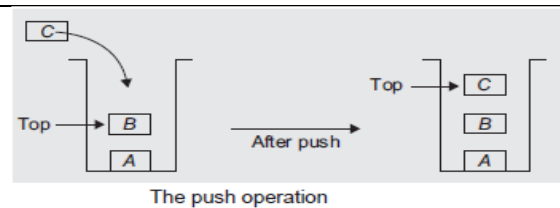
### Primitive Operations:

The following are the stack operations:

1. Push :- inserts an element on the top of the stack
2. Pop :- deletes an element from the top of the stack
3. GetTop :- reads (only reading, not deleting) an element from the top of the stack
4. Stack\_initialization—sets up the stack in an empty condition
5. Empty—checks whether the stack is empty
6. Full—checks whether the stack is full

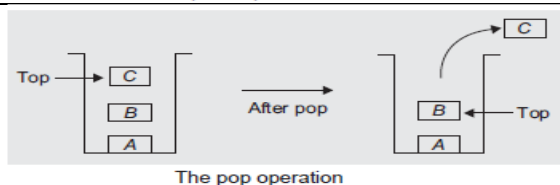
#### Push

The push operation inserts an element on the top of the stack. The recently added element is always at the top of the stack.



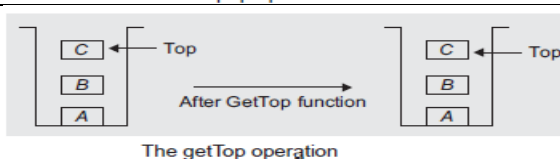
#### Pop

The pop operation deletes an element from the top of the stack. It modifies the stack so that the next element becomes the top element.



#### GetTop

The getTop operation gives information about the Top most elements and returns the element on the top of the stack.



### Applications of stack

The stack data structure is used in a wide range of applications. A few of them are the following:

1. Converting infix expression to postfix and prefix expressions
2. Evaluating the postfix expression
3. Checking well-formed parenthesis
4. Reversing a string
5. Processing function calls

6. Parsing of computer programs
7. Simulating recursion
8. In computations such as decimal to binary conversion
9. In backtracking algorithms.

### Stack ABSTRACT DATA TYPE

An **abstract data type**, or ADT, specifies a set of operations and the semantics of the operations but it does not specify the implementation of the operations.

A stack ADT is defined by the operations you can perform on it. Stacks can perform only the following operations:

1. **Create(S)**:- Creates an empty stack
2. **Push(i, S)** :- Inserts the element *i* on the stack *S* and returns the modified stack
3. **Pop(S)**:- Removes the topmost element from the stack *S* and returns the modified stack
4. **GetTop(S)**:- Returns the topmost element of stack *S*
5. **Is\_Empty(S)**:- Returns true if *S* is empty, otherwise returns false

To implement the ADT stack in C++, the operations are often implemented as functions to provide data abstraction. A program that uses stacks would access the stacks only through these functions and would not be concerned about the implementation. Example: Stack ADT specification

```
ADT Stack(element)
1. Declare Create() ↗stack
2. push(element, stack) ↗stack
3. pop(stack) ↗stack
4. getTop(stack) ↗element
5. Is_Empty(stack) ↗Boolean;
6. for all S stack, e ↗element, Let
7. Is_Empty(Create) = true
8. Is_Empty(push(e, S)) = false
9. pop(Create()) = error
10. pop(push(e, S)) = S
11. getTop(Create) = error
12. getTop(push(e, S)) = e
13. end
14. end stack
```

### REPRESENTATION STACKS USING ARRAYS

The simplest way to represent a stack is by using a one-dimensional array. It also makes the stack management simple. A stack implemented using an array is also called a *contiguous stack*. The only difficulty with an array is its static memory allocation. Once declared, the size cannot be modified during run-time. This leads to either poor utilization of the space.

<pre>class Stack { private: int Stack[50]; int MaxCapacity; int top; public: Stack() { MaxCapacity = 50; top = -1; } int getTop(); int pop(); void push(int Element); int Empty(); int CurrSize(); int IsFull(); }; int Stack :: getTop() { if(!Empty())</pre>	<pre>else return 0; } int Stack :: IsFull() { if(top == MaxCapacity - 1) return 1; else return 0; } int Stack :: CurrSize() { return(top + 1); } void Stack :: push(int Element) { if(!IsFull()) Stack[++top] = Element; } void main() { Stack S;</pre>
--	---

```

return(Stack[top]);
}
int Stack::pop()
{
if(!Empty())
return(Stack[top--]);
}
int Stack::Empty()
{
if(top == -1)
return 1;

```

```

S.pop();
S.push(1);
S.push(2);
cout << S.getTop() << endl;
cout << S.pop() << endl;
cout << S.pop() << endl;
}

```

### Prefix, Infix, Postfix Notations for Arithmetic Expression

The Polish Mathematician Han Lukasiewicz suggested a notation called Polish notation. The Polish notation gives two alternatives to represent an arithmetic expression:

1. The postfix notation
2. The prefix notation.

It indicates that the order in which the operations are to be performed is determined by the positions of the operators and operands in the expression. Parentheses is not required while writing expressions in Polish notation. The conventional way of writing the expression is called Infix. because the binary operators occur between the operands, and unary operators precede their operand. For example, the expression  $((A + B) \times C)/D$  is an infix expression.

- In postfix notation, the operator is written after its operands.
- In prefix notation, the operator precedes its operands.

The following figure shows one sample expression in all three notations.

INFIX	PREFIX	POSTFIX
(operand)(operator)(operand) (A+B)XC	(operator)(operand)(operand) X+ABC	(operand)(operand)(oprator) AB+C X

### Converting Infix Expression to Postfix Expression, Infix to Postfix Conversion

The following algorithm illustrates the infix to postfix conversion.

1. Scan expression E from left to right, character by character, till character is '#' ch = get\_next\_token(E)
2. while(ch != '#')
  - if(ch = ')') then ch = pop()
  - while(ch != '(')
  - Display ch
  - ch = pop()
  - end while
  - if(ch = operand) display the same
  - if(ch = operator) then
  - if(ICP > ISP) then push(ch)
  - else
  - while(ICP <= ISP)
  - pop the operator and display it
  - end while
  - ch = get\_next\_token(E)
  - end while
3. if(ch = '#') then while(!emptystack()) pop and display
4. stop

Example: Convert the following infix expression to its postfix form:

$$A \wedge B \times C - C + D / A / (E + F)$$

**Solution** Conversion of infix to postfix form can be illustrated as in the following Table:

Character scanned	Stack contents	Postfix expression
A	EMPTY	A
^	^	A
B	^	AB
X	X	AB^
C	X	AB^C
-	-	AB^CX
C	-	AB^CXC
+	+	AB^CXC-
D	+	AB^CXC-D
/	+ /	AB^CXC-D
A	+ /	AB^CXC-DA
/	+ /	AB^CXC-DA/
(	+ / (	AB^CXC-DA/
E	+ / (	AB^CXC-DA/E
+	+ / (+	AB^CXC-DA/E
F	+ / (+	AB^CXC-DA/EF
)	+ /	AB^CXC-DA/EF+
	EMPTY	AB^CXC-DA/EF+/+

### Evaluating the Postfix Expression

Postfix Expression Evaluation The postfix expression may be evaluated by making:

1. A left-to-right scan,
2. Stacking operands, and

### 3. Evaluating operators

#### 4. Placing the result onto the stack.

This process continues till the stack is not empty or on occurrence of the character #, which denotes the end of the expression. The following algorithm lists the steps involved in the evaluation of the postfix expression:

1. Let E denote the postfix expression

2. Let Stack denote the stack data structure to be used & let Top = -1

3. while(1) do

begin

X = get\_next\_token(E) // Token is an operator, operand, or delimiter

if(X = #) {end of expression} then return

if(X is an operand) then push(X) onto Stack

else {X is operator}

begin

OP1 = pop() from Stack

OP2 = pop() from Stack

Tmp = evaluate(OP1, X, OP2)

push(Tmp) on Stack

end

{

If X is operator then pop the correct number of operands from stack for operator X. Perform the operation and push the result, if any, onto the stack

}

end

4. stop

### Checking Well-formed (Nested) Parenthesis:

#### CHECKING CORRECTNESS OF WELLFORMED PARENTHESES

To ensure that the parentheses in a mathematical expression are nested correctly, we need to check that

1. There are equal numbers of right and left parentheses

2. Every right parenthesis is preceded by a matching left parenthesis A well-formed expression must satisfy .

The following two conditions:

1. The parenthesis count at each point in the expression is non-negative.

2. The parenthesis count at the end of the expression is 0.

We can use a stack to maintain the count of parenthesis in an expression. The following steps should be followed to determine the correctness of an expression:

- Whenever a left parenthesis is encountered, it is pushed onto the stack.
- Whenever a right parenthesis is encountered, the stack is examined. If the stack is empty, then the string is declared to be invalid.
- When the end of the string is reached, the stack must be empty. Otherwise, the string is declared to be invalid.

### Processing of Function Calls

Micro processors use a stack to process the function calls during the execution of a Computer Program. It is useful to remember the place where the call was made so that it can return there after executing the function. Suppose we have three functions, say, A, B, and C, and one main program. Let the main invoke A, A invoke B, and B in turn invoke C. This sequence follows the LIFO property, as shown below:

The output of the above procedure is shown in the following figure:

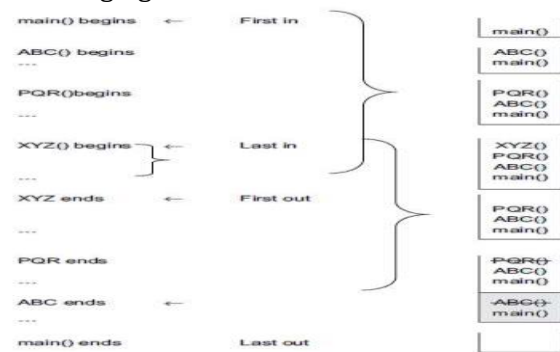


Figure: Use of stack for processing of function calls

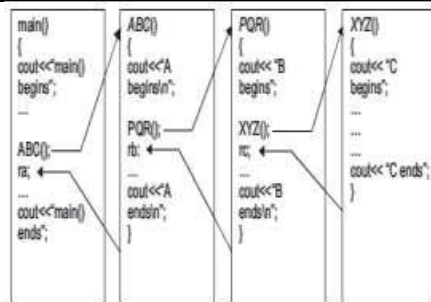


Figure: Processing of Function calls

### Reversing a String. (REVERSING A STRING WITH A STACK)

A stack can be very much useful to reverse a string in simple manner. Stack's LIFO property best suits to reverse a string. Let us consider a string ABCDEF is to be reversed. With a stack:

- We can simply scan the sequence
- Pushing each element onto the stack as it is encountered, until the end of the sequence.
- Pop the stack repeatedly and
- Send each popped element to the output, until the stack is empty.

The following table illustrates this algorithm:

Table : Reversal of a string using a stack

Input	Action	Stack	Display
ABCDEF	Push A	A ← top of stack	-
BCDEF	Push B	AB ← top of stack	-
CDEF	Push C	ABC ← top of stack	-
DEF	Push D	ABCD ← top of stack	-
EF	Push E	ABCDE ← top of stack	-
F	Push F	ABCDEF ← top of stack	-
End	Pop and display	ABCDE ← top of stack	F
	Pop and display	ABCD ← top of stack	FE
	Pop and display	ABC ← top of stack	FED
	Pop and display	AB ← top of stack	FEDC
	Pop and display	A ← top of stack	FEDCB
	Pop and display	Stack empty	FEDCBA
	Stop		

The following code reverses a string by using a stack:

```
main()
{
    Stack S; // here Stack is the character stack
    char str[]=" My String", ch;
    int i;
    ch = str[0];
    i = 1;
    while(ch != '\0')
    {
        S.push(ch);
        ch = str[i++];
    }
    while(!S.isEmpty())
    {
        cout << S.pop();
    }
}
```

## UNIT-II

### Recursion

Calling a function by itself is known as **recursion**. A function which implements recursion is called a **recursive function**. Recursion is extremely powerful:

- It is useful to express complex processes easily.
- It is widely used in calculating the factorial of a number.
- It is also used in playing complex games.

### Finding the Factorial of a Number using Recursion

A recursive function for factorial of  $n$  can be defined as

$$n! = n \times (n-1)!, \text{ Where } 1! = 1$$

The above recursive definition of factorial has the following two steps:

1. If  $n=1$ , then factorial of  $n=1$
2. Otherwise, factorial of  $n = n \times \text{factorial of } (n-1)$ .

The following Program Code implements recursion to find the Factorial of a Number:

```
int Factorial(int n)
{
    if(n == 1) // end condition
        return 1;
```



```

else
return Factorial(n - 1) * n;
}

```

## Recurrence

A *recurrence* is a well-defined mathematical function. A recurrence function can be applied within its own definition. For example, let us consider the Fibonacci sequence. The Fibonacci sequence is the sequence of numbers 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... The following function defines  $F(n)$ th value in the sequence:

$$F(n) = F(n-1) + F(n-2)$$

This function  $F$  is called *recurrence* since it computes the  $n$ th value in terms of  $(n-1)$ th and  $(n-2)$ th Fibonacci values. Such recurrence oriented problems can be easily expressed as recursive functions in programming. Let us consider how to compute the value of three to the fourth power:

$$3^4 = 3 \times 3^3$$

Three cubed can be defined as

$$3^3 = 3 \times 3^2$$

Three squared is

$$3^2 = 3 \times 3$$

finally,

$$3 = 3 \times 3^0 = 3 \times 1$$

The recurrence for this computation is

$$X^Y = X * X^{Y-1}$$

The following is a recursive code for computation of the above equation,  $XY = X * XY-1$

```

long int Power(int x, int y)
{
if(y == 0) // end condition
return(1);
else
return( x * Power(x, y - 1); // This is the "recursive call"
}

```

## Use of stack in recursion

The following code is useful to understand how recursive functions use the stack:

```

if(n <= 1)
return 1;
else
return n * Factorial(n - 1);

```

The following steps shows how a stack can be used when computing the factorial of 3 ( $3 \times 2 \times 1 = 6$ )

- When the function is called for the first time, the else statement is executed and pushes  $n$  (value =3) onto the stack
- When it is called second time with the value 2, the else statement is again executed and  $n$  (value =2) is pushed onto the stack.
- When the function calls itself for the third time with the value 1, the if statement is executed and returns 1.
- Then it pops the last value 2 from the stack and multiplies it by 1.
- Then it pops the next value 3 from the stack and multiplies it with the above factorial. It gives the value 6.

The recursive functions are categorized as:

1. Direct recursion
4. Tree recursion

2. Indirect recursion
5. Tail recursion

3. Linear recursion

### Direct recursion

When a function calls itself directly then it is known as Direct recursion. The following function is an example of direct recursion: Example:

```

int Power(int x, int y)
{
if(y == 1)
return x;
else
return (x * Power(x, y - 1));
}

```

### Indirect recursion

A function that calls another function, which in turn calls it is known as Indirect recursion. The following is an example of an indirect recursion:

```

int Fact(int n)
{
if(n <= 1)
return 1;
else
return (n * Dummy(n - 1));
}

```

### Linear Recursion

A recursive function is said to be *linearly recursive* when no pending operation involves another recursive call, for example, the Fact() function:

```

int Factorial(int n)
{
if(n == 1) // end condition
return 1;
else
return Factorial(n - 1) * n;
}

```

<pre>}</pre>	<pre>} void Dummy(int n) { Fact(n); }</pre>	
<p><b>Tree Recursion</b></p> <p>In a recursive function, if there is another recursive call in the set of operations to be completed after the recursion is over, this is called a <i>tree recursion</i>. Examples of tree recursive functions are the quick sort and merge sort algorithms.</p>	<p><b>Tail recursion</b></p> <p>When a function has no pending operations to be performed on return from a recursive call, then it is known as Tail recursion. The following Binary_Search() function is an example of a tail recursive function:</p> <pre>int Binary_Search(int A[], int low, int high, int key) { int mid; if(low &lt;= high) { mid = (low + high)/2; if(A[mid] == key) return mid; else if(key &lt; A[mid]) return Binary_Search(A, low, mid - 1, key); else return Binary_Search(A, mid + 1, high, key); } return -1; }</pre>	

### Execution of Recursive Calls

Recursive calls are executed in the following manner:

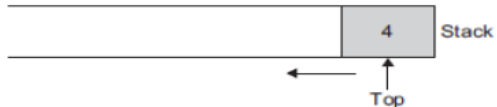
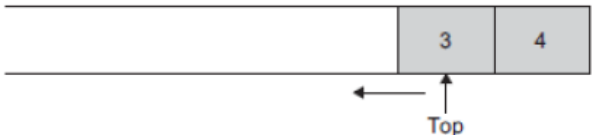
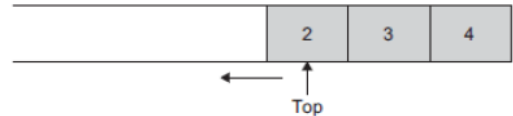
- On each recursive call, All reference parameters and local variables, Function values and return addresses are PUSHed onto the stack.
- The data is placed in a *stack frame*. A stack frame contains four different elements:
  - The reference parameters
  - Local variables
  - The return address
  - The expression to receive the return value,

if any Consider the following two lines from the Factorial() function:

**if(n <= 1) return 1;**

**else return n \* Factorial(n - 1);**

Consider the first call as Factorial(4). Now,

1. $n = 4$ Hence, statement 2, which is a recursive call, is executed.	 Push 4 onto the stack and call Factorial(4 - 1).
Now, 2. $n = 3$ Hence, push 3 onto the stack and call Factorial(2).	
Now, 3. $n = 2$ Hence, push 2 onto the stack and call Factorial(1).	

