DATA STRUCTURE

CONTENTS

DATA STRUCTURE

Chapter-1

INTRODUCTION

Data is the basic entity or fact that is used in calculation or manipulation process. There are two types of data such as numerical and alphanumerical data. Integer and floating-point numbers are of numerical data type and strings are of alphanumeric data type. Data may be single or a set of values and it is to be organized in a particular fashion. This organization or structuring of data will have profound impact on the efficiency of the program.

DATA:

Data is value or set of values which does not give any meaning. It is generally a raw fact.

For example:

1. 34

2. 13/05/2008

3.Chintan

4. 12,34,43,21

ENTITY

An entity is a thing or object in the real world that is distinguishable from all other objects.

- The entity has a set of properties or attributes and the values of some sets of these attributes may uniquely identify an entity.
- An entity set is a collection of entities.

Example:

Entity	Student			
Attributes	Roll No.	Name	DOB	% of marks scored
Values	123	Ram	01/01/1980	79%

All students of a particular class constitute an entity set.

INFORMATION

It can be defined as meaningful data or processed data. When the raw facts are processed, we get a related piece of information as its output/result.

Example:

Data (01/01/1980) becomes information if entity Ram is related to Date of birth attribute (01/01/1980) as follows:

DOB of student Ram is 01/01/1980

DATA TYPE

A data type is a term which refers to the kind of data that may appear in computation. Every programming language has its own set of built-in data types.

Example:

Data	**Data type**
34	Numeric
Chintech	String
21,43,56	Array of integers
12/05/2008	Date

In C, the following are the basic data types are: int, long, char, float, double, void etc.

Definition:

Data Structure is a specialized format for storing data so that the data's can be organized in an efficient way.

Or

"The data structure is the logical or mathematical model of particular organization of data.".

Or

A data structure is a mathematical or logical way of organizing data in the memory that consider not only the items stored but also the relationship to each other and also it is characterized by accessing functions.

Data Structure = Organized data + Allowed Operations.

OBJECTIVES OF DATA STRUCTURE

- To identify and create useful mathematical entities and operations to determine what classes of problems can be solved by using these entities and operations.
- To determine the representation of these abstract entities and to implement the abstract operations on these concrete representation.

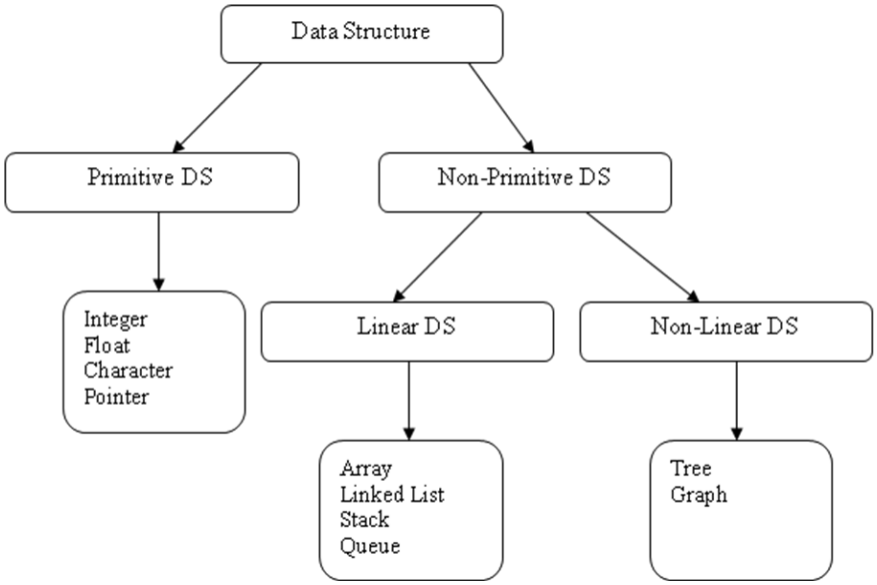
NEED OF A DATA STRUCTURE

- To understand the relationship of one data elements with the other and organize it within the memory.
- A data structures helps to analyze the data, store it and organize it in a logical or mathematical manner.
- Data structures allow us to achieve an important goal: component reuse.

CLASSIFICATION OF DATA STRUCTURE

Based on how the data items are operated, it will classify into two broad categories.

- Primitive Data Structure
- Non-Primitive Data Structure



Primitive Data Structure: These are basic DS and the data items are operated closest to the machine level instruction.
Example: integer, characters, strings, pointers and double.

Non-Primitive Data Structure: These are more sophisticated DS and are derived from primitive DS. Here data items are not operated closest to machine level instruction. It emphasize on structuring of a group of homogeneous (same type) or heterogeneous (different type) data items.

Linear Data Structure: In which the data items are stored in sequence order.
Example: Arrays, Linked Lists, Stacks and Queues

Non Linear Data Structure: In Non-Linear data structures, the data items are not in sequence.
Example: Trees, Graphs.

DATA STRUCTURE OPERATIONS:
The various operations that can be performed on different data structures are described below:

1. **Creating** – A data structure created from data.
2. **Traversal** – Processing each element in the list
3. **Search** – Finding the location of given element.

4. **Insertion** – Adding a new element to the list.
5. **Deletion** – Removing an element from the list.
6. **Sorting** – Arranging the records either in ascending or descending order.
7. **Merging** – Combining two lists into a single list.
8. **Modifying** – the values of DS can be modified by replacing old values with new ones.
9. **Copying** – records of one file can be copied to another file.
10. **Concatenating** – Records of a file are appended at the end of another file.
11. **Splitting** – Records of big file can be splitting into smaller files.

ABSTRACT DATA TYPES

Abstract data type (ADT) is a specification of a set of data and the set of operations that can be performed on the data.

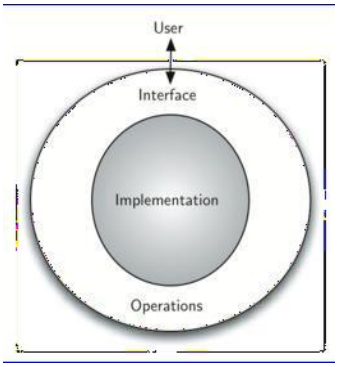
- It is a logical description of how we view the data and the operations that are allowed without knowing how they will be implemented.
- This means that we are concerned only with what the data is representing and not with how it will eventually be constructed.

Fig shows a picture of what an abstract data type is and how it operates.

- The user interacts with the interface by using the operations.
- The user is not concerned with the details of the implementation.
- The implementation of an abstract data type, often referred to as a data structure
- An ADT is a data declaration packaged together with the operations that are meaningful on the data type.

1. Declaration of Data
2. Declaration of Operations

Examples: Objects such as lists, sets and graphs with their operations can be called as ADT. For the SET ADT, we might have operations such as *union, intersection, and complement* etc.



Uses of ADT:

1. It helps to efficiently develop well designed program
2. Facilitates the decomposition of the complex task of developing a software system into a number of simpler subtasks
3. Helps to reduce the number of things the programmers has to keep in mind at any time.
4. Breaking down a complex task into a number of earlier subtasks also simplifies testing and debugging.

ALGORITHM:

An *algorithm* is a method of representing the step by step logical procedure for solving a problem. It is a tool for finding the logic of a problem.
In addition every algorithm must satisfy the following criteria:

- **Input:** There are zero or more quantities which are externally supplied. i.e. each algorithm must take zero, one or more quantities as input data.
- **Output:** At least one quantity is produced. i.e. produce one or more output values.
- **Definiteness:** Each instruction or step of an algorithm must be clear and unambiguous.
- **Finiteness:** An algorithm must terminate in a finite number of steps.
- **Effectiveness:** Each step must be effective, in the sense that it should be primitive (easily convertible to program).

ALGORITHM COMPLEXITY:

- After designing an algorithm, we have to be checking its correctness. This is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct.
- There may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on its complexity.
- Complexity refers to the rate at which the required storage or consumed time grows as a function of the problem size.
- It is the performance evaluation or analysis / measurement of an algorithm is obtained by totaling the number of occurrences of each operation when running the algorithm.
- Two important ways to characterize the effectiveness of an algorithm are its space complexity and time complexity.
- Space Complexity
- Time Complexity

SPACE COMPLEXITY:

- Space complexity of an algorithm or program is the amount of memory it needs to run to completion.
- We can measure the space by finding out that how much memory will be consumed by the instructions and by the variables used.

Example:

Suppose we want to add two integer numbers. To solve this problem we have following twoalgorithms:

Algorithm 1:
Step 1- Input A.

Step 2- Input B.

Step 3- Set C: = A+ B.

Algorithm 2:
Step 1- Input A.

Step 2- Input B.

Step 3- Write: 'Sum is ', A+B.

Step 4- Write: 'Sum is ', C.

Step 4- Exit.Step 5- Exit.

Both algorithms will produce the same result. But first takes 6 bytes and second takes 4 bytes (2 bytes for each integer). And first has more instructions than the second one. So we will choose the second one as it takes less space than the first one. (Space Complexity).

Suppose 1 second is required to execute one instruction. So the first algorithm will take 4 seconds and the second algorithm will take 3 seconds for execution. So we will choose the second one as it will take less time. (Time Complexity).

Time Complexity

- ➢ The time complexity of a program / algorithm is the amount of computer time that is need to run to completion.
- ➢ But the calculation of exact amount of time required by the algorithm is very difficult. So we can estimate the time and to estimate the time we use some asymptotic notation, which are described below. Let the no. of steps of an algorithm is measured by the function f(n).

1. Big 'O' notation

   The given function f(n) can be expressed by big 'O' notation as follows. f(n) = O (g(n)) if and only if there exist the +ve const. 'C' and no such that f(n) ≤ C * g(n) for all n ≥ no.

2. Omega (Ω) notation
   The function f(n) = Ω(g(n)) if and only if there exist the +ve const C and no such that f(n) ≥ C * g(n) for all n ≥ no

3. Theta (θ) notation:
   The function f(n) = θ(g(n)) if and only if there exist 3 +ve const 'C1' , 'C2' and no such that C1 * g(n)≤ f(n) ≤ C2 * g(n)

   Big 'O' is the upper bound Ω is the lower bound θ is the avg bound which can be estimated in time complexity.

Space Complexity

The space complexity is the program that the amount of memory that is needed to run to completion. The space complexity need by a program have two different Components.

1. Fixed space requirement: - The components refer to space requirement that do not depend upon the number and size of the programs input and output.

2. Variable space requirement: - This component consist of the space needed by structure variable whose size depends on the particular instruction I , of the program begin solved.

TIME-SPACE TRADE-OFF:

- There may be more than one approach (or algorithm) to solve a problem.
- The best algorithm (or program) to solve a given problem is one that requires less space in memory and takes less time to complete its execution.
- But in practice, it is not always possible to achieve both of these objectives.
- One algorithm may require more space but less time to complete its execution while the other algorithm requires less time space but takes more time to complete its execution.
- Thus, we may have to sacrifice one at the cost of the other.
- If the space is our constraint, then we have to choose a program that requires less space at the cost of more execution time.
- On the other hand, if time is our constraint such as in real time system, we have to choose a program that takes less time to complete its execution at the cost of more space.

Chapter-2
STRING PROCESSING

A string in C is a array of character.

- It is a one dimensional array type of char.
- Every string is terminated by null character('\0') .
- The predefined functions gets() and puts() in C language to read and display string respectively.

Declaration of strings:

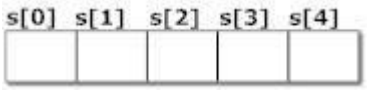
Strings are declared in C in similar manner as arrays. Only difference is that, strings are ofchar type.

Syntax:

char str_name[str_size];

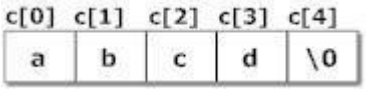
Example:

char s[5];



Initialization of strings

char c[]="abcd";



String handling functions

- Many library function are defined under header file <string.h> to perform differenttasks.
- Diffrent user defined functios are:
  - **Strlen()**
  - **Strcpy()**
  - **Strcmp()**
  - **Strcat()**

Strlen():

- The strlen( ) function is used to calculate the length of the string.
- It means that it counts the total number of characters present in the string which includes alphabets, numbers, and all special characters including blank spaces.

Example:

```
char str[] = "Learn C Online";
int strLength;
strLength = strlen(str);        //strLength contains the length of the string i.e. 14
```

Strcpy()

- strcpy function copies a string from a source location to a destination location and provides a null character to terminate the string.

Syntax:

strcpy(Destination_String,Source_String);

Example:

```
char *Destination_String;
char *Source_String = "Learn C Online";
```

```
strcpy(Destination_String,Source_String);
printf("%s", Destination_String);
```

Output:

Learn C Online

Strcmp()

- Strcmp() in C programming language is used to compare two strings.
- If both the strings are equal then it gives the result as zero but if not then it gives the numeric difference between the first non matching characters in the strings.

Syntax:

```
int strcmp(string1, string2);
```

Example:

```
char *string1 = "Learn C Online";
char *string2 = "Learn C Online";
int ret;
ret=strcmp(string1,     string2);
printf("%d",ret);
```

Output:

0

Strcat()

- The strcat() function is used for string concatenation in C programming language. Itmeans it joins the two strings together

Syntax:

```
strncat(Destination_String, Source_String,no_of_characters);
```

Example:

```
char *Destination_String="Visit ";
char *Source_String = "Learn C Online is a great site";
strncat(Destination_String, Source_String,14);
puts( Destination_String);
```

Output:

Visit Learn C Online

Program to check whether a word is palindrome or not

```
#include<stdio.h>
#include<string.h>
void main()
{
        char s1[10], s2[10];
        int x;
        gets(s1);
        strcpy(s2,s1);
        strrev(s1);
        x=strcmp(s1,s2);
        if(x==0)
            printf("pallindrome");
        else
        printf("not pallindrome");
        getch();
        }
```

Chapter-3

ARRAYS

An array is a finite, ordered and collection of homogeneous data elements.

finite – It contain limited no. of element.

ordered – All the elements are stored one by one in a contiguous memory location.

homogeneous -All the elements of an array of same type.

The elements of an array are accessed by means of an index or subscript.

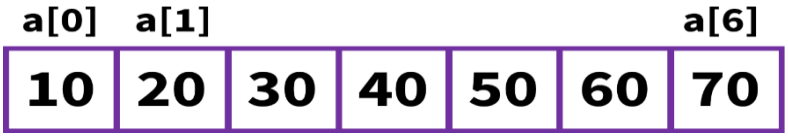
That's why array is called subscripted variable.

LINEAR ARRAY

If one subscript is required to refer all the elements in an array then this is called Linear array or one-dimensional array.

Representation of Linear Array in memory

Let a is the name of an integer array.
It contains a sequence of memory location.



Let b = address of the 1st element in the array i.e. base address If w = word size

then address of any element in array can be obtained as address of

(a[i]) = b + i x w i = index no. address of

a[3] = b + i x w

= b + 3 x 2 = b + 6

Operation on Array

1. TRAVERSAL

This operation is used to visit all the elements of the array.

```
Void traverse (int a[], int n)
{
int i;
for(i=0; i<n; i++)
Printf("%d",a[i]);
}
```

2. INSERTION

Inserting the array at the end position can be done easily, but to insert at the middle of the array we have to move the element to create a vacant position to insert the new element.

```
int insert (int a[], int n, int pos, int ele)
{
int i;
for(i=n-1; i>=pos-1; i--)
{
a[i+1] = a[i];
}
a[pos-1] = ele;
n++;
return(n);
}
```

3. DELETION

Deleting an element at the end of the array can be done easily by only decreasing the array size by 1.