4.  $n = 1$

Now it executes statement 1 and returns the value 1.

5. Pop the contents and  $n = 2$ , so now the expression becomes  $2 \times 1$ .

6. Now,  $n = 3$  after popping the top of the stack contents. Therefore, the expression is  $3 \times 2 \times 1$ .

7. After popping the top of the stack contents applying  $n = 4$ , the expression is  $4 \times 3 \times 2 \times 1 = 24$ .

8. Now the stack is empty, and the answer is  $4! = 24$ .

### Writing Recursive Code

A recursive function can be written by using the following steps:

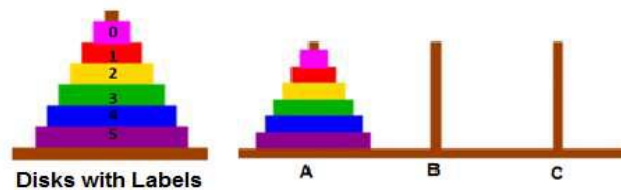
1. Write the function header. Identify some unit of measure.
2. Decompose the problem into sub problems. Identify clearly the non recursive case( Base Case or End Condition) of the problem.
3. Write recursive calls to solve those subproblems.
4. Write the code to combine, enhance, or modify the results of the recursive call(s).
5. Write the end condition(s) to handle any situations.

### Tower of Hanoi: An Example of Recursion

The recursive function for the Towers of Hanoi solution takes the largest disk as a parameter to be moved. It takes three parameters to indicate three pegs:

- **Source Peg** → To indicates from which peg the tower should be moved
- **Destination Peg** → To indicate which peg it should go
- **Spare Peg** → To use it temporarily.

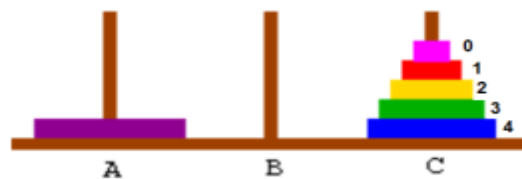
The following figure shows the initial position of the problem:



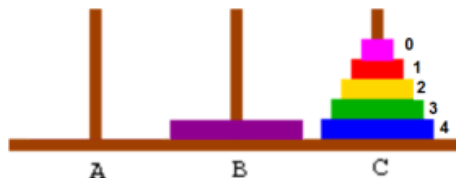
### Hanoi—initial position

We can break this into three basic steps.

1. Move the disk 4 and the ones smaller than that from the peg A (source) to peg C (spare), using peg B (dest) as a spare. We achieve it by recursively using the same function. After finishing this, we will have all the disks smaller than disk 4 on peg C, as shown below:



2. Now, with all the smaller disks on the spare peg C, we can move disk 5 from peg A to peg B, as shown below:



3. Finally, we want disk 4 and the smaller disks to be moved from peg C to peg B. We do this recursively using the same function again. At the end, we have disk 5 and the smaller ones on peg B as shown below:



Figure: Tower of Hanoi – Final Step

The following algorithm illustrates Tower of Hanoi recursive function: algorithm

```
HTower(disk, source, dest, spare)
if disk == 0, then
    move disk from source to dest
else
    HTower(disk - 1, source, spare, dest) // Step 1
    move disk from source to dest // Step 2
    HTower(disk - 1, spare, dest, source) // Step 3
end if
```

**Iteration versus Recursion** *Recursion is a top-down approach* of problem solving. It divides the problem into pieces, postponing the rest.

**Iteration is more of a bottom-up approach.** It begins with what is known and from this constructs the solution step by step. The non-recursive version is more efficient as the overhead of parameter passing in most compilers is heavy.

#### **Demerits of Recursive Algorithms**

1. Many programming languages do not support recursion.
2. Mathematical functions using recursion requires additional execution time and memory space
3. A recursive function can be called from within or outside itself. For this it must save the return addresses in some order.

#### **Demerits of Iterative Methods**

1. Iterative code is not readable. It is not easy to understand.
2. It requires looping statements and needs a complex logic.
3. It may result in a lengthy code.

### **QUEUES**

*Queue* is a special type of data structure that performs insertions at one end called the *rear* and deletions at the other end called the *front*. A queue is a *first in first out* (FIFO) structure.

- Queues are frequently used in operating systems, network and database implementations, and other areas.
- Queues are very useful in time-sharing and distributed computer systems.
- Whenever a user places a request, the operating system adds the request at the end of the queue. The CPU executes the job at the front of the queue.

#### **The basic/Primitive operations on Queues:**

A minimal set of operations on a queue is as follows:

1. `create()`—creates an empty queue, *Q*
2. `add(i,Q)`—adds the element *i* to the rear end of the queue, *Q* and returns the new queue
3. `delete(Q)`—takes out an element from the front end of the queue and returns the resulting queue
4. `getFront(Q)`—returns the element that is at the front position of the queue
5. `Is_Empty(Q)`—returns true if the queue is empty; otherwise returns false

#### **Queue - Abstract Data Type**

A queue abstract data type (ADT) needs a suitable data structure for storing the elements in the queue and the functions operating on it. The following algorithm gives a complete specification for the queue ADT:

1. `class queue(element)`
2. `declare create() A queue`
3. `add(element, queue) A queue`
4. `delete(queue) A queue`
5. `getFront(queue) A queue`
6. `Is_Empty(queue) A Boolean;`
7. `For all Q OE queue, i OE element let`
8. `Is_Empty(create()) = true`
9. `Is_Empty(add(i,Q)) = false`
10. `delete(create()) = error`
11. `delete(add(i,Q)) =`
12. `if Is_Empty(Q) then create`
13. `else add(i, delete(Q))`
14. `getFront(create) = error`
15. `getFront(add(i, Q)) =`
16. `if Is_Empty(Q) then i`
17. `else getFront(Q)`
18. `end`
19. `end queue`

**Representation Queues Using Arrays** The following are various operations on the queue using arrays:

**Create:** This operation should create an empty queue. Here max is the maximum initial size that is defined.

```
#define max 50
int Queue[max];
int Front = Rear = -1;
```

This declaration creates an empty queue of size max. The two variables Front and Rear are initialized to represent an empty queue.

**Is\_Empty:** This operation checks whether the queue is empty or not. If Front = Rear, then Is\_Empty returns true, else returns false.

```
bool Is_Empty()
```

```

{
    if(Front == Rear)
        return 1;
    else
        return 0;
}

```

**Is\_Full:** A queue must be checked before inserting any element. When Rear points to the last location of the array, it indicates that the queue is full, that is, there is no space to accommodate any more elements.

```

bool Is_Full()
{
    if(Rear == max - 1)
        return 1;
    else
        return 0;
}

```

**Add** This operation adds an element in the queue if it is not full. The new element is added at the (rear =1)th location.

```

void Add(int Element)
{
    if(Is_Full())
        cout << "Error, Queue is full";
    else
        Queue[++Rear] = Element;
}

```

**Delete** This operation deletes an element from the front of the queue and sets Front to point to the next element

```

int Delete()
{
    if(Is_Empty())
        cout << "Sorry, queue is Empty";
    else
        return(Queue[++Front]);
}

```

**getFront** The operation getFront returns the element at the front.

```

int getFront()
{
    if(Is_Empty())
        cout << "Sorry, queue is Empty";
    else
        return(Queue[Front + 1]);
}

```

**The following program implements queues using array representation**

<pre> //Queue ADT class queue { private: int Rear, Front; int Queue[50]; int max; int Size; public: queue() {     Size = 0; max = 50;     Rear = Front = -1 ; } int Is_Empty(); int Is_Full(); void Add(int Element); </pre>	<pre> void queue :: Add(int Element) {     if(!Is_Full())         Queue[++Rear] = Element;         Size++; }  int queue :: Delete() {     if(!Is_Empty())     {         Size--;         return(Queue[++Front]);     } }  int queue :: getFront() { </pre>
--	---

```

int Delete();
int getFront();
};
int queue :: Is_Empty()
{
if(Front == Rear)
return 1;
else
return 0;
}
int queue :: Is_Full()
{
if(Rear == max - 1)
return 1;
else
return 0;
}

```

```

if(!Is_Empty())
return(Queue[Front + 1]);
}
void main(void)
{
queue Q;
Q.Add(11);
Q.Add(12);
Q.Add(13);
cout << Q.Delete() << endl;
Q.Add(14);
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
cout << Q.Delete() << endl;
Q.Add(15);
Q.Add(16);
cout << Q.Delete() << endl;
}

```

### Circular Queues

Array implementation of queues leads to the Queue\_Full state even though the queue is not actually full. The technique of *wraround* upon reaching the end of the array eliminates this problem. A technique which allows the queues to wraparound from end to start is called a *circular queue*.

A circular queue allows us to add elements in the empty cells even though we have reached the end of the array. The queue is said to be full only when there are  $n$  elements in the queue.

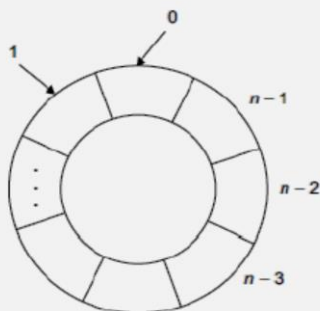


Figure: A Circular Queue A conceptual view

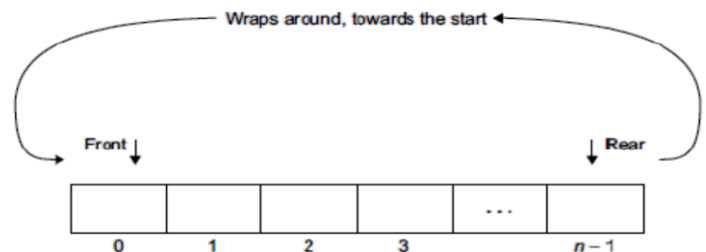


Figure: Circular queue - Physical view

In a circular queue, when the *rear* is  $n-1$  and a new element is to be added, the *rear* should be set to 0. Initially, both the *front* and the *rear* are set to 1.

**The following program illustrates a circular queue:**

```

#include<iostream.h>
class Cqueue
{
private:
int Rear, Front;
int Queue[50];
int Max;
int Size;
public:
Cqueue() {Size = 0; Max = 50; Rear = Front = -1;}
int Empty();
int Full();
void Add(int Element);
int Delete();
int getFront();
};
int Cqueue :: Empty()
{

```

```

int Cqueue :: Delete()
{
if(!Empty())
Front = (Front + 1) % Max;
Size--;
return(Queue[Front]);
}
int Cqueue :: getFront()
{
int Temp;
if(!Empty())
Temp = (Front + 1) % Max;
return(Queue[Temp]);
}
void main(void)
{
Cqueue Q;
Q.Add(11);

```

<pre> if(Front == Rear) return 1; else return 0; } int Cqueue :: Full() { if(Rear == Front) return 1; else return 0; } void Cqueue :: Add(int Element) { if(!Full()) Rear = (Rear + 1) % Max; Queue[Rear] = Element; Size++; } </pre>	<pre> Q.Add(12); Q.Add(13); cout &lt;&lt; Q.Delete() &lt;&lt; endl; Q.Add(14); cout &lt;&lt; Q.Delete() &lt;&lt; endl; cout &lt;&lt; Q.Delete() &lt;&lt; endl; cout &lt;&lt; Q.Delete() &lt;&lt; endl; Q.Add(15); Q.Add(16); cout &lt;&lt; Q.Delete() &lt;&lt; endl; } </pre>
---	---

### Advantages of Using Circular Queues:

1. By using circular queues, data shifting is avoided. The mod() operation wraps the queue back to its beginning.
2. Circular queue is advantageous for fixed size elements.
3. Many practical applications such as printer queue, priority queue, and simulations use the circular queue.

### Double Ended Queue (Deque)

- The word *deque* is a short form of double ended queue. It is pronounced as 'deck'. *Deque* defines a data structure where Insertion and Deletion occur at both the ends i.e. front and rear of the queue.
- A *deque* is a generalization of both a stack and a queue. It supports both stack-like and queue-like capabilities.
- A deque is optimized for fast index-based access and efficient insertion at either of its ends.

The following figure shows the representation of a deque:

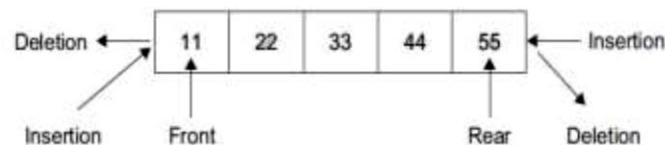


Fig. Representation of a deque

The following are the four operations associated with deque:

1. EnqueueFront()—adds elements at the front end of the queue
2. EnqueueRear()—adds elements at the rear end of the queue
3. DequeueFront()—deletes elements from the front end of the queue
4. DequeueRear()—deletes elements from the rear end of the queue

### Applications of deque

Deque is useful:

- Where the data to be stored has to be ordered
- Compact storage is needed
- Retrieval of data elements has to be faster.

### Variations of deque

There are two variations of a deque:

- The *input-restricted deque*: the inputrestricted deque allows insertions only at one end. It has the functions DequeueFront(), DequeueRight(), and EnqueueFront()**or** EnqueueRear()
- The *output-restricted deque*: The outputrestricted deque allows deletions from only one end. It has the functions: DequeueFront()**or** DequeueRight(), EnqueueFront() and EnqueueRear()

### Applications of Queues:

- The most useful application of queues is the simulation of a real world situation.
- Queues are also very useful in a timesharing computer systems.
- Operating system uses the queues to execute the user jobs
- Shared I/O devices maintains its own queue of requests.
- A queue is used for finding a path using the *breadth-first search* of graphs.

## Linked Lists

A *linked list* is an ordered collection of data. Each element (node) in a linked list contains a minimum of two values, *data* and *link(s)* to its successor (and/or predecessor). A link is made from each item to the next item in the list as shown in Figure:

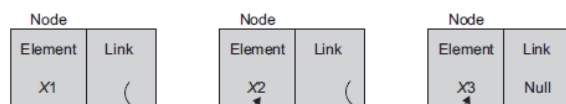


Fig. Linked list

- A linked list can be implemented using arrays, dynamic memory management, and pointers.
- The dynamic memory management can allocate memory at run-time.
- Each linked list has a head pointer that refers to the first node of the list and the data nodes storing data member(s).
- The linked list may have a *header node*, *tail pointer*, and so on.

### Linked List Terminology

The following terms are commonly used in linked lists:

**Header node** A header node is a special node that is attached at the beginning of the linked list. This header node may contain special information (metadata) about the linked list. The following figure shows a header node:

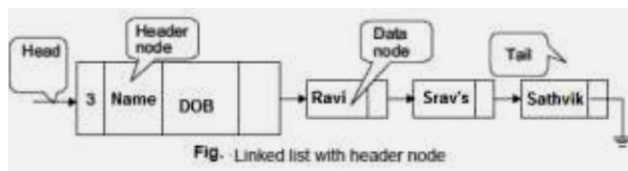


Fig. Linked list with header node

**Data node** The list contains data nodes that store the data members and link(s) to its predecessor (and/or successor).