But deleting an element at the middle of the array require that each subsequent element from the position where to be deleted should be moved to fill up the array.

```
int delete (int a[], int n, int pos)
{
```

```
for (i = pos-1; i<n-1; i++)
{
a[i] = a[i+1];
} n- - ;
return(0);
}
```

Memory Representation of 2D Array

- The array having two subscript is called as 2D array.
- In 2D array the elements are stored in contiguous memory location as in single dimensional array.
- There are 2 ways of storing any matrix in memory.
  1. Row-major order
  2. Column-major order

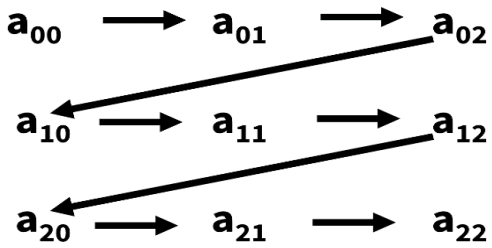
Row-major Order

In row-major order the row elements are focused first that means elements of matrix are stored on a row-by-row basis.

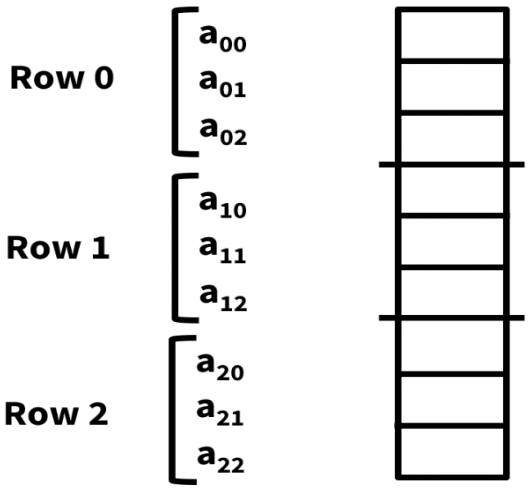
Logical Representation of array a[3][3]

$$\begin{bmatrix} a[0][0] & a[0][1] & a[0][2] \\ a[1][0] & a[1][1] & a[1][2] \\ a[2][0] & a[2][1] & a[2][2] \end{bmatrix}$$

Row major order representation of a[3][3]



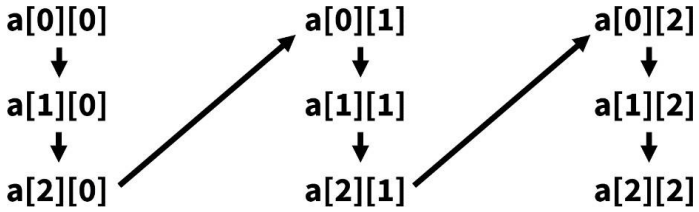
Storage Representation in Row-major order



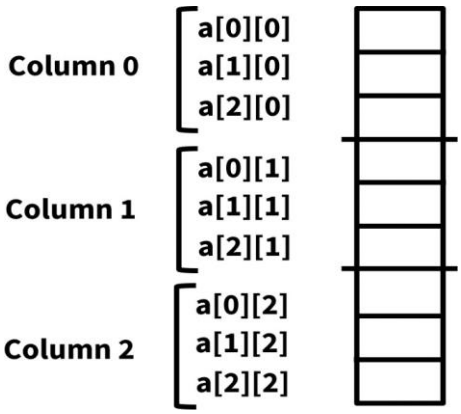
Column-major Order

In column major order the column elements are focused first that means elements of the matrix are stored in column-by-column basis.

Ex: Column major order representation of a[3][3]



Storage Representation of matrix in Column-major Order



Address Calculation of Matrix in Memory

Row-major order

Suppose a [u1] [u2] is a 2D array u1 = no. of row

u2 = no. of column

Address of a[i][j] = b+(i x u2 + j) x w

Column-major order

Address of a[i][j] = b + (j x u1 + i) x w

Pointers

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer.

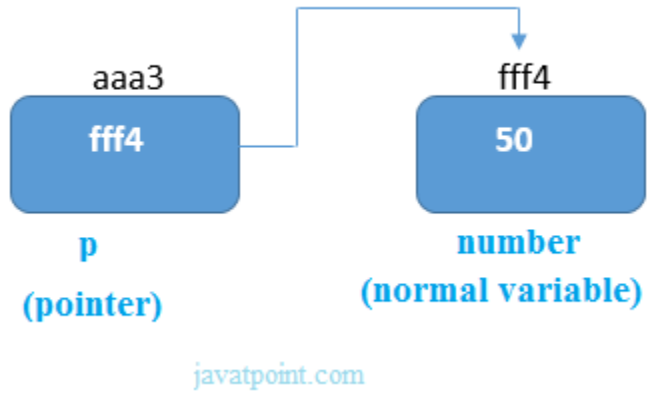
Syntax: datatype *var_name;

Example:

1. **int** *a;//pointer to int
2. **char** *c;//pointer to char

Pointer Example

An example of using pointers to print the address and value is given below.



In the above figure, pointer variable stores the address of number variable, i.e., fff4. The value of number variable is 50. But the address of pointer variable p is aaa3.

By the help of * (**indirection operator**), we can print the value of pointer variable p.

Let's see the pointer example as explained for the above figure.

```
#include<stdio.h>
int main(){
int number=50;
int *p;
p=&number;//stores the address of number variable
printf("Address of p variable is %x \n",p);
 printf("Value of p variable is %d \n",*p);
return 0;
}
```

Output

```
Address of number variable is fff4
Address of p variable is fff4
Value of p variable is 50
```

Sparse matrices

Matrix with relatively a high proportion of zero entries are called sparse matrix. In other words, the sparse matrix can be defined as the matrix that has a greater number of zero elements than the non-zero elements.

Benefits of using the sparse matrix

i)Storage - We know that a sparse matrix contains lesser non-zero elements than zero, so less memory can be used to store elements. It evaluates only the non-zero elements.

ii)Computing time: In the case of searching in sparse matrix, we need to traverse only the non-zero elements rather than traversing all the sparse matrix elements. It saves computing time by logically designing a data structure traversing non-zero elements.

Representation of sparse matrix

There are two ways to represent the sparse matrix that are listed as follows -

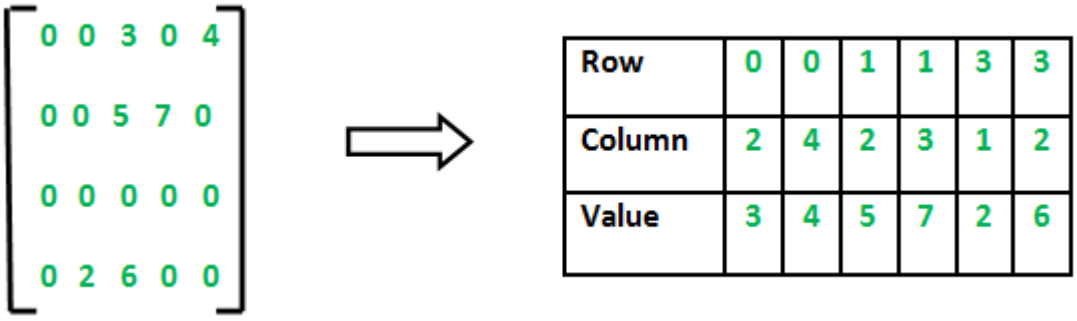
- Array representation
- Linked list representation

Array representation of the sparse matrix

Representing a sparse matrix by a 2D array leads to the wastage of lots of memory. This is because zeroes in the matrix are of no use, so storing zeroes with non-zero elements is wastage

of memory. To avoid such wastage, we can store only non-zero elements. If we store only non-zero elements, it reduces the traversal time and the storage space.
2D array is used to represent a sparse matrix in which there are three rows named as

- **Row:** Index of row, where non-zero element is located
- **Column:** Index of column, where non-zero element is located
- **Value:** Value of the non zero element located at index – (row,column)



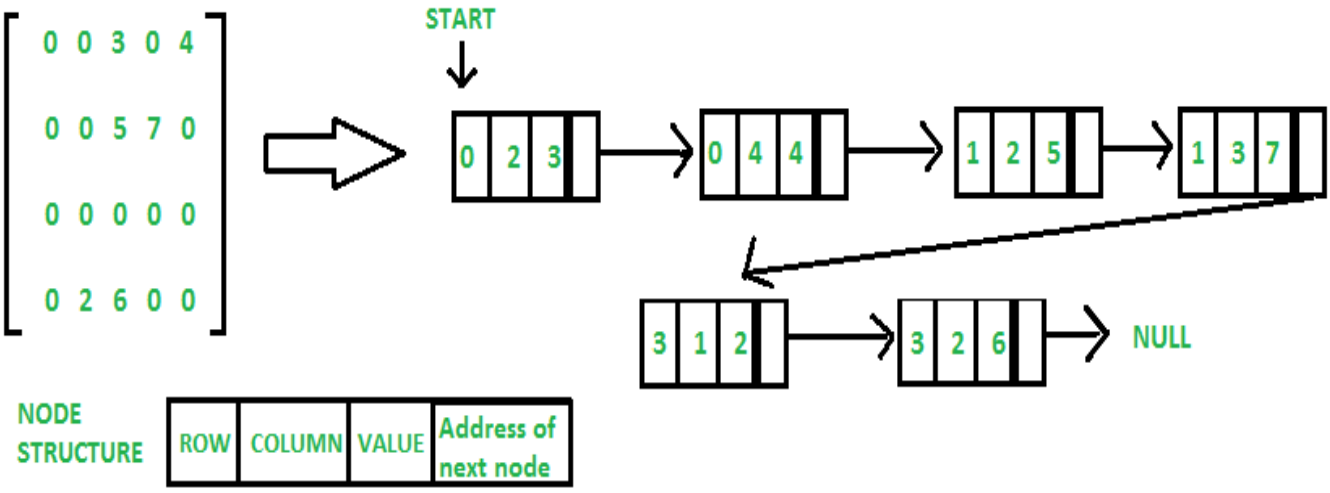
Row	0	0	1	1	3	3
Column	2	4	2	3	1	2
Value	3	4	5	7	2	6

Linked List representation of the sparse matrix

In a linked list representation, the linked list data structure is used to represent the sparse matrix. The advantage of using a linked list to represent the sparse matrix is that the complexity of inserting or deleting a node in a linked list is lesser than the array.

In linked list, each node has four fields. These four fields are defined as:

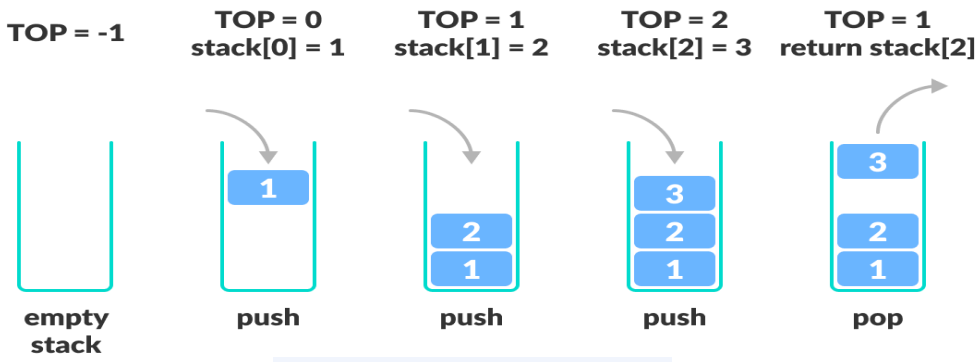
- Row: Index of row, where non-zero element is located
- Column: Index of column, where non-zero element is located
- Value: Value of the non zero element located at index – (row,column)
- Next node: Address of the next node


0 0 3 0 4
0 0 5 7 0
0 0 0 0 0
0 2 6 0 0
START
0 2 3
0 4 4
1 2 5
1 3 7
3 1 2
3 2 6
NULL
NODE STRUCTURE
ROW
COLUMN
VALUE
Address of next node

Chapter-4

STACKS & QUEUES

- Stack is a linear data structure in which an element may be inserted or deleted at one end called TOP of the stack.
- That means the elements are removed from a stack in the reverse order of that in which they were inserted into the stack.
- Stack is called LIFO (Last-in-first-Out) Str. i.e. the item just inserted is deleted first.
- There are 2 basic operations associated with stack
  1. Push :- This operation is used to insert an element into stack.
  2. Pop :- This operation is used to delete an element from stack



Working of Stack Data Structure

Condition also arise :

1. Overflow: - When a stack is full and we are attempting a push operation, overflow condition arises.
2. Underflow: - When a stack is empty, and we are attempting a pop operation then underflow condition arises.

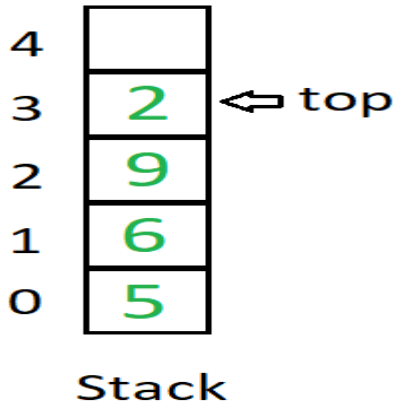
Representation of Stack in memory

A stack may be represented in the memory in two ways:

1. Using one dimensional array i.e. Array representation of Stack.
2. Using single linked list i.e. Linked list representation of stack.

Array Representation of Stack :

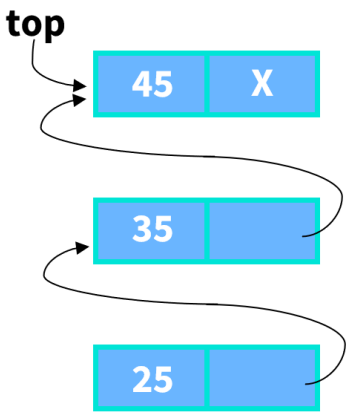
To implement a stack in memory, we need a pointer variable called TOP that hold the index of the top element of the stack, a linear array to hold the elements of the stack and a variable MAXSTK which contain the size of the stack.



MAXSTK=5

Linked List Representation of Stack :

- Array representation of Stack is very easy and convenient but it allows only to represent fixed sized stack.
- But in several application size of the stack may very during program execution, at that cases we represent a stack using linked list.
- Single linked list is sufficient to represent any Stack.



- Here the 'info' field for the item and 'next' field is used to print the next item.

INSERTION IN STACK (PUSH OPERATION)

This operation is used to insert an element in stack at the TOP of the stack.

Algorithm :-

PUSH (STACK, TOP, MAXSTK, ITEM)
This procedure pushes an item into stack.
1. If TOP = 0 the print Underflow and Return.
2. Set ITEM = STACK[TOP] (Assigns TOP element to ITEM)
3. Set TOP = TOP-1 (Decreases TOP by 1)
4. Return

(Where Stack = Stack is a list of linear structure
TOP= pointer variable which contain n the location of Top element of the stack
MAXSTK= variable which contain size of the stack

ITEM= Item to be inserted)

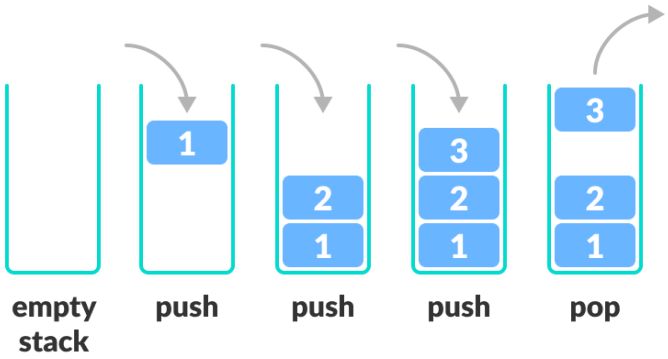
DELETION IN STACK (POP OPERATION)

This operation is used to delete an element from stack.

Algorithm :-

POP(STACK, TOP, ITEM)
This procedure deletes the top element of STACK and assigns it to the variable ITEM.