**Head pointer:** A linked list must always have at least one pointer pointing to the first node (head) of the list. This pointer is often called *head pointer*, **Tail pointer** A pointer pointing to the last node of a linked list called the *tail pointer*.

### Primitive Operations

The following are the basic operations to be performed on linked lists:

1. Creating an empty list
2. Inserting a node
3. Deleting a node
4. Traversing the list

Some more operations, which are based on the basic operations, are as follows:

5. Searching a node
6. Updating a node
7. Printing the node or list
8. Counting the length of the list
9. Reversing the list
10. Sorting the list using pointer manipulation
11. Concatenating two lists
12. Merging two sorted lists into a third sorted list

### Linked List Abstract Data Type

The most flexible implementation of a linked list is by using pointers. In C++, we can view the entire linked list as an object of the class LList. The following figure shows an abstract representation of a linked list.



Fig. Abstract representation of linked list

### TYPES OF LINKED LIST

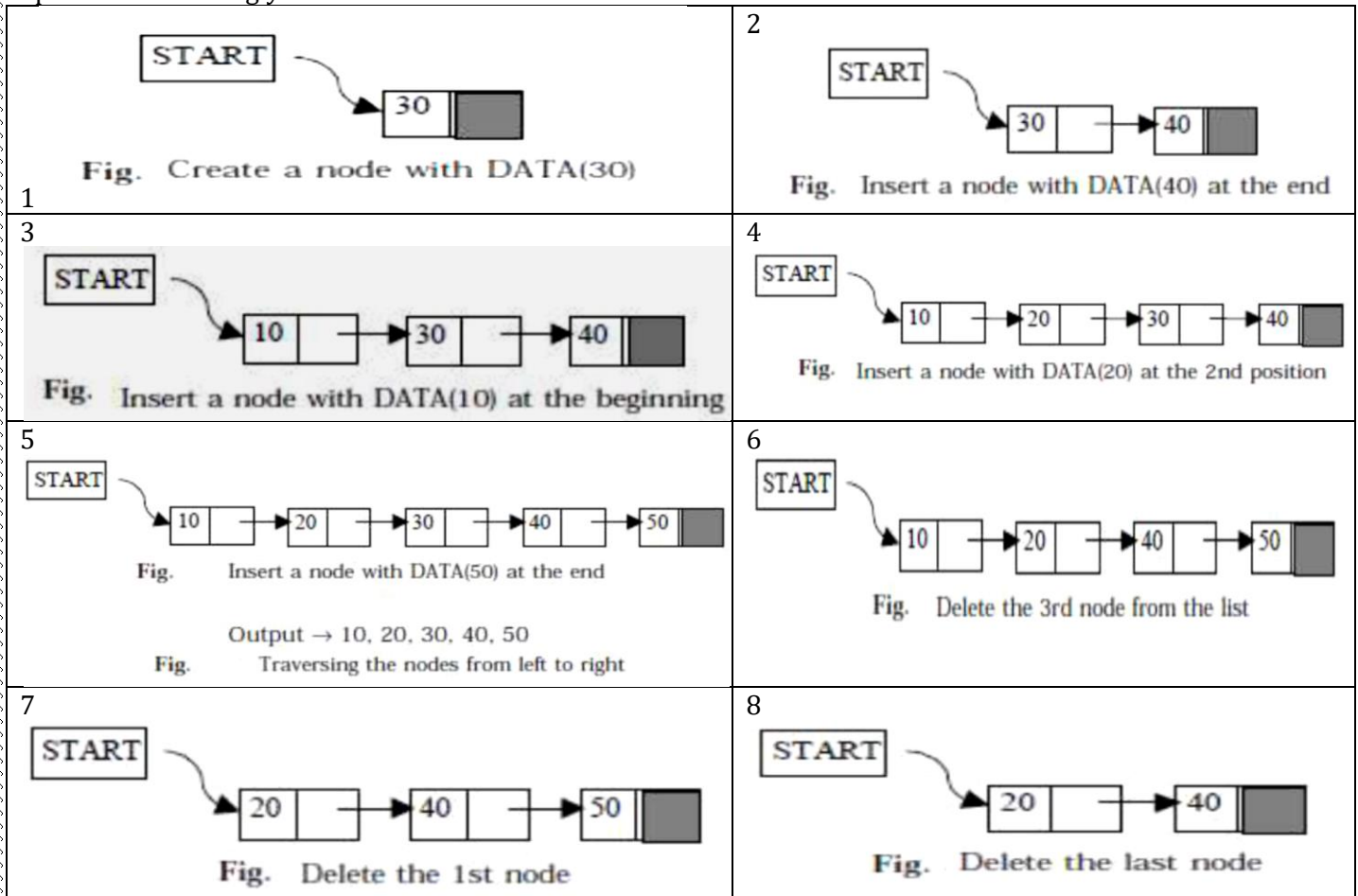
Basically we can divide the linked list into the following three types:

1. Singly linked list
2. Doubly linked list
3. Circular linked list

### SINGLY LINKED LIST



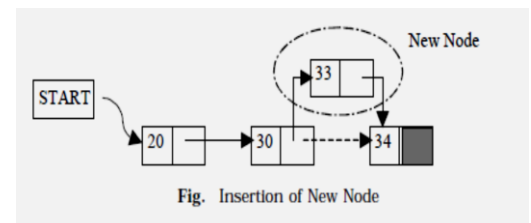
All the nodes in a singly linked list are arranged sequentially by linking with a pointer. A singly linked list can grow or shrink, because it is a dynamic data structure. Following figure explains the different operations on a singly linked list.



### ALGORITHM FOR INSERTING A NODE

Suppose,

- **START** is the first position in linked list.
- Let **DATA** be the element to be inserted in the new node.
- **POS** is the position where the new node is to be inserted.
- **TEMP** is a temporary pointer to hold the node address.



#### Insert a Node at the beginning

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. If (START equal to NULL)
  - (a) NewNode → Link = NULL
5. Else
  - (a) NewNode → Link = START
6. START = NewNode
7. Exit

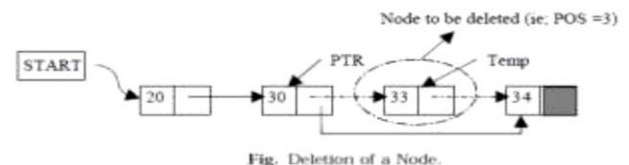
#### Insert a Node at the end

1. Input DATA to be inserted
2. Create a NewNode
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
8. If (START equal to NULL)
  - (a) START = NewNode
9. Else
  - (a) TEMP = START
  - (b) While (TEMP → Next not

### ALGORITHM FOR DELETING A NODE

Suppose,

- **START** is the first position in linked list.
  - Let **DATA** be the element to be deleted.
  - **TEMP**, **HOLD** is a temporary pointer to hold the node address.
1. Input the DATA to be deleted
  2. if ((START → DATA) is equal to DATA)
    - (a) TEMP = START
    - (b) START = START → Next
    - (c) Set free the node TEMP, which is deleted
    - (d) Exit
  3. HOLD = START
  4. while ((HOLD → Next → Next) not equal to NULL)
    - (a) if ((HOLD → NEXT → DATA) equal to



equal to NULL) (i) TEMP = TEMP → Next 10. TEMP → Next = NewNode 11. Exit <b>Insert a Node at any specified position</b> 1. Input DATA and POS to be inserted 2. initialise TEMP = START; and j = 0 3. Repeat the step 3 while( k is less than POS) (a) TEMP = TEMP → Next (b) If (TEMP is equal to NULL) (i) Display "Node in the list less than the position" (ii) Exit (c) k = k + 1 4. Create a New Node 5. NewNode → DATA = DATA 6. NewNode → Next = TEMP → Next 7. TEMP → Next = NewNode 8. Exit	DATA) (i) TEMP = HOLD → Next (ii) HOLD → Next = TEMP → Next (iii) Set free the node TEMP, which is deleted (iv) Exit (b) HOLD = HOLD → Next 5. if ((HOLD → next → DATA) == DATA) (a) TEMP = HOLD → Next (b) Set free the node TEMP, which is deleted (c) HOLD → Next = NULL (d) Exit 6. Display "DATA not found" 7. Exit
--	--

<b>ALGORITHM FOR SEARCHING A NODE</b> Suppose, ☐ START is the address of the first node in the linked list. ☐ DATA is the information to be searched. ☐ After searching, if the DATA is found, POS will contain the corresponding position in the list. 1. Input the DATA to be searched 2. Initialize TEMP = START; POS = 1; 3. Repeat the step 4, 5 and 6 until (TEMP is equal to NULL) 4. If (TEMP → DATA is equal to DATA) (a) Display "The data is found at POS" <b>B.SC-Computer Science III Semester Data Structures</b> <a href="https://mguugcs.blogspot.in">https://mguugcs.blogspot.in</a> 28 (b) Exit 5. TEMP = TEMP → Next 6. POS = POS+1 7. If (TEMP is equal to NULL) (a) Display "The data is not found in the list" 8. Exit	<b>ALGORITHM FOR DISPLAY ALL NODES</b> Suppose START is the address of the first node in the linked list. Following algorithm will visit all nodes from the START node to the end. 1. If (START is equal to NULL) (a) Display "The list is Empty" (b) Exit 2. Initialize TEMP = START 3. Repeat the step 4 and 5 until (TEMP == NULL) 4. Display "TEMP → DATA" 5. TEMP = TEMP → Next 6. Exit
---	---

<b>1</b> <b>//The following C++ program will implement a Single Linked List operations:</b> <pre>#include&lt;iostream.h&gt; #include&lt;conio.h&gt; #include&lt;process.h&gt; class Linked_List { //Structure declaration for the node struct node {</pre>	<b>2</b> <b>//following function will add new element at the beginning</b> <pre>void Linked_List::AddAtBeg(int data) { struct node *tmp; tmp=(struct node*)new(struct node); tmp-&gt;info=data; tmp-&gt;link=start; start=tmp; }/*End of addatbeg()*/</pre>
--	---

<pre> int info; struct node *link; }; //private structure variable declared struct node *start; public: Linked_List()//Constructor defined { start = NULL; } //public fucntion declared void Create_List(int); void AddAtBeg(int); void AddAfter(int,int); void Delete(); void Count(); void Search(int); void Display(); void Reverse(); }; //This a new linked list of elements void Linked_List::Create_List(int data) { struct node *q,*tmp; //New node is created with new operator tmp= (struct node *)new(struct node); tmp-&gt;info=data; tmp-&gt;link=NULL; if (start==NULL) /*If list is empty */ start=tmp; else { /*Element inserted at the end */ q=start; while(q-&gt;link!=NULL) q=q-&gt;link; q-&gt; link=tmp; } }/*End of create_list()*/ </pre>	<pre> //This function will add new element at any specified position void Linked_List::AddAfter(int data,int pos) { struct node *tmp,*q; int i; q=start; //Finding the position in the linked list to insert for(i=0;i&lt;pos-1;i++) { q=q-&gt;link; if (q==NULL) { cout&lt;&lt;"\n\nThere are less than "&lt;&lt;pos&lt;&lt;" elements"; getch(); return; } }/*End of for*/ tmp=(struct node*)new(struct node); tmp-&gt;link=q-&gt;link; tmp-&gt;info=data; q-&gt;link=tmp; }/*End of addafter()*/ //Funtion to delete an element from the list void Linked_List::Delete() { struct node *tmp,*q; int data; if(start==NULL) { cout&lt;&lt;"\n\nList is empty"; getch(); return; } } </pre>
<pre> 3 cout&lt;&lt;"\n\nEnter the element for deletion : "; cin&gt;&gt;data; if(start-&gt;info == data) { tmp=start; start=start-&gt;link; //First element deleted delete(tmp); return; } q=start; while(q-&gt;link-&gt;link != NULL) { if(q-&gt;link-&gt;info==data) //Element deleted in between { tmp=q-&gt;link; q-&gt;link=tmp-&gt;link; delete(tmp); } } </pre>	<pre> 4 void Linked_List::Count() { struct node *q=start; int cnt=0; while(q!=NULL) { q=q-&gt;link; cnt++; } cout&lt;&lt;"Number of elements are \n"&lt;&lt;cnt; getch(); }/*End of count() */ void Linked_List::Reverse() { struct node *p1,*p2,*p3; if(start-&gt;link==NULL) /*only one element*/ return; p1=start; p2=p1-&gt;link; p3=p2-&gt;link; </pre>

```

return;
}
q=q->link;
}/*End of while */
if(q->link->info==data) //Last element
deleted
{
tmp=q->link;
delete(tmp);
q->link=NULL;
return;
}
cout<<"\n\nElement "<<data<<" not
found";
getch();
}/*End of del()*/
void Linked_List::Display()
{
struct node *q;
if(start == NULL)
{
cout<<"\n\nList is empty";
return;
}
q=start;
cout<<"\n\nList is : ";
while(q!=NULL)
{
cout<<q->info;
q=q->link;
}
cout<<"\n";
getch();
}/*End of display() */

```

```

p1->link=NULL;
p2->link=p1;
while(p3!=NULL)
{
p1=p2;
p2=p3;
p3=p3->link;
p2->link=p1;
}
start=p2;
}/*End of rev()*/
void Linked_List::Search(int data)
{
struct node *ptr = start;
int pos = 1;
while(ptr!=NULL)
{
if(ptr->info==data)
{
cout<<"\n\nItem "<<data<<" found at
position
"<<pos;
getch();
return;
}
ptr = ptr->link;
pos++;
}
if(ptr == NULL)
cout<<"\n\nItem "<<data<<" not found in
list";
getch();
}

```

```

5
void main()
{
int choice,n,m,position,i;
Linked_List po;
while(1)
{
clrscr();
cout<<"1.Create List\n";
cout<<"2.Add at begining\n";
cout<<"3.Add after \n";
cout<<"4.Delete\n";
cout<<"5.Display\n";
cout<<"6.Count\n";
cout<<"7.Reverse\n";
cout<<"8.Search\n";
cout<<"9.Quit\n";
cout<<"\nEnter your choice:";
cin>>choice;
switch(choice)
{
case 1:
cout<<"\n\nHow many nodes you want:";
cin>>n;
for(i=0;i<n;i++)
{
cout<<"\nEnter the element:";
cin>>m;
po.Create_List(m);
}

```

```

6
case 3:
cout<<"\n\nEnter the element:";
cin>>m;
cout<<"\nEnter the position after which
this
element is inserted:";
cin>>position;
po.AddAfter(m,position);
break;
case 4:
po.Delete();
break;
case 5:
po.Display();
break;
case 6:
po.Count();
break;
case 7:
po.Reverse();
break;
case 8:
cout<<"\n\nEnter the element to be
searched:";
cin>>m;
po.Search(m);
break;
case 9:
exit(0);

```

```
break;
case 2:
cout<<"\n\nEnter the element:";
cin>>m;
po.AddAtBeg(m);
break;
```

```
default:
cout<<"\n\nWrong choice";
}/*End of switch */
}/*End of while */
}/*End of main() */
```

## DOUBLY LINKED LIST

A doubly linked list is one in which all nodes are linked together by multiple links. It helps in accessing both the successor (next) and predecessor (previous) node for any arbitrary node within the list. Every node in the doubly linked list has three fields:

**LeftPointer, RightPointer and DATA.**

The following figure shows a typical doubly linked list.

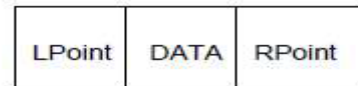


Fig. A typical doubly linked list node

LPoint will point to the node in the left side (or previous node) that is LPoint will hold the address of the previous node.

RPoint will point to the node in the right side (or next node) that is RPoint will hold the address of the next node. DATA will store the information of the node.



Fig. Doubly Linked List



Fig. Memory Representation of Doubly Linked List

## Representation of a Doubly linked List:

A node in the double linked list can be represented with the following declarations:

```
struct Node{
    int DATA;
    struct Node *RChild;
    struct Node *LChild;
};
typedef struct node *NODE;
```

The following figure illustrates the insertion and deletion of nodes:



Fig. Add(20)

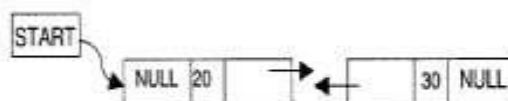


Fig Insert (30) at the end



Fig Insert (10) at the beginning

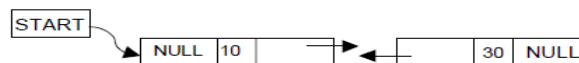


Fig Delete a node at the 2nd position

## ALGORITHM FOR INSERTING A NODE

## ALGORITHM FOR DELETING A NODE

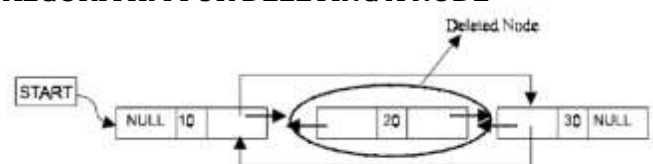


Fig. Delete a node at the 2nd position

Suppose,

➤ START is the address of the first node in the

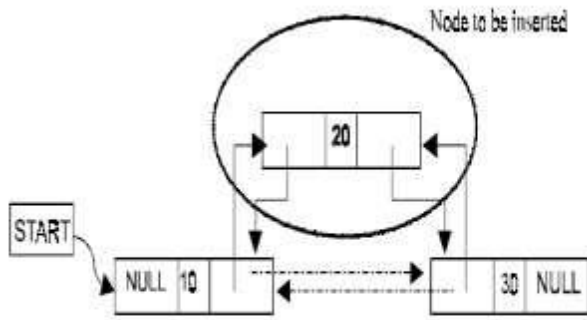


Fig. 5.31. Insert a node at the 2nd position

Suppose START is the first position in linked list. Let DATA be the element to be inserted in the new node. POS is the position where the New Node is to be inserted. TEMP is a temporary pointer to hold the node address.

1. Input the DATA and POS
2. Initialize TEMP = START; i = 0
3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4. TEMP = TEMP → RPoint; i = i + 1
5. If (TEMP not equal to NULL) and (i equal to POS)
  - (a) Create a New Node
  - (b) NewNode → DATA = DATA
  - (c) NewNode → RPoint = TEMP → RPoint
  - (d) NewNode → LPoint = TEMP
  - (e) (TEMP → RPoint) → LPoint = NewNode
  - (f) TEMP → RPoint = New Node
6. Else
  - (a) Display "Position NOT found"
7. Exit

linked list.

- Let POS is the position of the node to be deleted.
- TEMP is the temporary pointer to hold the address of the node.
- After deletion, DATA will contain the information on the deleted node.

1. Input the POS
2. Initialize TEMP = START; i = 0
3. Repeat the step 4 if (i less than POS) and (TEMP is not equal to NULL)
4. TEMP = TEMP → RPoint; i = i + 1
5. If (TEMP not equal to NULL) and (i equal to POS)
  - (a) Create a New Node
  - (b) NewNode → DATA = DATA
  - (c) NewNode → RPoint = TEMP → RPoint
  - (d) NewNode → LPoint = TEMP
  - (e) (TEMP → RPoint) → LPoint = NewNode
  - (f) TEMP → RPoint = New Node
6. Else
  - (a) Display "Position NOT found"
7. Exit

**The following C++ program illustrates a doubly linked list operations:**

```

1
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
class Dllist
{
private:
    struct Node
    {
        string name;
        Node* next;
        Node* prev;
    };
    Node* head;
    Node* last;
public:
    Dllist()
    {
        head = NULL;
        last = NULL;
    }
    bool empty() const { return head==NULL; }
    friend ostream& operator<<(ostream&
, const
Dllist& );

```

```

2
Node* curr;
curr = head;
while( s>curr->name && curr->next !=
last-
>next) curr = curr->next;
if(curr == head)
{
    Node* temp = new Node;
    temp->name = s;
    temp->prev = curr;
    temp->next = NULL;
    head->next = temp;
    last = temp;
    // cout<<" Inserted "<<s<<" After
"<<curr->name<<endl;
}
else
{
    if(curr == last && s>last->name)
    {
        last->next = new Node;
        (last->next)->prev = last;
        last = last->next;
        last->next = NULL;
        last->name = s;
        // cout<<" Added "<<s<<" at the end

```

<pre> void Insert(const string&amp; ); void Remove(const string&amp; ); }; void Dllist::Insert(const string&amp; s) { // Insertion into an Empty List. if(empty()) { Node* temp = new Node; head = temp; last = temp; temp-&gt;prev = NULL; temp-&gt;next = NULL; temp-&gt;name = s; } else { </pre>	<pre> "&lt;&lt;endl; } else { Node* temp = new Node; temp-&gt;name = s; temp-&gt;next = curr; (curr-&gt;prev)-&gt;next = temp; temp-&gt;prev = curr-&gt;prev; curr-&gt;prev = temp; // cout&lt;&lt;" Inserted "&lt;&lt;s&lt;&lt;" Before "&lt;&lt;curr- &gt;name&lt;&lt;endl; } } }}</pre>
<pre> 3 ostream&amp; operator&lt;&lt;(ostream&amp; ostr, const Dllist&amp; dl ) { if(dl.empty()) ostr&lt;&lt;" The list is empty" &lt;&lt; endl; else { Dllist::Node* curr; for(curr = dl.head; curr != dl.last- &gt;next; curr=curr-&gt;next) ostr&lt;&lt;curr-&gt;name&lt;&lt;" "; ostr&lt;&lt;endl; return ostr; } } void Dllist::Remove(const string&amp; s) { bool found = false; if(empty()) { cout&lt;&lt;" This is an empty list! "&lt;&lt;endl; return; } else { Node* curr; for(curr = head; curr != last-&gt;next; curr = curr-&gt;next) { if(curr-&gt;name == s) { found = true; break; } } if(found == false) { cout&lt;&lt;" The list does not contain specified Node"&lt;&lt;endl; return; } else { if (curr == head &amp;&amp; found) </pre>	<pre> 4 if (curr == last &amp;&amp; found) { last = curr-&gt;prev; delete curr; return; } (curr-&gt;prev)-&gt;next = curr-&gt;next; (curr-&gt;next)-&gt;prev = curr-&gt;prev; delete curr; } } } int main() { Dllist dl; int ch; string temp; while(1) { cout&lt;&lt;endl; cout&lt;&lt;" Doubly Linked List Operations "&lt;&lt;endl; cout&lt;&lt;" ----- "&lt;&lt;endl; cout&lt;&lt;" 1. Insertion "&lt;&lt;endl; cout&lt;&lt;" 2. Deletion "&lt;&lt;endl; cout&lt;&lt;" 3. Display "&lt;&lt;endl; cout&lt;&lt;" 4. Exit "&lt;&lt;endl; cout&lt;&lt;" Enter your choice : "; cin&gt;&gt;ch; switch(ch) { case 1: cout&lt;&lt;" Enter Name to be inserted : "; cin&gt;&gt;temp; dl.Insert(temp); break; case 2: cout&lt;&lt;" Enter Name to be deleted : "; cin&gt;&gt;temp; dl.Remove(temp); break; case 3: cout&lt;&lt;" The List contains : "; cout&lt;&lt;dl; break; case 4: system("pause"); </pre>

```

{
if(curr->next != NULL)
{
head = curr->next;
delete curr;
return;
}
{
delete curr;
head = NULL;
last = NULL;
return;
}}

```

```

return 0;
break;
}
}
return 0;
}

```

### CIRCULAR LINKED LIST

A circular linked list is one, which has no beginning and no end. A singly linked list can be made a circular linked list by simply storing the address of the very first node in the linked field of the last node. The following figure shows a circular linked list:

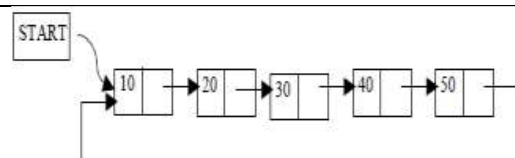


Fig. Circular Linked list

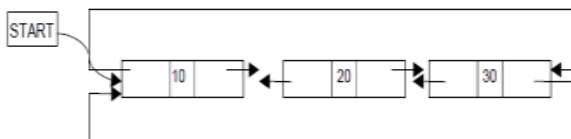


Fig. Circular Doubly Linked list

A circular doubly linked list has both the successor pointer and predecessor pointer in circular manner. The following figure shows a circular doubly linked list:

**The following C++ program illustrates the Circular Linked List:**

```

1
#include<iostream.h>
#include<process.h>
//class is created for the circular
linked list
class Circular_Linked
{
//structure node is created
struct node
{
int info;
struct node *link;
};
struct node *last;
typedef struct node *NODE;
public:
//Constructor is defined
Circular_Linked()
{
last=NULL;
}
void create_list(int);
void addatbeg(int);
void addafter(int,int);
void del();
void display();
};
//A circular list created in this
function
void Circular_Linked::create_list(int
num)
{
NODE q,tmp;
//New node is created
tmp = (NODE)new(struct node);
tmp->info = num;

```

```

2
tmp->link = q->link;
tmp->info = num;
q->link = tmp;
if(q==last) /*Element inserted at the
end*/
last=tmp;
}/*End of addafter()*/
//Function to delete a node from the
circular linked
list
void Circular_Linked::del()
{
int num;
if(last == NULL)
{
cout<<"\nList underflow\n";
return;
}
cout<<"\nEnter the number for
deletion:";
cin>>num;
NODE tmp,q;
if( last->link == last && last->info ==
num)
/*Only one element*/
{
tmp = last;
last = NULL;
//deleting the node
delete(tmp);
return;
}
q = last->link;
if(q->info == num)
{

```



```

if (last == NULL)
{
last = tmp;
tmp->link = last;
}
else
{
tmp->link = last->link; /*added at the
end of list*/
last->link = tmp;
last = tmp;
}
}/*End of create_list()*/
//This function will add new node at the
beginning
void Circular_Linked::addatbeg(int num)
{
NODE tmp;
tmp = (NODE)new(struct node);
tmp->info = num;
tmp->link = last->link;
last->link = tmp;
}/*End of addatbeg()*/
//Function to add new node at any
position of the
circular list
void Circular_Linked::addafter(int
num,int pos)
{
NODE tmp,q;
int i;
q = last->link;
//finding the position to insert a new
node
for(i=0; i < pos-1; i++)
{
q = q->link;
if (q == last->link)
{
cout<<"There are less than "<<pos<<"
elements\n";
return;
}
}/*End of for*/
//creating the new node
tmp = (NODE)new(struct node);

```

```

3
while(q != last)
{
cout<< q->info;
q = q->link;
}
cout<<"\n"<<last->info;
}/*End of display()*/
void main()
{
int choice,n,m,po,i;
Circular_Linked co;//Object is created
for the class
while(1)
{
//Menu options
cout<<"\n1.Create List\n";
cout<<"2.Add at begining\n";

```

```

tmp = q;
last->link = q->link;
//deleting the node
delete(tmp);
return;
}
while(q->link != last)
{
if(q->link->info == num) /*Element
deleted in
between*/
{
tmp = q->link;
q->link = tmp->link;
delete(tmp);
cout<<"\n"<<num<<" deleted\n";
return;
}
q = q->link;
}/*End of while*/
if(q->link->info == num) /*Last element
deleted
q->link=last*/
{
tmp = q->link;
q->link = last->link;
delete(tmp);
last = q;
return;
}
cout<<"\nElement "<<num<<" not found\n";
}/*End of del()*/
//Function to display all the nodes in
the circular
linked list
void Circular_Linked::display()
{
NODE q;
if(last == NULL)
{
cout<<"\nList is empty\n";
return;
}
q = last->link;
cout<<"\nList is:\n"

```

```

4
break;
case 2:
cout<<"\nEnter the element:";
cin>>m;
co.addatbeg(m);
break;
case 3:
cout<<"\nEnter the element:";
cin>>m;
cout<<"\nEnter the position after which
this
element is inserted:";
cin>>po;
co.addafter(m,po);
break;
case 4:
co.del();

```

```

cout<<"3.Add after \n";
cout<<"4.Delete\n";
cout<<"5.Display\n";
cout<<"6.Quit\n";
cout<<"\nEnter your choice:";
cin>>choice;
switch(choice)
{
case 1:
cout<<"\nHow many nodes you want:";
cin>>n;
for(i=0; i < n;i++)
{
cout<<"\nEnter the element:";
cin>>m;
co.create_list(m);
}

```

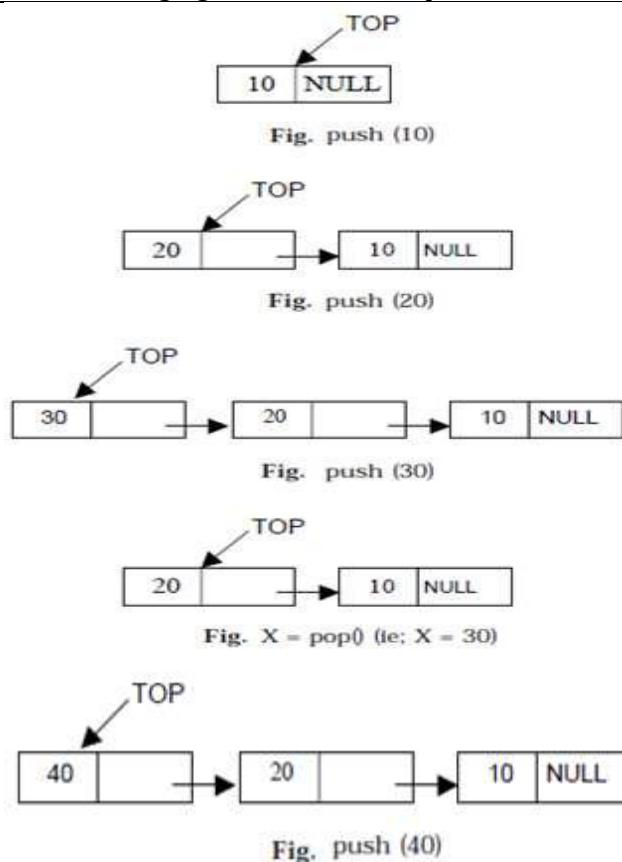
```

break;
case 5:
co.display();
break;
case 6:
exit(0);
default:
cout<<"\nWrong choice\n";
}/*End of switch*/
}/*End of while*/
}/*End of main()*/

```

### Stack using Single Linked List

The following figure shows the implementation issues of the stack (Last In First Out - LIFO) using linked list:



### ALGORITHM FOR PUSH OPERATION

Suppose,

- TOP is a pointer, which is pointing towards the topmost element of the stack.
- TOP is NULL when the stack is empty.
- DATA is the data item to be pushed.

1. Input the DATA to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = TOP
5. TOP = NewNode
6. Exit

### ALGORITHM FOR POP OPERATION

Suppose,

- TOP is a pointer, which is pointing towards the topmost element of the stack.
- TOP is NULL when the stack is empty.
- TEMP is pointer variable to hold any nodes address.
- DATA is the information on the node which is just deleted.