1. If TOP = 0 the print Underflow and Return.
2. Set ITEM = STACK[TOP] (Assigns TOP element to ITEM)
3. Set TOP = TOP-1 (Decreases TOP by 1)
4. Return

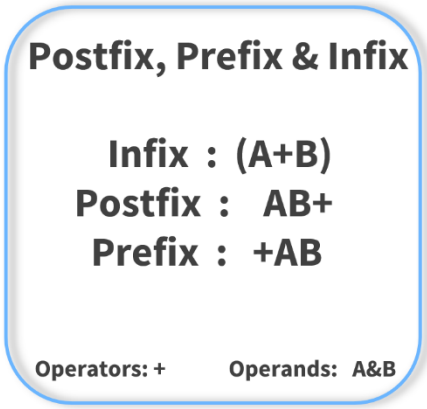


Stack Push and Pop Operations

ARITHMETIC EXPRESSION

There are 3 notation to represent an arithmetic expression.

1. Infix notation
2. Prefix notation
3. Postfix notation



1. **INFIX NOTATION**
   - The conventional way of writing an expression is called as infix.
     Ex: A+B, C+D, E*F, G/M etc.
   - Here the notation is
     <Operand><Operator><Operand>
   - This is called infix because the operators come in between the operands.
2. **PREFIX NOTATION**
   - This notation is also known as “POLISH NOTATION”
   - Here the notation is
     <Operator><Operand><Operand>
   - Ex: +AB, -CD, *EF, /GH

3. POSTFIX NOTATION

- This notation is called as postfix or suffix notation where operator is suffixed by operand.
- Here the notation is
  <Operand ><Operand><Operator>
- Ex: AB+, CD-, EF*, GH/
- This notation is also known as " REVERSE POLISH NOTATION."

CONVERSION FROM INFIX TO POSTFIX EXPRESSION

Algorithm:

POLISH(Q,P)

Q is an arithmetic expression written in infix notation. This algorithm finds the equivalent postfix expression P.

1. Push " (" onto stack and add ") " to the end of Q.
2. Scan Q from left to right and repeat steps 3 to 6 for each element of Q until the STACK is empty.
3. If an operand is encountered, add it top.
4. If a left parenthesis is encountered, push it onto stack.
5. If an operator X is encountered, then:
   a) Repeatedly POP from STACK and add to P each operator (on the top of the STACK) which has the same precedence as or higher precedence than the operator X .
   b) Add the operator X to STACK.
6. If a right parenthesis is encountered then:
   a) Repeatedly POP from STACK and add to P each operator (on the top of the STACK) until a left parenthesis is encountered.
   b) Remove the left parenthesis.

[End of if str.]

[End of Step-2 Loop]

7. Exit

Ex: A+(B*C – (D/E^F)*G)*H)

Symbol Scanned	STACK	EXPRESSION (POSTFIX)
	(	
A	(	A
+	(+	A
(	(+(	A
B	(+(	AB
*	(+(*	AB
C	(+(*	ABC
-	(+(-	ABC*
C	(+(-(	ABC*
D	(+(-(	ABC*D
/	(+(-(/	ABC*D
E	(+(-(/	ABC*DE
^	(+(-(/^	ABC*DE
F	(+(-(/^	ABC*DEF
)	(+(-	ABC*DEF^/
*	(+(-*	ABC*DEF^/
G	(+(-*	ABC*DEF^/G
)	(+	ABC*DEF^/G*-
*	(+*	ABC*DEF^/G*-
H	(+*	ABC*DEF^/G*-H
)		ABC*DEF^/G*-H*+

Equivalent Postfix Expression : ABC*DEF^/G*-H*+

EVALUATION OF POSTFIX EXPRESSION

Algorithm:

This algorithm finds the value of an arithmetic expression P written in Postfix notation.

1. Add a right parenthesis ") " at the end of P.
2. Scan P from left to right and repeat steps 3 and 4 for each element of P until the ") " is encountered.
3. If an operand is encountered, put it on STACK.
4. If an operator X is encountered then:
    a) Remove the two top element of STACK, where A is top element and B is the next-to-top element.

b) Evaluate B X A.
c) Place the result of (b) on STACK.

[End of if str.]

5. Set value equal to the top element on STACK.
6. Exit

Ex: 5, 6, 2, +, *, 12, 4, /, -)

Symbol Scanned	STACK
5	5
6	5,6
2	5,6,2
+	5,8
*****	40
12	40,12
4	40,12,4
/	40,3
-	37 (Result)
)	

APPLICATIONS OF STACK:

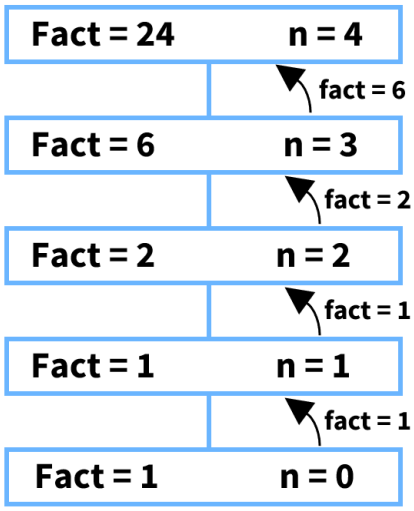
- Recursion process uses stack for its implementation.
- Quick sort uses stack for sorting the elements.
- Evaluation of antithetic expression can be sone by using STACK.
  - Conversion from infix to postfix expression
  - Evaluation of postfix expression
  - Conversion from infix to prefix expression
  - Evaluation of prefix expression.
- Backtracking
- Keeptrack of post-visited (history of a web- browsing)

IMPLEMENTATION OF RECURSION

A function is said to be Recursive function if it call itself or it call a function such that the function call back the original function.

This concept is called as Recursion.

The Recursive function has two properties.



I. The function should have a base condition.
II. The function should come closer towards the base condition.

The following is one of the example of recursive function which is described below.

Factorial of a no. using Recursion

The factorial of a no. 'n' is the product of positive integers from 1 to n.

n ! = 1 X 2 X 3 X . . . . . .X (n-) X n
Physically proved n ! = n X (n-1) !

The factorial function can be defined as follows.

I. If n=0 then n ! = 1
II. If n>0 then n ! = n.(n-1) !

The factorial algorithm is given below factorial (fact , n)

This procedure calculates n! and returns the value in the variable fact.

1. If n=0 then fact=1 and Return.
2. Call factorial (fact, n-1)
3. Set fact = fact*n
4. Return

Ex: Calculate the factorial of 4.

4 ! = 4 x 3 !
3 ! = 3 x 2 !
2 ! = 2 x 1 !
1 ! = 1 x 0 !
0 ! = 1
1 ! = 1 x 1 = 1
2 ! = 2 x 1 = 2
3 ! = 3 x 2 = 6

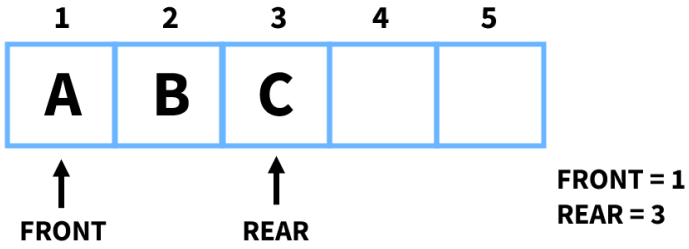
4 ! = 4 x 6 = 24

QUEUE

- Queue is a linear data structure or sequential data structure where insertion take place at one end and deletion take place at the other end.
- The insertion end of Queue is called rear end and deletion end is called front end.
- Queue is based on (FIFO) First in First Out Concept that means the node i.e. added first is processed first.
- Here Enqueue is Insert Operation.
- Dequeue is nothing but Delete Operation.