1. if (TOP is equal to NULL)
  - (a) Display "The stack is empty"
2. Else
  - (a) TEMP = TOP
  - (b) Display "The popped element TOP → DATA"
  - (c) TOP = TOP → Next
  - (d) TEMP → Next = NULL
  - (e) Free the TEMP node
3. Exit

**The following program illustrates the implementation of a stack using the Linked List:**

```

1
#include<conio.h>
#include<iostream.h>
#include<process.h>
//Class is created for the linked list
class Stack_Linked
{
//Structure is created for the node
struct node
{

```

```

2
//This is to display all the element in
the stack
void Stack_Linked::display()
{
if(top==NULL)//Checking whether the
stack is
empty or not
cout<<"\nStack is empty\n";
else

```

```

int info;
struct node *link;//A link to the next
node
};
//A variable top is been declared for
the structure
struct node *top;
//NODE is defined as the data type of
the structure
node
typedef struct node *NODE;
public:
//Constructor is defined for the class
Stack_Linked()
{
//top pointer is initialized
top=NULL;
}
//function declarations
void push();
void pop();
void display();
};
//This function is to perform the push
operation
void Stack_Linked::push()
{
NODE NewNode;
int pushed_item;
//A new node is created dynamically
NewNode=(NODE)new(struct node);
cout<<"\nInput the new value to be
pushed on the
stack:";
cin>>pushed_item;
NewNode->info=pushed_item;//Data is
pushed to
the stack
NewNode->link=top;//Link pointer is set
to the
next node
top=NewNode;//Top pointer is set
}/*End of push()*/
//Following function will implement the
pop
operation
void Stack_Linked::pop()
{
NODE tmp;
if(top == NULL)//checking whether the
stack is
empty or not
cout<<"\nStack is empty\n";
else
{ tmp=top;//popping the element
cout<<"\nPopped item is:"<<tmp->info;
top=top->link;//resetting the top
pointer
tmp->link=NULL;
delete(tmp);//freeing the popped node
}
}/*End of pop()*/

```

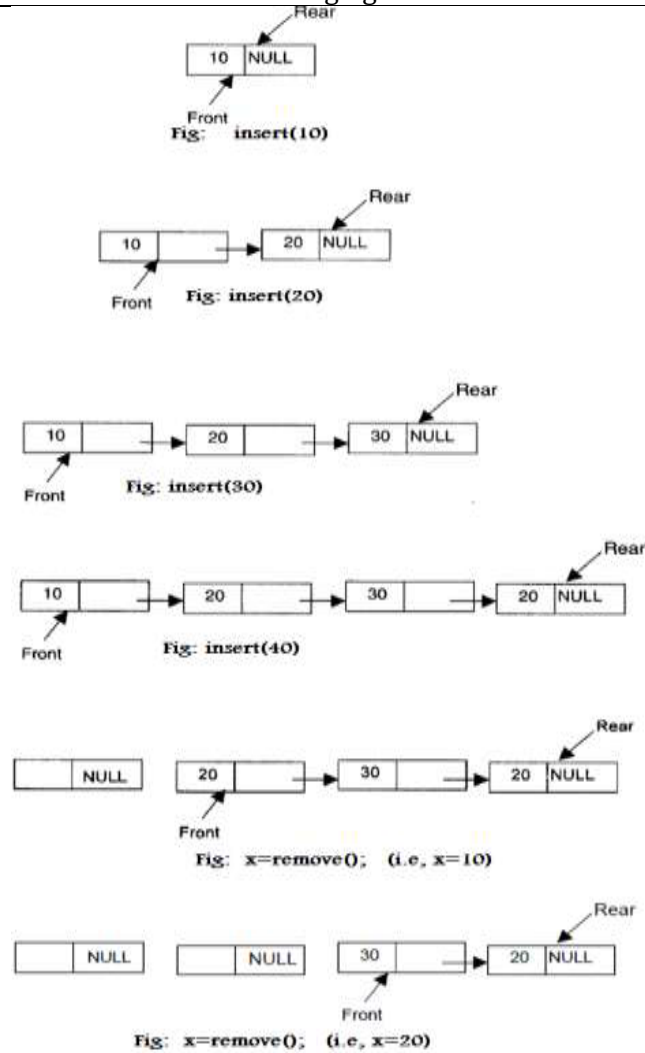
```

{
NODE ptr=top;
cout<<"\nStack elements:\n";
while(ptr != NULL)
{
cout<<"\n"<<ptr->info;
ptr = ptr->link;
}/*End of while */
}/*End of else*/
}/*End of display()*/
void main()
{
char opt;
int choice;
Stack_Linked So;
do
{
clrscr();
//The menu options are listed below
cout<<"\n1.PUSH\n";
cout<<"2.POP\n";
cout<<"3.DISPLAY\n";
cout<<"4.EXIT\n";
cout<<"\nEnter your choice : ";
cin>>choice;
switch(choice)
{
case 1:
So.push();//push function is called
break;
case 2:
So.pop();//pop function is called
break;
case 3:
So.display();//display function is
called
break;
case 4:
exit(1);
default:
cout<<"\nWrong choice\n";
}/*End of switch */
cout<<"\n\nDo you want to continue (Y/y)
= ";
cin>>opt;
}while((opt == 'Y') || (opt == 'y'));
}/*End of main() */

```

## Queue using Linked List

Queue is a First In First Out [FIFO] data structure. Implementation issues of the queue (FIFO) using linked list is illustrated in the following figures:



### ALGORITHM FOR PUSHING AN ELEMENT TO A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element to be pushed.

1. Input the DATA element to be pushed
2. Create a New Node
3. NewNode → DATA = DATA
4. NewNode → Next = NULL
5. If (REAR not equal to NULL)
  - (a) REAR → next = NewNode;
6. REAR = NewNode;
7. Exit

### ALGORITHM FOR POPPING AN ELEMENT FROM A QUEUE

REAR is a pointer in queue where the new elements are added. FRONT is a pointer, which is pointing to the queue where the elements are popped. DATA is an element popped from the queue.

1. If (FRONT is equal to NULL)
  - (a) Display "The Queue is empty"
2. Else
  - (a) Display "The removed element is FRONT → DATA"
  - (b) If (FRONT is not equal to REAR)
    - (i) FRONT = FRONT → Next
  - (c) Else
  - (d) FRONT = NULL;
3. Exit

*The following program illustrates the implementation of a Queue using Linked Lists:*

```
1
#include<iostream.h>
#include<conio.h>
#include<malloc.h>
//class is created for the queue
class Queue_Linked
{
//A structure is created for the node in
queue
struct queu
{
int info;
struct queu *next;//Next node address
};
struct queu *front;
struct queu *rear;
typedef struct queu *NODE;
public:
//Constructor is created
Queue_Linked()
{
front = NULL;
rear = NULL;
}
void insert();
void remove();
```

```
2
if (front != rear)
front=front->next;
else
front = NULL;
}
}
//Function to display the element of the
queue
void Queue_Linked::traverse()
{
//The queue is empty when the front
pointer is
NULL
if (front==NULL)
cout<<"\nThe Queue is empty";
else
{
NODE Temp_Front=front;
cout<<"\nThe element(s) is/are = ";
while(Temp_Front != rear)
{
cout<<Temp_Front->info;
Temp_Front=Temp_Front->next;
};
cout<<Temp_Front->info;
```

<pre> void traverse(); }; //This function will insert an element into the queue void Queue_Linked::insert() { NODE NewNode; //New node is created to insert the data NewNode=(NODE)malloc(sizeof(struct queu)); cout&lt;&lt;"\nEnter the no to be inserted = "; cin&gt;&gt;NewNode-&gt;info; NewNode-&gt;next=NULL; //setting the rear pointer if (rear != NULL) rear-&gt;next = NewNode; rear=NewNode; if (front == NULL) front = rear; } //This function will remove the element from the queue void Queue_Linked::remove() { //The Queue is empty when the front pointer is NULL if (front == NULL) { cout&lt;&lt;"\nThe Queue is empty"; rear = NULL; } else { //Front element in the queue is removed cout&lt;&lt;"\nThe removed element is = "&lt;&lt;front- &gt;info; </pre>	<pre> } } void main() { int choice; char option; Queue_Linked Qo; do { clrscr(); cout&lt;&lt;"\n1. insert\n"; cout&lt;&lt;"2. remove\n"; cout&lt;&lt;"3. DISPLAY\n"; cout&lt;&lt;"\nEnter your choice = "; cin&gt;&gt;choice; switch(choice) { case 1: //calling the insert function Qo.insert(); break; case 2: //calling the remove function by passing //front and rear pointers Qo.remove(); break; case 3: Qo.traverse(); break; } cout&lt;&lt;"\n\nPress (Y/y) to continue = "; cin&gt;&gt;option; }while(option == 'Y'    option == 'y'); } </pre>
---	---

### Application of Linked List

#### Garbage collection

Memory is just an array of words. After a series of memory allocations and de-allocations, there are blocks of free memory scattered throughout the available heap space. To be able to reuse this memory, the memory allocator will usually link the freed blocks together in a free list.

Operating system's memory management module consists of unused memory cells. This list implemented as a linked organization is called the *list of available space*, *free storage list*, or the *free pool*.

For good memory utilization, the operating system periodically collects all the free blocks and inserts into the free pool. Any technique that does this collection is called *garbage collection*.

**Garbage collection usually takes place in two phases.**

- First, the process runs through all the lists, tagging those cells, which are currently in use.
- In the second phase, the process runs through memory, collecting all untagged blocks and inserting the same in free pool. In general, garbage collection takes place:
  - when either overflow or underflow occurs. And also
  - when the CPU is idle, the garbage collection starts.

**Overflow:** Sometimes, a new data node is to be inserted into data structure, but there is no available space, that is, free pool is empty. This situation is called *overflow*.

**Underflow:** This refers to the situation where the programmer wants to delete a node from the empty list.

A circular DLL is the most suitable data structure for garbage collection.. It allows the process of search to be unending.

## UNIT-III

## Trees

A *tree* is a non-linear data structure that emulates a tree structure with a set of linked nodes.

- Trees are used popularly in computer programming.
- They can be used for improving database search times,
- In game programming,
- 3D graphics programming,
- Arithmetic scripting languages
- Data compression
- File systems.

## Basic Terminology

A graph  $G$  consists of a non-empty set  $V$ , a set  $E$ , and a mapping from the set  $E$  to set  $V$ .

- Here,  $V$  is the set of *nodes*, also called as *vertices points*, of the graph, and
- $E$  is the set of edges of the graph.
- For *finite graphs*,  $V$  and  $E$  are finite. We can represent them as  $G = (V, E)$ .

## Adjacent Nodes

Any two nodes that are connected with an edge are called as *adjacent nodes*.

## Directed and Undirected Graphs

- In a graph  $G(V, E)$ , an edge that is directed from one node to another is called a *directed Edge*.
- An edge that has the no specific direction is called an *undirected edge*.
- A graph where every edge is directed is called as a *directed graph* or *digraph*.
- A graph where every edge is undirected is called as an *undirected graph*.
- If some of edges are directed and some are undirected in a graph, the graph is called as a *mixed graph*.

## Parallel Edges and Multi graph

In the above figure, the edges  $e_1$ ,  $e_2$ , and  $e_3$  are incident to vertices  $a$  and  $b$ . Such edges are called as **parallel edges**. Here,  $e_1$ ,  $e_2$ , and  $e_3$  are three parallel edges. In addition,  $e_5$  and  $e_6$  are two parallel edges. Any graph that contains parallel edges is called a **multi graph**. A graph that has no parallel edges is called a **simple graph**.

## Weighted Graph

A graph where weights are assigned to every edge is called a *weighted graph*.

## Null Graph and Isolated Vertex

- In a graph, a node that is not adjacent to any other node is called an *isolated node*.
- A graph containing only isolated nodes is called a *null graph*.

## Forest and Trees

A *forest* is a graph that contains no cycles, and a connected forest is a *tree*. For example, the following figure shows a forest with three components, each of which is a tree:



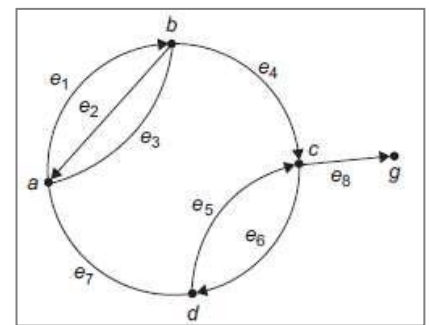
**Directed tree** An acyclic directed graph is a *directed tree*.

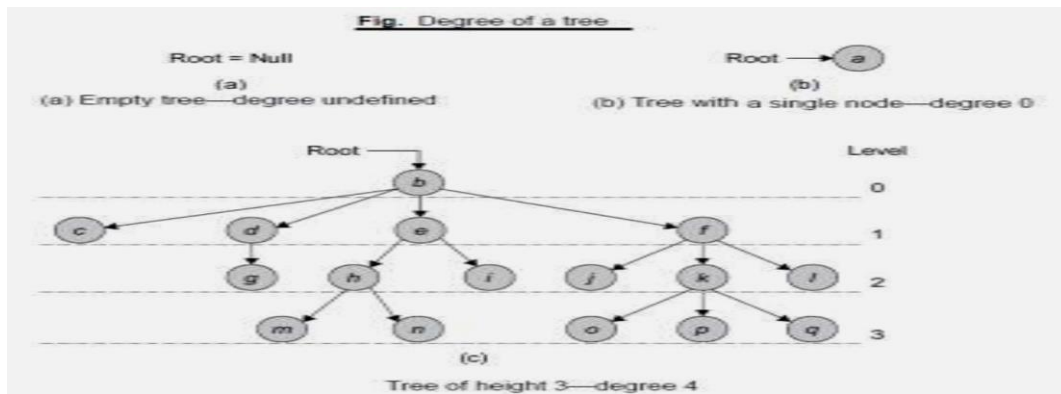
- **Root** A directed tree has one node called its *root*, with indegree zero, whereas for all other nodes, the indegree is 1
- **Terminal node (leaf node)** In a directed tree, any node that has an out degree zero is a *terminal node*. The terminal node is also called as *leaf node* (or *external node*).
- **Branch node (internal node)** All other nodes whose out degree is not zero are called as *branch nodes*.
- **Level of node** The level of any node is its path length from the root. The level of the root of a directed tree is zero, whereas the level of any node is equal to its distance from the root. Distance from the root is the number of edges to be traversed to reach the root.

## General Tree

A tree  $T$  is defined recursively as follows:

1. A set of zero items is a tree, called the *empty tree* (or null tree).
2. If  $T_1, T_2, \dots, T_n$  are  $n$  trees for  $n > 0$  and  $R$  is a *node*, then the set  $T$  containing  $R$  and the trees  $T_1, T_2, \dots, T_n$  are a tree. Within  $T$ ,  $R$  is called the *root* of  $T$ , and  $T_1, T_2, \dots, T_n$  are called *subtrees*. Consider the following trees:





For the above tree in (c), the root node has four subtrees.

- ❖ The roots of these subtrees are called the *children* of the root.
- ❖ There are 16 nodes in the tree, so there are 15 non-empty subtrees.
- ❖ The nodes with no subtrees are called *terminal nodes* or more commonly, *leaves*.
- ❖ There are 10 leaves in the above tree

### Representation of a General Tree

We can use either a sequential organization or a linked organization for representing a tree. The following figure shows a general tree:

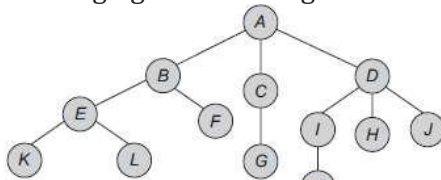


Fig. Sample tree

The list representation of this tree is shown in the following figure:

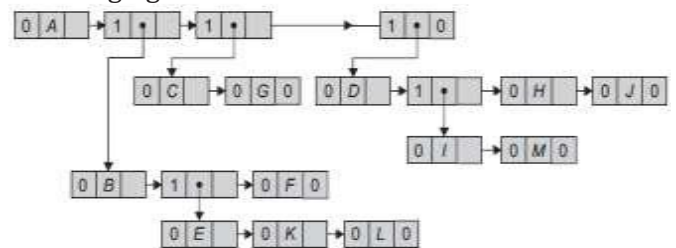


Fig. List representation

### Binary Tree

Most commonly used classes of tree is a binary tree.

- ❖ A binary tree has the degree two, with each node having at most two children. A binary tree is either An empty tree or
- ❖ Consists of a node, called *root*, and two children, *left* and *right*, each of which is itself binary tree.

The following figure shows a binary tree:

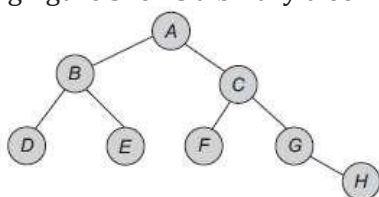


Fig. Binary tree

### Properties of a Binary Tree

A tree is a connected acyclic graph.

The following are some simple properties of trees: Let  $T$  be a tree. Then the following properties hold true:

1. There exists a **unique path between every two vertices**.
2. **The number of vertices is one more than the number of edges** in the tree.
3. A tree with two or more vertices has at least two leaves.

Let us refer to the following figure for proving these properties.