Representation Of Queue In Memory

- The Queue is represented by a Linear array "Q" and two pointer variable FRONT and REAR.
- FRONT gives the location of element to be deleted and REAR gives the location after which the element will be inserted.
- The deletion will be done by setting
  Front = Front + 1
- The insertion will be done by setting
  Rear = Rear + 1



INSERTION IN QUEUE(Enqueue)

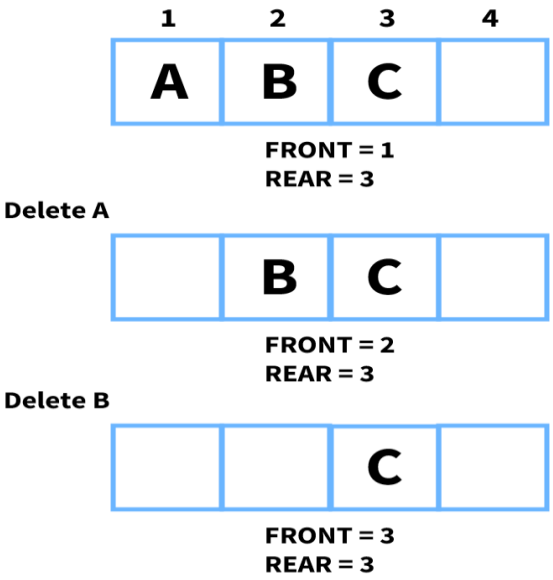
Algorithm:

Insert (Q, ITEM, Front, Rear)
This procedure insert ITEM in queue Q.

1. If Front = 1 and Rear = Max
   Print 'Overflow' and Exit.
2. If front = NULL than
   Front = 1
   Rear = 1
   else

Rear = Rear +1

3. Q [Rear] = ITEM
4. Exit



DELETION IN QUEUE(Dequeue)

Algorithm:

Delete (Q, ITEM, FRONT, REAR)

This procedure remove element from queue Q.

1. If Front = Rear = NULL

Print 'Underflow' and Exit.

2. ITEM = Q (FRONT)
3. If Front = Rear

Front = NULL

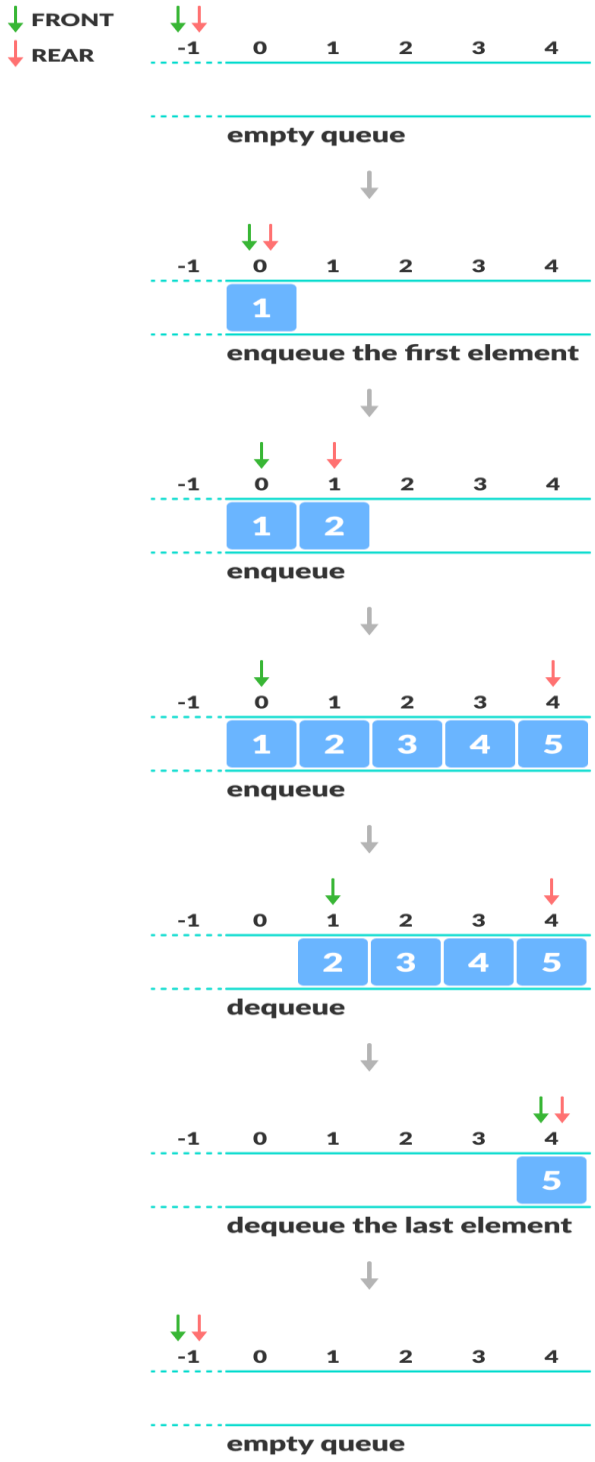
Rear = NULL

else

Front = Front + 1

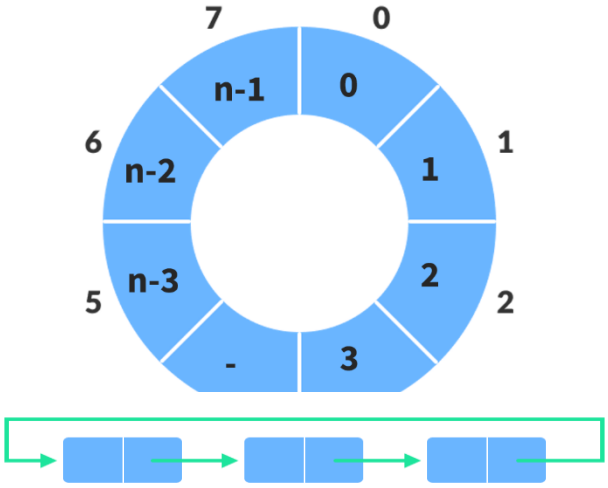
4. Exit

Enqueue and Dequeue Operations



CIRCULAR QUEUE

- Let we have a queue that contain 'n' elements in which Q[1] comes after Q[n].
- When this technic is used to construct a queue then the queue is called circular queue.
- In Circular queue when REAR is 'n' and any element is inserted then REAR will set as 1.
- Similarly when the front is n and any element is deleted then front will be 1.



Circular Queue representation

INSERTION ALGORITHM OF CIRCULAR QUEUE

Insert (Q, N, FRONT, REAR, ITEM)
This procedure inserts an element ITEM into the circular queue Q.

1. If Front = 1 and Rear = N

then print 'Overflow' and Exit.

2. If front = NULL

then Front = 1 and Rear = 1
else if Rear = N then
set Rear = 1
else
set Rear = Rear+1
(End of if Str.)

3. Set Q [Rear] = ITEM (Insert a new element)
4. Exit

DELETION ALGORITHM OF CIRCULAR QUEUE

Delete (Q, N, ITEM, FRONT, REAR)
This procedure delete an element from a circular queue.

1. If Front = NULL then write Underflow and Return.
2. Set ITEM = Q [FRONT]
3. If Front = Rear

Then Front = NULL and Rear = NULL
Else if Front = N then
Set Front = 1
else
Front = Front+1
[End of if str.]

4. Return

PRIORITY QUEUE

- If is a type of queue in which each element has been assigned a priority such that the order in which the elements are processed according to the elements are processed according to the following rule
  I. This element of high priority is processed first.
  II. The element having same priority is processed according to the order in which they are inserted.
- The lower ranked no. enjoy high priority.

Representation of Priority Queue in memory

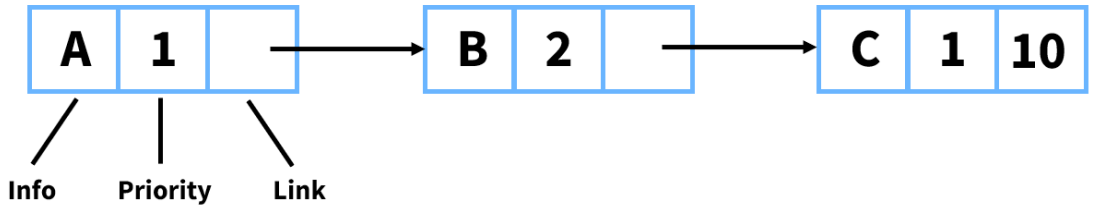
Priority Queue can be represented in memory in 2 ways.

I. One way list representation

II. Array representation

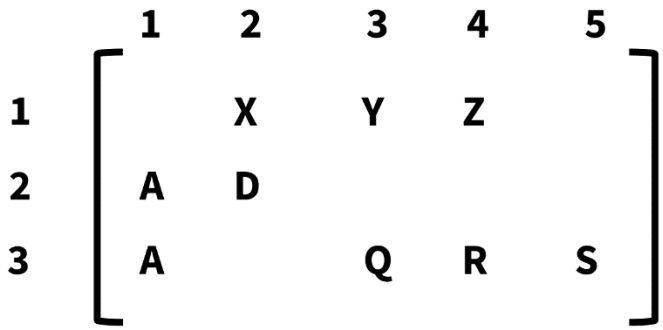
One way list representation

Each node in the list contain 3 fields.i.e. INFO, PRIORITY, LINK



Array Representation

Here the priority queue is represented by means of 2D array.
Where row value represents the priority number and each row maintained as a circular queue.
Each row has its FRONT and REAR value to represent the starting and ending of row.



FRONT	REAR
2	4
1	2
3	1

Types Of Priority Queue

1.Ascending Priority Queue :-
It is the collection of items into which items can be inserted arbitrarily and from which only the smallest item can be removed.
2.Descending Priority Queue :-
This is similar to ascending priority Queue but it allows deletion of only the largest item.

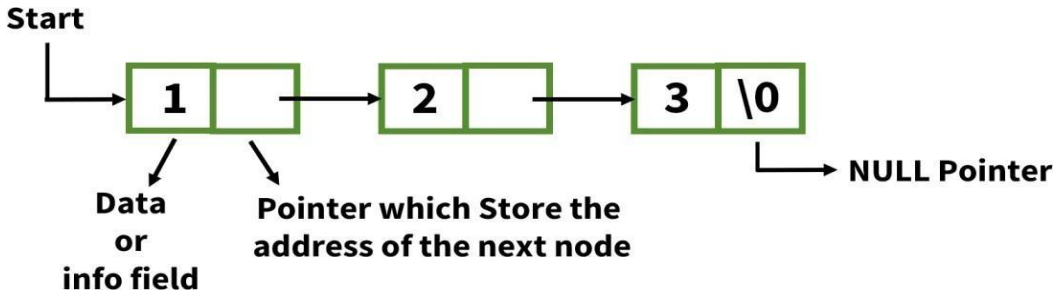
Chapter-5

LINKED LIST

Linked List is a collection of data elements called as nodes.
The node has 2 parts

- Info is the data part
- Next i.e. the address part that means it points to the next node.

If linked list adjacency between the elements are maintained by means of links or pointers.
A link or pointer actually the address of the next node.



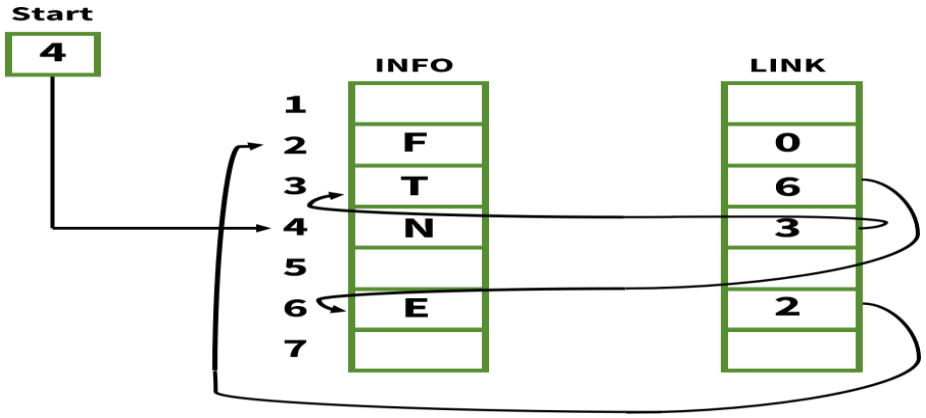
The last node of the list contain '\0' NULL which shows the end of the list.
The linked list contain another pointer variable 'Start' which contain the address the first node.
Linked List is of 4 types

- Single Linked List
- Double Linked List
- Circular Linked List
- Header Linked List

Representation of Linked List in Memory

- Linked List can be represented in memory by means 2 linear arrays i.e. Data or info and Link or address.
- They are designed such that info[K] contains the Kth element and Link[K] contains the next pointer field i.e. the address of the Kth element in the list.
- The end of the List is indicated by NULL pointer i.e. the pointer field of the last element will be NULLorZero

.



Start = 4
info [4] = N Link [4] = 3
info [3] = T Link [3] = 6
info [6] = E Link [6] = 2
info [2] = F Link [2] = 0 i.e. the null value
So the list has ended .

Representation of a node in a Linked List

```
Struct Link
{
int data ;
Struct Link * add ;
};
```

SINGLE LINKED LIST

A Single Linked List is also called as one-way list. It is a linear collection of data elements called nodes, where the linear order is given by means of pointers.
Each node is divided into 2 parts.

I. Information
II. Pointer to next node.

OPERATION ON SINGLE LINKED LIST

1.Traversal

Algorithm:
Display (Start) This algorithm traverse the list starting from the 1st node to the end of the list.
Step-1 : "Start" holds the address of the first node.
Step-2 : Set Ptr = Start [Initializes pointer Ptr]
Step-3 : Repeat Step 4 to 5, While Ptr ≠ NULL
Step-4 : Process info[Ptr] [apply Process to info(Ptr)]
Step-5 : Set Ptr = next[Ptr] [move Print to next node]
[End of loop]
Step-6 : Exit

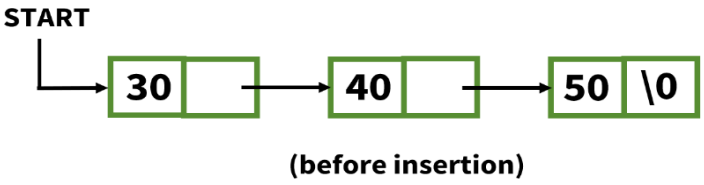
2.Insertion

The insertion operation is used to add an element in an existing Linked List. There is various positions where node can be inserted.

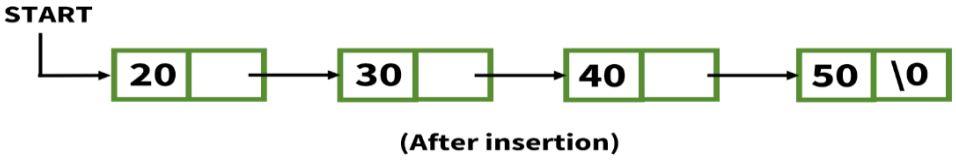
- Insert at the beginning
- Insert at end
- Insert at specific location.

Insert At The Beginning

Suppose the new mode whose information field contains 20 is inserted as the first node.



(before insertion)



(After insertion)

Algorithm:

This algorithm is used to insert a node at the beginning of the Linked List. Start holds the address of the first node.

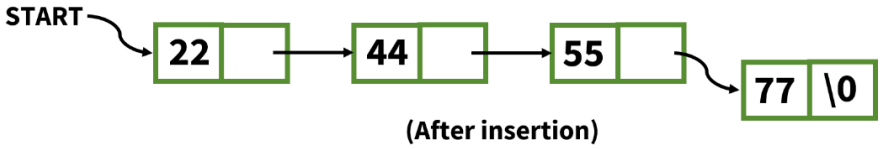
Step-1 : Create a new node named as P.
Step-2 : If P == NULL then print "Out of memory space" and exit.
Step-3 : Set info [P] = x (copies a new data into a new node)
Step-4 : Set next [P] = Start (new node now points to original first node)
Step-5 : Set Start = P
Step-6 : Exit

Insert At The End

To insert a node at the end of the list, we need to traverse the List and advance the pointer until the last node is reached.

Suppose the new node whose information field contains 77 is inserted at the last node.