Fig : Binary trees

(a) Sample 1

(b) Sample 2

### Other Properties Of Binary Tree

1. The maximum number of nodes of level  $i$  in a binary tree is  $2^i - 1$ , where  $i \geq 1$ .
2. The maximum number of nodes of depth  $d$  in a binary tree is  $2^d - 1$ , where  $d \geq 1$ .

### Binary Tree Abstract Data Type

We have defined a binary tree. Let us now define it as an abstract data type (ADT), which includes a list of operations that process it. ADT btree

1. Declare create() btree
2. makebtree(btree, element, btree) btree
3. isEmpty(btree) boolean
4. leftchild(btree) btree
5. rightchild(btree) btree
6. data(btree) element



```

8. isEmpty(create) = true
9. isEmpty(makebtree(l,e,r)) = false
10. leftchild(create()) = error
11. rightchild(create()) = error
12. leftchild(makebtree(l,e,r)) = l
13. rightchild(makebtree(l,e,r)) = r
14. data(makebtree(l,e,r)) = e
15. end
end btree

```

### Implementation of Binary Trees

The implementation of a binary tree should represent the hierarchical relationship between a parent node and its left and right children. Linked implementation is more popular than the sequential structure due to the following two main reasons:

1. A binary tree has a natural implementation in a linked storage.
2. The linked structure is more convenient for insertions and deletions.

### Array Implementation of Binary Trees

- ❖ It can store the nodes level-by-level, starting from the level 0 (root).
- ❖ It requires sequential numbering of the nodes.
- ❖ A complete binary tree of height  $h$  has  $(2^{h+1} - 1)$  nodes in it.
- ❖ The nodes can be stored in a one dimensional array.
- ❖ The root node is stored in the first memory location in the array. The following rules can be used to decide the location of any  $i$ th node of a tree: For any node with index  $i$ ,  $0 \leq i \leq n-1$ ,

1.  $\text{Parent}(i) = \lfloor (i-1)/2 \rfloor$  if  $i \neq 0$ ; if  $i = 0$ , then it is the root that has no parent.
2.  $\text{Lchild}(i) = 2 \times i + 1$  if  $2i + 1 \leq n - 1$ ; if  $2i \geq n$ , then  $i$  has no left child.
3.  $\text{Rchild}(i) = 2i + 2$  if  $2i + 2 \leq n - 1$ ; if  $(2i + 1) \geq n$ , then  $i$  has no right child.

Let us consider the following complete binary tree:

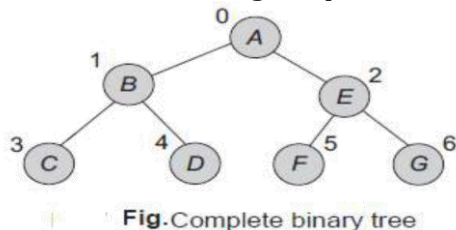


Fig. Complete binary tree

The following is an Array representation of the above binary tree:

0	1	2	3	4	5	6	7	8
A	B	E	C	D	F	G	-	-

This representation of binary trees is the easiest one. But it has some drawbacks.

**Advantages** of representing binary trees using arrays:

1. Any node can be accessed from any other node.
2. The data is stored without any pointers.
3. Trees in BASIC, Fortran can use array representation only.

**Disadvantages** of representing binary trees using arrays:

1. Majority of the array entries may be empty (Except Full Binary trees)
2. It allows only static representation.
3. The array size cannot be changed during the execution.
3. Inserting a node to it or deleting a node requires processing time.

### Linked Implementation of Binary Trees

In a linked organization all the nodes should be allocated dynamically. Hence, we need each node with data and link fields. Each node of a binary tree has both a left and a right subtree. Each node will have three fields—Lchild, Data, and Rchild. This can be shown in the following figure:



Fig. Tree node

Consider the following binary tree and its linked representation:

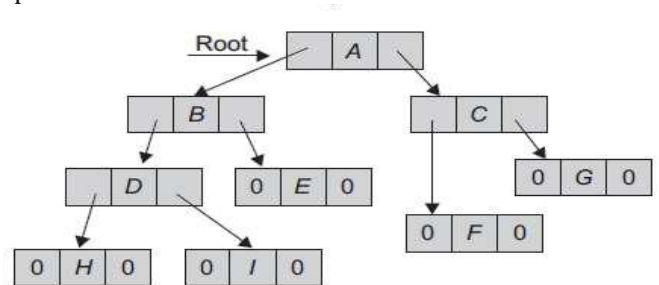
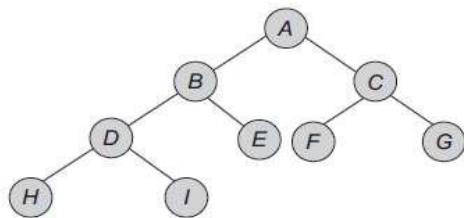


Fig. Sample tree 1 and its linked representation

Here, 0 (zero) at Lchild or Rchild indicates that the respective child is not present.





In this node structure, Lchild and Rchild are the two link fields to store the addresses of left child and right child of a node.

**Advantages** of representing binary trees in linked representations:

1. It overcomes the drawbacks of the sequential representation.
2. The memory is not wasted.
3. Insertions and deletions are more efficient.
4. It is useful for dynamic data.

**Disadvantages** of representing binary trees in linked representations:

1. There is no direct access to any node.
2. It has to be traversed from the root to reach to a particular node.
3. The memory needed per node is more.
4. This cannot be used in some PL's where dynamic memory management is not available.

## Binary Tree Traversal

Traversal of a tree means *stepping through the nodes of a tree*.

- ❖ **Traversal** can be done by means of the connections between parents and children.
- ❖ This is also called **walking the tree**.
- ❖ Traversal means visiting every node of a binary tree.

There are different traversal methods. Such as:

1. Inorder Traversal
2. Postorder Traversal
3. Preorder Traversal

### Preorder Traversal

The preorder traversal is also called as *depth-first traversal*. Preorder Traversal can be defined in the following steps:

#### Preorder (DLR) Algorithm:

1. Visit the root node, say D.
2. Traverse the Left subtree of the node in preorder.
3. Traverse the Right subtree of the node in preorder.

Consider the following expression tree

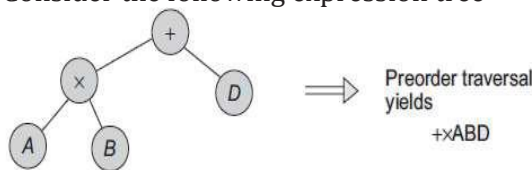


Fig. Expression tree and its preorder traversal

Consider the following binary tree

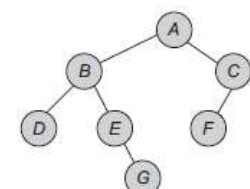


Fig. Binary tree

### Pre order sequence: ABDEGCF

→ The following code implements Preorder traversal:

```

void BinaryTree ::
Preorder(TreeNode*)
{
if(Root != Null)
{
cout << Root->Data;
Preorder(Root->Lchild);
Preorder(Root->Rchild);
}
}
  
```

### Inorder Traversal

The inorder traversal is also called as *symmetric traversal*. In this traversal, the left subtree is visited first ininorder followed by the root and then the right subtree in inorder.

#### Inorder (LDR) Algorithm:

1. Traverse the left subtree of the root node in inorder.
2. Visit the root node *node*.
3. Traverse the right subtree of the root node in inorder.

Consider the following expression tree:

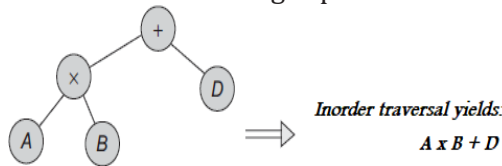


Fig. Expression Tree and its Inorder Traversal

Consider the following binary tree

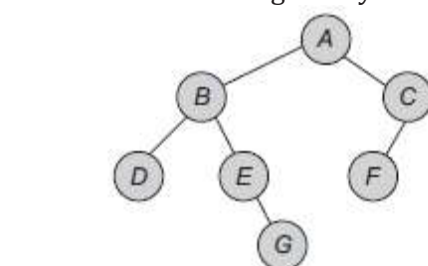


Fig. Binary tree

An *inorder traversal* of the above tree visits the node in the following sequence. **Inorder sequence: D B E G A F C**

The following code implements Inorder traversal:

```

void BinaryTree ::
Inorder(TreeNode*)
{
if(Root != Null)
{
Inorder(Root->Lchild);
cout << Root->Data;
Inorder(Root->Rchild);
}
}
  
```

**Postorder Traversal**

In Postorder traversal, the left subtree is visited first in postorder followed by the right subtree in postorder and then the root.

**Postorder (LRD) Algorithm:**

1. Traverse the root's left child (subtree) of the root node in postorder.
2. Traverse the root's right child (subtree) of the root node in postorder.
3. Visit the root node.

Consider the following expression tree:

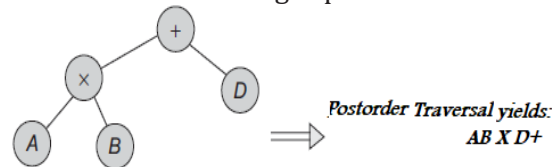


Fig. Expression Tree and its Postorder Traversal

Consider the following binary tree:

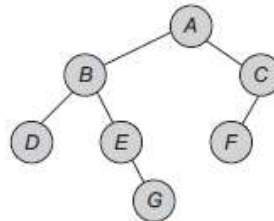


Fig. Binary tree

An *Postorder traversal* of the above tree visits the node in the following sequence. **Postorder sequence: D G E B F C A**

The following code implements Postorder traversal:

```
void BinaryTree ::  
Postorder(TreeNode*)
```

```
{  
if(Root != Null)  
{  
Postorder(Root->Lchild);  
Postorder(Root->Rchild);  
cout << Root->Data;  
}  
}
```

**Applications of Binary Trees**

There is a vast set of applications of the binary trees. Some of them are:

1. Expression tree
2. Huffman tree for coding
3. Decision trees and
4. Gaming trees

**Expression Tree**

A binary tree representing an arithmetic expression is called as *expression tree*.

- ❖ The leaves of an expression tree are *operands*.
- ❖ The branch nodes (internal nodes) represent the operators.
- ❖ A binary tree is most suitable for arithmetic expressions. Consider the following expression E:

Let  $E = ((A \times B) + (C - D)) / (C - E)$

Then the following is an expression tree for the expression E:

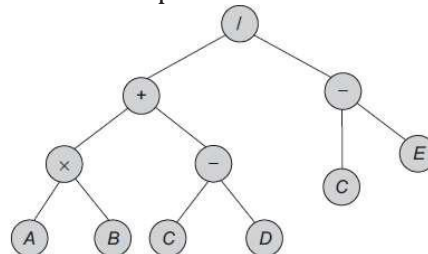


Fig. Expression tree for  $E = ((A \times B) + (C - D)) / (C - E)$

**Decision Tree**

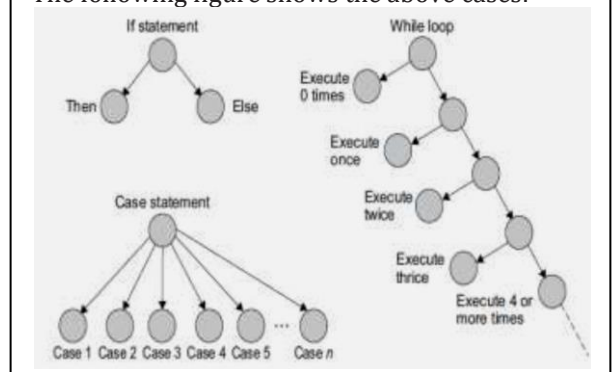
The *decision tree* is a classifier tree where each node is either a branch node or a leaf node. A decision tree can be used for classification by starting at the root of the tree and moving through it until a leaf node that provides the classification of the instance is reached. We can visualize the different ways in which a program may execute through a decision tree:

- We can represent the root of the decision tree with the code that is always run at the start of a program.
- For an **if statement**, there are two children of the root—one if the Boolean expression is true (then clause). And another in case the expression is false (else clause). For a **case statement**,
- A different child is drawn for each different case identified by the code, because different paths are followed for each of these situations.

The **advantages** of decision trees:

1. Decision trees are most suitable for listing all possible decisions.
2. They are suitable for classification.

The following figure shows the above cases:



3. They are popularly used in expert systems.

The **Disadvantages** of decision trees:

1. Decision trees are prone to errors.
2. They are computationally expensive for complex problems.

### Huffman's Coding

Huffman code is a type of optimal prefix code. It is commonly used for lossless data compression. This algorithm has been developed by David A. Huffman. It works by creating a binary tree of nodes. Nodes count depends on the number of symbols.

The algorithm proceeds in a series of rounds. In each round, the algorithm takes the two binary trees with the smallest frequencies and merges them into a single binary tree. It repeats this process until only one tree is left.

### Huffman's algorithm

The algorithm is given as follows:

1. Organize the data into a row as ascending order frequency weights. Each character is the leaf node of a tree.
2. Find two nodes with the smallest combined weights and join them to form the third node. This will form a new two-level tree. The weight of the new third node is the addition of two nodes.
3. Repeat step 2 till all the nodes on every level are combined to form a single tree.

### Gaming Tree

One of the most exciting applications of trees is in games such as tic-tac-toe, chess, nim, checkers, go, and so on. We shall consider tictac- toe as an example for explaining this application of trees.

- The game starts with an empty board
- Each time a player tries for the best move from the given board position.
- Each player is initially assigned a symbol of either 'X' or 'O'.
- Depending on the board position the user has to decide how good the position seems to be for a player.
- Considering all the possible positions, it is possible to construct a tree of the possible board positions, called a *game tree*.

## SEARCHING

The process of locating target data is known as *searching*. The following are the two basic search techniques:

### 1. Sequential search

### 2. Binary search

### Sequential Search / Linear Search

Sequential search is the easiest search technique. Sequential search begins with the first available record and proceeds to the next available record repeatedly until we find the target key or conclude that it is not found. Sequential search is also called as *linear search*. The following algorithm shows the steps in sequential search:

1. Set  $i = 0$ , flag = 0
2. Compare key[i] and target  
if(key[i] = target)  
Set flag = 1, location = i and goto step 5
3. Move to next data element  
 $i = i + 1$
4. if( $i < n$ ) goto step 2
5. if(flag = 1) then return i as position of target located  
else  
report as 'Target not found'
6. stop

The following figure shows a sample sequential unordered data and traces the search for the target data of 89

Index	0	1	2	3	4	5	6	7	8
Elements	23	12	9	10	11	89	78	66	88

Fig. Sequential search for target data of 89

Initially,  $i = 0$  and the target element 89 is to be searched. At each pass, the target 89 is compared with the element at the  $i$ th location till it is found or the index  $i$  exceeds the size. At  $i = 5$ , the search is successful. The following is the C++ code to implement Sequential search:

```
int SeqSearch (int A[max], int key, int n)
{
    int i, flag = 0, position;
```

```

for(i = 0; i < n; i++)
{
if(key == A[i])
{
position = i;
flag = 1;
break;
}
}
if(flag == 1)
return(position);
else
return(-1);
}

```

### **Efficiency of Sequential Search/ Linear Search:**

The total number of comparisons depends on the position of the target data. Average complexity is the sum of number of comparisons for each position of the target data divided by  $n$  and is given as follows:

$$\begin{aligned}
 \text{Average number of comparisons} &= (1 + 2 + 3 + \dots + n)/n \\
 &= (\Sigma n)/n \\
 &= ((n(n + 1))/2) \times 1/n \\
 &= (n + 1)/2
 \end{aligned}$$

### **Pros and Cons of Sequential Search**

#### **Pros**

1. A simple and easy method
2. Efficient for small lists
3. Suitable for unsorted data
4. Suitable for sequential storage etc.
5. Best case is one comparison, worst case is  $n$  comparisons, and average case is  $(n + 1)/2$  comparisons
6. Time complexity is in the order of  $n$  denoted as  $O(n)$ .

#### **Cons**

1. Highly inefficient for large data
2. In the case of ordered data other search techniques such as binary search are found more suitable.

### **Binary search**

Binary search is an extremely efficient algorithm because it searches "data" in minimum possible comparisons. Consider that the list is sorted in ascending order. In binary search algorithm, to search for a particular element:

- ❖ It is first compared with the element at the middle position, and if it is found, the search is successful,
- ❖ else if the middle position value is greater than the target, the search will continue in the first half of the list;
- ❖ Otherwise, the target will be searched in the second half of the list.
- ❖ The same process is repeated for one of the halves of the list till the list is reduced to size one.

The following algorithm illustrates the Binary Search:

```

1. Let n be size of the list
Let target be the element to be searched
Let flag = 0, low = 0, high = n-1
2. if low ≤ high, then
middle = (low + high)/2
else goto step (5)
3. if(key[middle] = target)
Position = middle, flag = 1
Goto step (5)
else if(key[middle] > target) then
high = middle - 1
else
low = middle + 1
4. Goto step(2)
5. if flag = 1
report as target element found at location 'position'
else

```

The following code implements Binary Search:

```

int Binary_Search_non_recursive(int A[], int n,
int key)
{
int low = 0, high = n - 1, mid;
while(low <= high)
{
mid = (low + high)/2;
mid = (first + last)/2)
if(A[mid] == key)
return mid;
else if(key < A[mid])
high = mid - 1;
else
low = mid + 1;
}
return -1;
}

```

report that element is not found in the list

6. stop

### Time Complexity Analysis

Time Complexity is measured by the number  $f(n)$  of comparisons to locate "data" in A, which contain  $n$  elements. Observe that in each comparison the size of the search area is reduced by half. Hence in the worst case, at most  $\log_2 n$  comparisons required. So  $f(n) = O(\lceil \log_2 n \rceil + 1)$ .

### Pros and Cons of Binary Search

The following are the pros and cons of a binary search:

Pros	Cons
<ol style="list-style-type: none"> <li>1. Suitable for sorted data</li> <li>2. Efficient for large lists</li> <li>3. Suitable for storage structures that support direct access to data</li> <li>4. Time complexity is <math>O(\log_2(n))</math></li> </ol>	<ol style="list-style-type: none"> <li>1. Not applicable for unsorted data</li> <li>2. Not suitable for sequential storage structures.</li> <li>3. Inefficient for small lists</li> </ol>

### Sorting

*Sorting* can be defined as the process of converting an unordered set of elements to an ordered set.

The following are different sorting techniques:

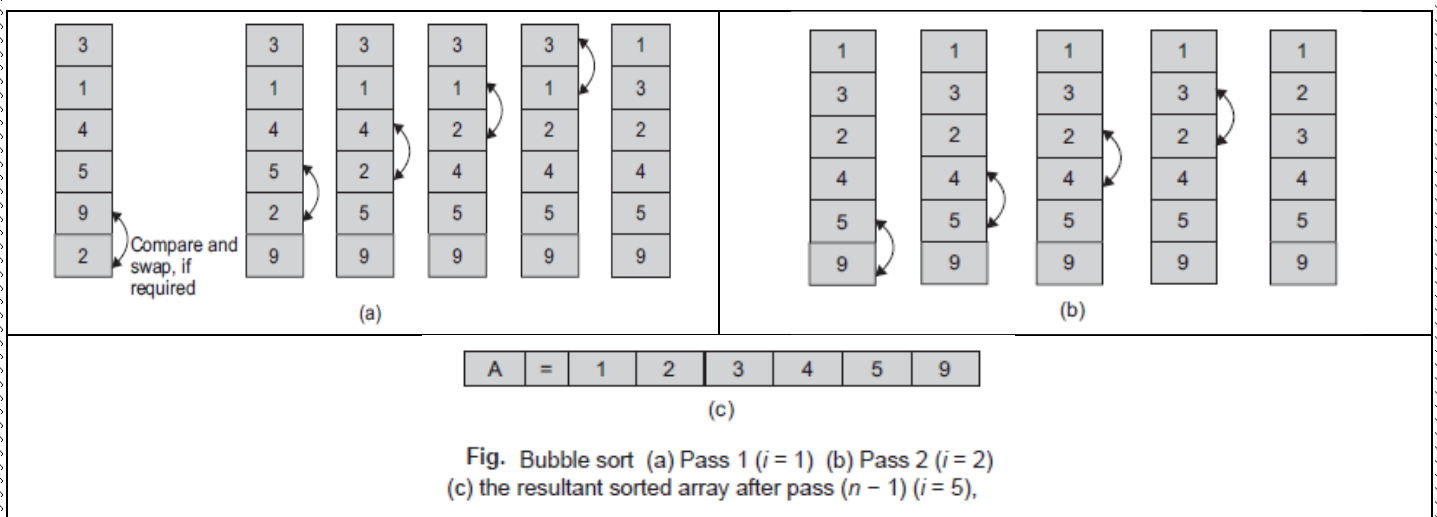
1. Bubble sort
2. Insertion sort
3. Selection sort
4. Quick sort
5. Merge sort

### Bubble Sort

The bubble sort is the oldest and the simplest sort. Unfortunately, it is also the slowest.

- ❖ The bubble sort works by comparing each item in the list with the item next to it and swapping them if required.
- ❖ The algorithm repeats this process until it makes a pass all the way through the list without swapping any items.
- ❖ This causes larger values to 'bubble' to the end of the list while smaller values 'sink' towards the beginning of the list.

The following figure illustrates the bubble technique by showing numbers and their moves during each pass.



The following algorithm illustrates Bubble Sort:

```

1. Let A be the array to be sorted
2. for i = 1 to n - 1
   for j = 0 to n - i
   begin
   if A[j] > A[j+1] then
   Swap A[j] with A[j + 1] as follows
   temp = A[j]
   A[j] = A[j + 1]
   A[j + 1] = temp
   end

```

The following Program Code illustrates the bubble sort function.

```

void bubblesort(int A[max], int n)
{
  int i, j, temp;
  for(i = 1; i < n; i++)
  {
    for(j = 0; j < n - i; j++)
    {
      if( A[j] > A[j + 1] )
      {

```



end  
3. stop

```
temp = A[j]; // swap A[j] with A[j + 1]
A[j] = A[j + 1];
A[j + 1] = temp;
} }
}}
```

### Analysis of Bubble Sort

The bubble sort makes  $(n - 1)$  comparisons in the first iteration,  $(n - 2)$  comparisons in the second iteration, ..., one comparison in the last iteration. This totals up to

$(n - 1) + (n - 2) + (n - 3) + \dots + 1 = n(n - 1)/2$ . Thus, the total number of comparisons is  $n(n - 1)/2$ , which is  $O(n^2)$ .

Hence, the time complexity for each of the cases is given by the following:

1. Average case complexity =  $O(n^2)$
2. Best case complexity =  $O(n^2)$
3. Worst case complexity =  $O(n^2)$

### Insertion Sort

Insertion sort is a simple sorting algorithm. It is twice as fast as the bubble sort and somewhat faster than the selection sort. It involves the following steps:

**Step 1:** The second element of the array will be compared with the first element. If the second element is smaller than first element, then, the second element is inserted in the position of first element. After first step, first two elements of the array will be in a sorted order.

**Step 2:** The third element of the array will be compared with the element that appears before it (first and second elements). If they are not in proper order, then the elements will be inserted at proper positions. After second step, first three elements of the array will be in a sorted order.

**Step 3:** Similarly, the fourth element of the array will be compared with the elements that appears before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of the array will be in a sorted order.

For example, consider the following array:

12   3   1   5   8

12   3   1   5   8

3   12   1   5   8

1   3   12   5   8

1   3   5   12   8

1   3   5   8   12

Sorted Array

The following algorithm illustrates the steps in Insertion Sort:

1. Set  $J = 2$ , where  $J$  is an integer
2. Check if  $\text{list}(J) < \text{list}(J - 1)$ : if so interchange them; set  $J = J - 1$  and repeat step (2) until  $J = 1$
3. Set  $J = 3, 4, 5, \dots, N$  and keep on executing step (2)

The following program code implements the InsertionSort() function.

```
void InsertionSort(int A[], int n)
{
    int i, j, element;
    for(i = 1; i < n; i++)
    {
        element = A[i];
        j = i;
        while((j > 0) && (A[j - 1] > element))
        {
            A[j] = A[j - 1];
            j = j - 1;
        }
        A[j] = element;
    }
}
```

### Analysis of Insertion Sort

Although the insertion sort is almost always better than the bubble sort, the time required in both the methods is approximately the same, that is, it is proportional to  $n^2$ , where  $n$  is the number of data items in the array. The total number of comparisons is given as follows:

$(n - 1) + (n - 2) + \dots + 1 = (n - 1) \cdot n/2$  which is  $O(n^2)$ .

### Selection Sort

The name Selection Sort comes from the idea of selecting the smallest element from those unsorted elements. The smallest element is then swapped with the first unsorted element. For Example: The basic process of sorting an 'n'- element array, A:

1. Find the smallest element from A[0]...A[n]
2. Swap that smallest element with A[0]
3. Find the smallest element from A[1]...A[n]
4. Swap that smallest element with A[1]
5. Find the smallest element from A[2]...A[n]
6. Swap that smallest element with A[2]

---

---

Continue this process until the last element in the array.

### Selection Sort Example

Here we are sorting an array containing the following numbers: 8, 27, 33, 2, 20, 12, 19, 5

In the following elements:

- The already sorted part is shown in *italics*
- The first unsorted element is shown underlined
- The minimum element of the unsorted part is

shown in **bold**

8, 27, 33, **2**, 20, 12, 19, 5

2, 27, 33, 8, 20, 12, 19, **5**

2, 5, 33, **8**, 20, 12, 19, 27

2, 5, 8, 33, 20, **12**, 19, 27

2, 5, 8, 12, 20, 33, **19**, 27

2, 5, 8, 12, 19, 33, **20**, 27

2, 5, 8, 12, 19, 20, 33, **27**

2, 5, 8, 12, 19, 20, 27, **33**

2, 5, 8, 12, 19, 20, 27, 33 → Resulting sorted array

The following program code implements Selection Sort function:

```
void SelectionSort(int A[], int n)
```

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

### Analysis of Selection Sort

In Program Code 9.9, we can note that there are two loops, one nested within the other. During the first pass,  $(n - 1)$  comparisons are made. In the second pass,  $(n - 2)$  comparisons are made. In general, for the  $i$ th pass,  $(n - i)$  comparisons are required. The total number of comparisons is as follows:

$$(n - 1) + (n - 2) + \dots + 1 = n(n - 1)/2$$

Therefore, the number of comparisons for the selection sort is proportional to  $n^2$ , which means that it is  $O(n^2)$ .

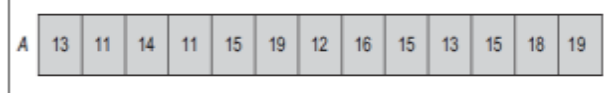
### Quick Sort

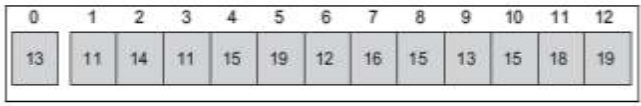
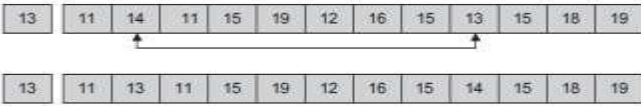
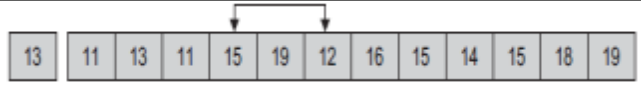

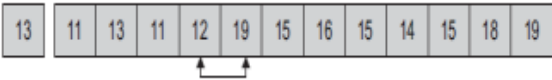



The quicksort algorithm uses the idea of divide and conquer. It is based on partitioning of array. A large array is partitioned into two arrays based on a pivot. The **left array** holds values **smaller** than the pivot and the **right array** holds values **greater** than pivot value. It finds the element called Pivot, which divides an array into two halves in such a way that elements in the left half are smaller than the pivot and elements in the right half are greater than pivot. The quicksort algorithm uses recursion. It calls three steps recursively:

1. Find the Pivot that divides the array into two halves.
2. Quicksort the left half
3. Quicksort the right half.

Example: Let the list of numbers to be sorted be {13, 11, 14, 11, 15, 19, 12, 16, 15, 13, 15, 18, 19}.

Now, the first element 13 becomes pivot. We need to place 13 at a proper location so that all elements to its left are smaller and the right are greater.



Initially, the array is pivoted about its first element $A[\text{pivot}] = 13$ .	
Let us first find the elements larger than the pivot, 13. In addition, let us find the last element not larger than the pivot. These elements are in positions 2 and 9. Let us swap those.	
Let us again start scanning from both the directions	
The elements 12 and 15 are to be swapped to get the following sequence	
Let us repeat the steps to get the following sequence:	
Here, the lower and upper bounds have crossed. So let us now swap the pivot-with element 12.	
Here, we get two partitions as represented in the following sequence:	
Recursively applying similar steps to each sublist on the right and left side of the pivot, we get,	

This is the final sorted array.

The following algorithm illustrates Quick Sort:

1. Select pivot =  $A[\text{Low}]$ , pivot location  $P = \text{low}$
2.  $i = \text{low}$  and  $j = \text{high}$ ;
3. Increment index  $i$  till  $A[i] \geq \text{pivot}$
4. Decrement index  $j$  till  $A[j] \leq \text{pivot}$
5. Swap  $A[i]$  with  $A[j]$
6. Repeat steps 4, 5, 6 till  $i < j$
7. if  $i < j$   
Swap  $a[P]$  with  $a[j]$
8. call Quicksort( $\text{low}, j - 1$ )
9. call Quicksort( $j + 1, \text{high}$ )
10. Stop

The following program implements the Quick sort:

```

#define max 20
void read(int A[max], int n)
{
    int i;
    for(i = 0; i < n; i++)
        cin >> A[i];
}
void display(int A[max], int n)
{
    int i;
    for(i = 0; i < n; i++)
        cout << A[i];
}
void swap(int *x, int *y)
{
    int temp;
    temp = *x;
    *x = *y;
    *y = temp;
}
void qsort(int A[], int low, int high)
{
    int partition(int A[],int low, int high)
    {
        int pivot, i, j;
        pivot = A[low];
        j = high + 1; i = low;
        do
        {
            i++;
            while(A[i] < pivot && low <= high)
            do
            {
                j++;
            } while(pivot < A[j]);
            if(i < j)
                swap(A[i],A[j]);
        } while(i < j);
        A[low] = A[j];
        A[j] = pivot;
        return j;
    }
    main()

```



```

{
int k;
if(low < high)
{
K = partition(A, low, high);
qsort(A, low, j - 1);
qsort(A, j + 1, high);
}
}

```

```

{
int A[max], n;
int i, choice;
cout << "Enter number of Elements:";
cin >> n;
cout << "Enter numbers:";
read(A, n);
qsort(A, 0, n - 1);
cout << "Sorted array is:";
display(A, n);
}
***** Output *****
Enter number of Elements: 7
Enter numbers: 10 5 23 67 20 30 60
Sorted array is: 5 10 20 23 30 60 67

```

### Analysis of Quick Sort

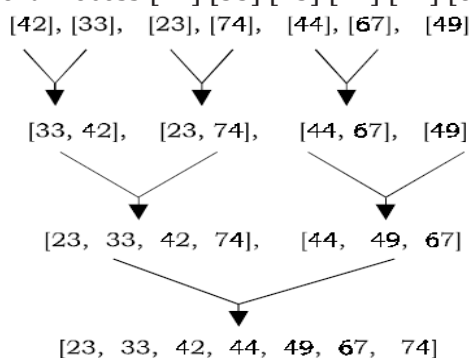
The average complexity of the quick sort algorithm is  $O(n \log n)$ . However, the worst case time complexity is  $O(n^2)$ .

### Merge Sort

Merge Sort is the most common external sorting algorithm. Merging is the process of combining two or more sorted files into the third sorted file. Divide and conquer strategy is used for merge sort. It has three steps to sort an input sequence  $S$  with  $n$  elements:

1. Divide—partition  $S$  into two sequences  $S1$  and  $S2$  of about  $n/2$  elements each
2. Recur—recursively sort  $S1$  and  $S2$
3. Conquer—merge  $S1$  and  $S2$  into a sorted sequence

Consider the following 7 elements to illustrate the Merge Sort Process: [42] [33] [23] [74] [44] [67] [49]



The following Mergesort() function implements this algorithm by calling it recursively :

```

List mergesort(list L, int n)
{
if(n == 1)
return(L);
else
{
split L into two halves L1 and L2;
return(merge(mergesort(L1, n/2),
(mergesort(L2, n/2))
}
}

```

### Time Complexity

Let  $T(n)$  be the running time of merge sort on an input list of size  $n$ . Then,  $T(n) < C1$  (if  $n = 1$ ), where  $C1$  is a constant and  $T(n) < 2T(n/2) + C2n$ . Here,  $2T(n/2)$  is for two recursive calls, and  $C2n$  is the cost of merging the two sorted lists.

Now, by the substitution method,

$$T(n) = 2T(n/2) + C2n$$

If  $n = 2^k$  for some  $k$ , it can be shown that after  $k$  steps

$$T(n) = 2^k T(n/2^k) + C2^2 k$$

Hence, for  $n = 2^k$

$$T(n) = n \log_2 n$$

That is,  $T(n) = O(n \log n)$

The following code implements the Merge Sort:

```

void merge (int A[],int low, int high, int mid)
{
int i, j, k, C[max];
i = low;
j = mid + 1;
k = 0;
while((i <= mid) && (j <= high))
{

```

```

for(i = low, j = 0; i <= high; i++, j++)
{
A[i] = C[j];
}
}
void MergeSort(int A[], int low, int high)
{
int mid;

```

```

if(A[i] < A[j])
C[k] = A[i++];
else
C[k] = A[j++];
k++;
}
while(i <= mid)
C[k++] = A[i++];
while(j <= high)
C[k++] = A[j++];

```

```

if(low < high)
{
mid = (low + high)/2;
MergeSort(A, low, mid);
MergeSort(A, mid + 1, high);
merge(A, low, high, mid);
}
}

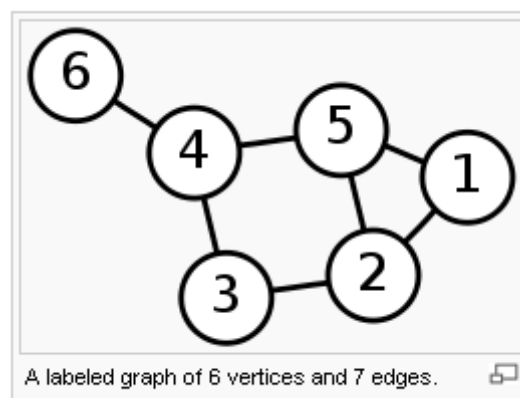
```

### Comparison of Sorting Techniques

Sorting method	Technique in brief	Best case	Worst case	Memory requirement	Is stable?	Pros	Cons
Bubble sort	Repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if required	$O(n^2)$	$O(n^2)$	No extra space needed	Yes	1. A simple and easy method 2. Efficient for small lists	Highly inefficient for large data
Selection sort	Finds the minimum value in the list and then swaps it with the value in the first position, repeats these steps	$O(n^2)$	$O(n^2)$	No extra space needed	No	Recommended for small files	Inefficient for large lists
Insertion sort	Every repetition of insertion sort removes an element from the input data, inserts it into the correct position in the already sorted list until no input elements remain.	$O(n)$	$O(n^2)$	No extra space needed	Yes	Relatively simple and easy to implement	Inefficient for large lists
Quick sort	Picks an element, called a pivot, from the list. Reorders the list so that all elements with values less than the pivot. Recursively sorts the sub-list of the lesser elements and the sub-list of the greater elements.	$O(n \log 2n)$	$O(n^2)$	No extra space needed	No	1. Extremely fast 2. Inherently recursive	Very complex algorithm
Merge sort	The algorithm divides the unsorted list into two sub-lists of about half the size. Then, it sorts each sub-list recursively by reapplying the merge sort and then merges the two sub-lists back into one sorted list.	$O(n \log 2n)$	$O(n \log 2n)$ $O(n \log 2n)$	Extra space proportional to $n$ is needed	Yes	Good for external file sorting	1. It requires twice the memory of the heap sort

## 5.22 GRAPH

Graph data structure consists mainly of a finite (and possibly mutable) [set](#) of ordered pairs, called **edges** or **arcs**, of certain entities called **nodes** or **vertices**. As in mathematics, an edge  $(x,y)$  is said to **point** or **go from**  $x$  to  $y$ .



The basic operations provided by a graph data structure  $G$  usually include

- **adjacent( $G, x, y$ ):** tests whether there is an edge from node  $x$  to node  $y$ .
- **neighbors( $G, x$ ):** lists all nodes  $y$  such that there is an edge from  $x$  to  $y$ .
- **add( $G, x, y$ ):** adds to  $G$  the edge from  $x$  to  $y$ , if it is not there.
- **delete( $G, x, y$ ):** removes the edge from  $x$  to  $y$ , if it is there.
- **get\_node\_value( $G, x$ ):** returns the value associated with the node  $x$ .
- **set\_node\_value( $G, x, a$ ):** sets the value associated with the node  $x$  to  $a$ .

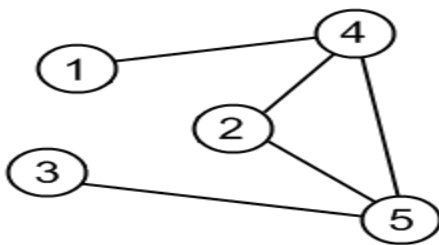
### 5.23 graphs representation

There are several possible ways to represent a graph inside the computer. Two of them are: **adjacency matrix** and **adjacency list**.

#### Adjacency matrix

Each cell  $a_{ij}$  of an adjacency matrix contains **0**, if there is an edge between  $i$ -th and  $j$ -th vertices, and **1** otherwise.

example.



Graph

	1	2	3	4	5
1	0	0	0	1	0
2	0	0	0	1	1
3	0	0	0	0	1
4	1	1	0	0	1
5	0	1	1	1	0

Adjacency matrix

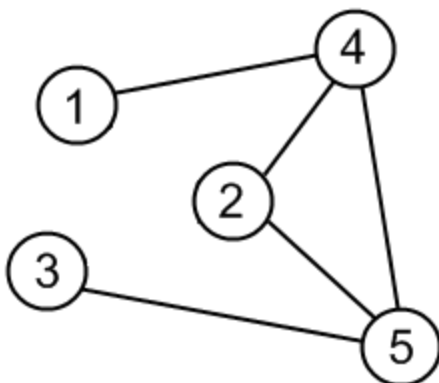
**Advantage:** Adjacency matrix is very convenient to work.

**Disadvantage:** Adjacency matrix consumes huge amount of memory for storing big graphs.

#### Adjacency list

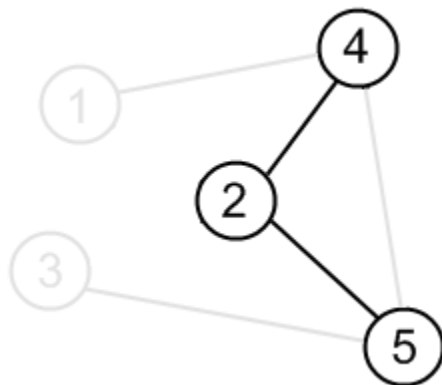
This kind of the graph representation is one of the alternatives to adjacency matrix. It requires less amount of memory. For every vertex adjacency list stores a list of vertices, which are adjacent to current one.

example.



1	4
2	4 5
3	5
4	1 2 5
5	2 3 4

Graph



Vertices, adjacent to {2}

Adjacency list

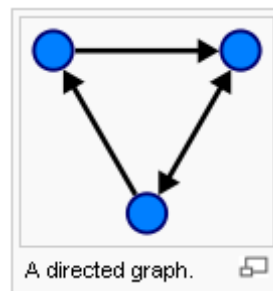
1	4
2	4 5
3	5
4	1 2 5
5	2 3 4

Row in the adjacency list

## 5.24 Types of graphs.

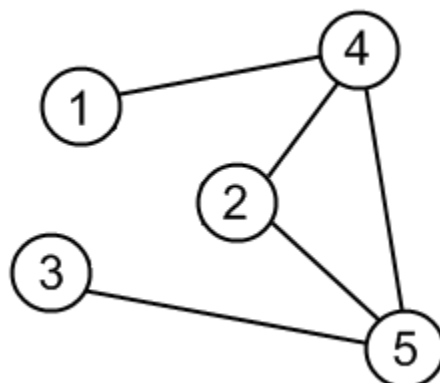
### Directed Graph

A graph is a directed graph if the edges have a direction, i.e. if they are arrows with a head and a tail.



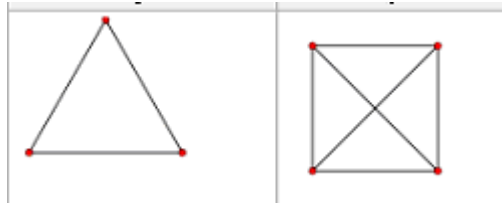
### Undirected Graph

A graph is undirected if the edges are "two way" (have no direction).



### Connected Graph

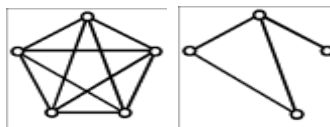
A graph in which each point (node) is connected to every other point



### Sub graph

A *graph* whose *vertices* and *edges* are *subsets* of another *graph*.

A subgraph of a graph is some smaller portion of that graph. Here is an example of a subgraph:



A graph

A subgraph

### Complete graph :

a graph is said to be complete graph if it has  $n$  vertices and  $n(n-1)$  edges

$K_1:0$	$K_2:1$	$K_3:3$	$K_4:6$

## 5.25 applications of graph

- Graphs are used in many applications few of them are listed below:
- Graphs are used analyze electric networks
- Graphs are used represent chemical bond structures
- Graphs are used to represent air lines
- Graphs are used to represent computer networks

## 5.26 GRAPH SEARCHES OR GRAPH TRAVERSALS

A graph search (or traversal) technique visits every node exactly one in a systematic fashion.

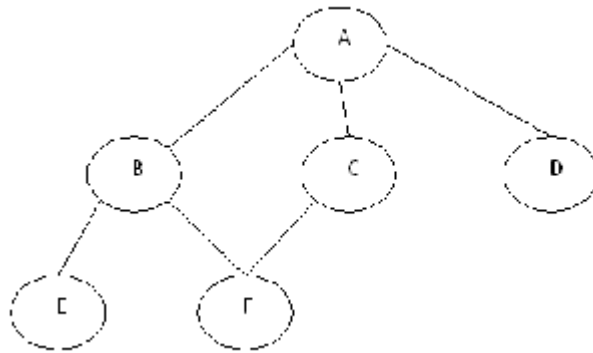
Two standard graph search techniques have been widely used:

- **Depth-First Search (DFS)**
- **Breadth-First Search (BFS)**

### Depth-First Search

- DFS follows the following rules:

- Initialize all nodes to ready state.
- Start with a node and push the node A in the stack.
- POP the top node X from the stack. Print X.
- Push all its neighbors omitting the processed ones in to the stack.
- Repeat the above two steps till all the elements of graph are visited.
- Pop all the elements of the stack and print them to complete DFS.

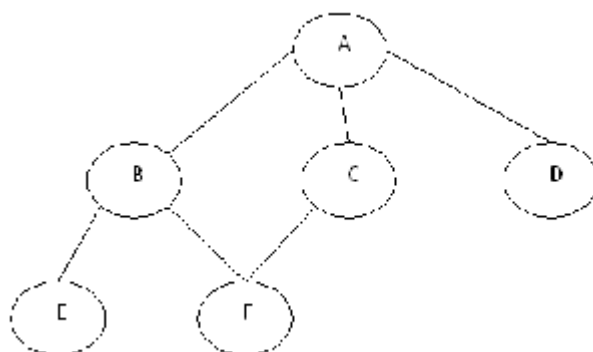


output : “A B E F C D”.

### Breadth-First Search

- BFS follows the following rules:
  - Initialize all nodes to ready state.
  - Start with a node and place it in the Queue.
  - Remove the element X from queue and print X.
  - Place all its neighbors omitting the processed ones in the queue
  - Repeat the above two steps till the elements of graph are visited.
  - Delete all the elements of the queue and print them to complete BFS

Example:



output “A B C D E F”.

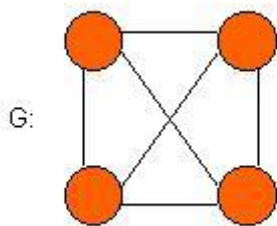
### 5.27 MINIMUM SPANNING TREE

A **spanning tree** is a subgraph of G, is a tree, and contains all the vertices of G. A **minimum spanning tree** is a spanning tree, but has weights or lengths associated with the

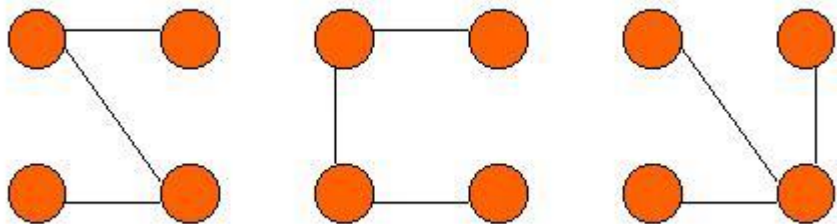
edges, and the total weight of the tree (the sum of the weights of its edges) is at a minimum.

Here are some examples:

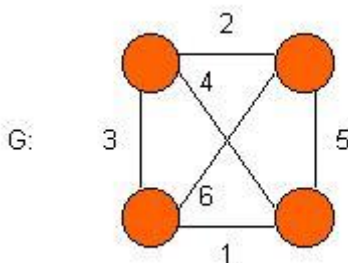
A graph G:



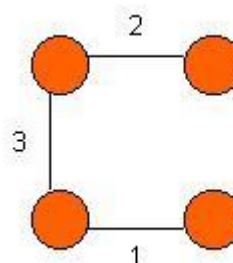
Three (of the many possible) spanning trees from graph G:



A weighted graph G:



The minimum spanning tree from weighted graph G:



there are two algorithms for finding minimum spanning tree

### **Kruskal's Algorithm:**

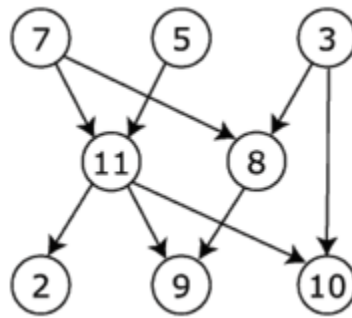
Kruskal's algorithm works as follows: Take a graph with 'n' vertices, keep adding the shortest (least cost) edge, while avoiding the creation of cycles, until  $(n - 1)$  edges have been added. (NOTE: Sometimes two or more edges may have the same cost. The order in which the edges are chosen, in this case, does not matter. Different MSTs may result, but they will all have the same total cost, which will always be the minimum cost)

### **Prim's Algorithm:**

This algorithm builds the MST one vertex at a time. It starts at any vertex in a graph (vertex A, for example), and finds the least cost vertex (vertex B, for example) connected to the start vertex. Now, from either 'A' or 'B', it will find the next least costly vertex connection, without creating a cycle (vertex C, for example). Now, from either 'A', 'B', or 'C', it will find the next least costly vertex connection, without creating a cycle, and so on it goes. Eventually, all the vertices will be connected, without any cycles, and an MST will be the result. (NOTE: Two or more edges may have the same cost, so when there is a choice by two or more vertices that is exactly the same, then one will be chosen, and an MST will still result)

## **5.27 TOPOLOGICAL ORDERING**

In [graph theory](#), a **topological sort** or **topological ordering** of a [directed acyclic graph](#) (DAG) is a linear ordering of its nodes in which each node comes before all nodes to which it has outbound edges. Every DAG has one or more topological sorts.



the valid topological sorts include the following

- 7, 5, 3, 11, 8, 2, 9, 10 (visual left-to-right)
- 3, 5, 7, 8, 11, 2, 9, 10 (smallest-numbered available vertex first)

## 5.28 Difference between tree and graph

A tree is a specialized case of a graph. A tree is a connected graph with no cycles and no self loops.

## 5.29 Difference between tree and binary tree

A tree, in turn, is a directed acyclic graph with the condition that every node is accessible from a single root. This means that every node has a "parent" node and 0 or more "child" nodes, except for the root node which has no parent.

A binary tree is a tree with one more restriction: no node may have more than 2 children