(before insertion)

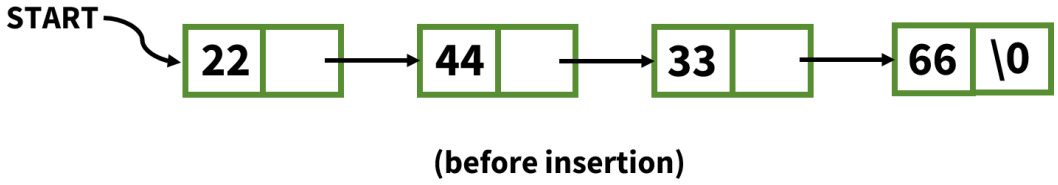


(After insertion)

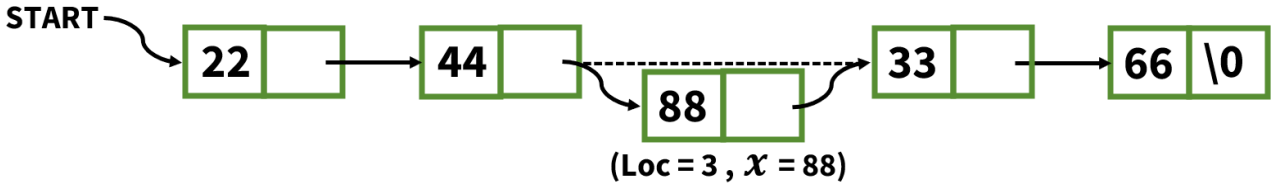
Algorithm:
This algorithm is used to insert a node at the end of the linked list. 'Start' holds the address of the first node.
Step-1 : Create a new node named as P.
Step-2 : If P = NULL then print "Out of memory space" and Exit.
Step-3 : Set info [P] = x (copies a new data into a new node)
Step-4 : Set next [P] = NULL
Step-5 : Set Ptr = Start
Step-6 : Repeat Step-7 while Ptr ≠ NULL
Step-7 : Set temp = Ptr
Ptr = next [Ptr] (End of step-6 loop)
Step-8 : Set next [temp] = P
Step-9 : Exit

Insert At Any Specific Location

To insert a new node at any specific location we scan the List from the beginning and move up to the desired node where we want to insert a new node.
In the below fig. Whose information field contain 88 is inserted at 3rd location.



(before insertion)



(Loc = 3 , x = 88)

(after insertion)

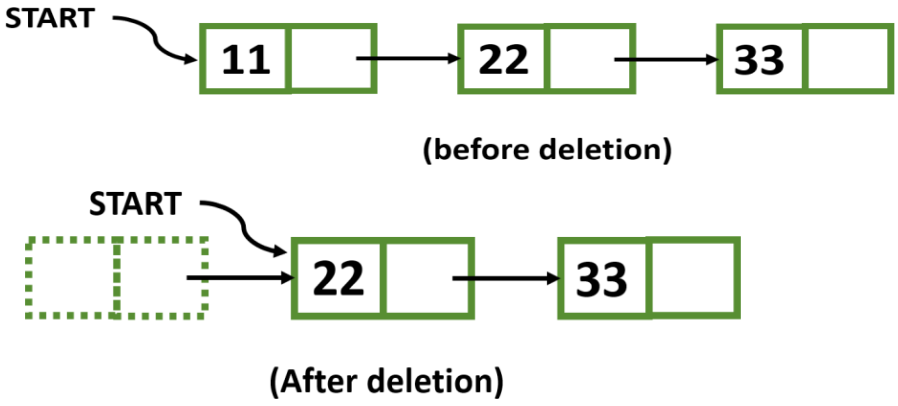
Algorithm:
'Start' holds the address of the 1st node.
Step-1 : Set Ptr = Start
Step-2 : Create a new node named as P.
Step-3 : If P = NULL then write 'Out of memory' and Exit.
Step-4 : Set info [P] = x (copies a new data into a new node)
Step-5 : Set next [P] = NULL

Step-6 : Read Loc Step-7 : Set i = 1
Step-8 : Repeat steps 9 to 11 while Ptr ≠ NULL and i < Loc
Step-9 : Set temp = Ptr.
Step-10 : Set Ptr = next [Ptr]
Step-11 : Set i = i + 1
[End of step-7 loop]
Step-12 : Set next [temp] = P
Step-13 : Set next [P] = Ptr
Step-14 : Exit

3. Deletion

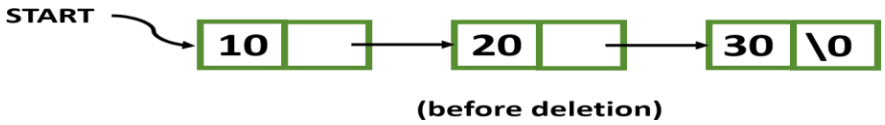
The deletion operation s used to delete an element from a single linked list. There are various position where node can be deleted.

Delete the 1st Node



Algorithm:
Start holds the address of the 1st node.
Step-1 : Set temp = Start
Step-2 : If Start = NULL then write 'UNDERFLOW' & Exit.
Step-3 : Set Start = next[Start]
Step-4 : Free the space associated with temp.
Step-5 : Exit

Delete the last node

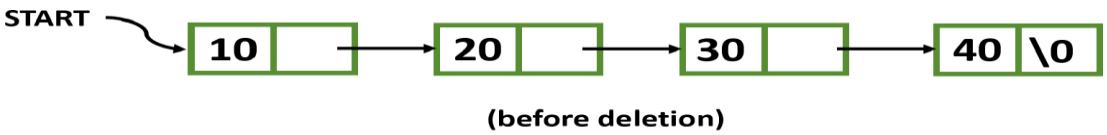


(After deletion)

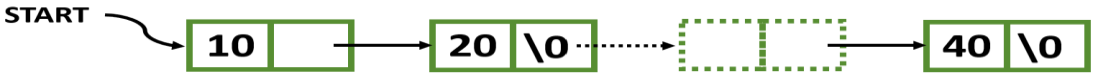
Algorithm:
Start holds the address of the 1st node.

Step-1 : Set Ptr = Start Step-2 : Set temp = Start
Step-3 : If Ptr = NULL then write 'UNDERFLOW' & Exit. Step-4 : Repeat Step-5 and 6 While next[Ptr] ≠ NULL Step-5 : Set temp = Ptr
Step-6 : Set Ptr = next[Ptr] (End of step 4 loop)
Step-7 : Set next[temp] = NULL
Step-8 : Free the space associated with Ptr.
Step-9 : Exit

Delete the node at any specific location



(before deletion)



(After deletion (Loc=3))

Algorithm:
Start holds the address of the 1st node.
Step-1 : Set Ptr = Start
Step-2 : Set temp = Start
Step-3 : If Ptr = NULL then write 'UNDERFLOW' and Exit.
Step-4 : Set i = 1
Step-5 : Read Loc
Step-6 : Repeat Step-7 to 9 while Ptr ≠ NULL and i < Loc
Step-7 : Set temp = Ptr
Step-8 : Set Ptr = next[Ptr]
Step-9 : Set i = i+1
(End of Step 6 loop)
Step-10 : Set next[temp] = next[Ptr]
Step-11 : Free the space associated with Ptr.
Step-12 : Exit

4. SEARCHING

Searching means finding an element from a given list.

Algorithm:
Start holds the address of the 1st node.
Step-1 : Set Ptr = Start
Step-2 : Set Loc = 1
Step-3 : Read element
Step-4 : Repeat Step-5 and 7 While Ptr ≠ NULL
Step-5 : If element = info[Ptr] then Write 'Element found at position', Loc and Exit.
Step-6 : Set Loc = Loc+1
Step-7 : Set Ptr = next[Ptr]
(End of step 4 loop)
Step-8 : Write 'Element not found'
Step-9 : Exit

Header linked list

Header Linked List is a modified version of Singly Linked List.
In Header linked list, we have a special node, the Header Node present at the beginning of the linked list.
The Header Node is an extra node at the front of the list storing meaningful information about the list.
It does not represent any items of the list like other nodes rather than the information present in the Header node is global for all nodes such as Count of Nodes in a List, Maximum among all Items, Minimum value among all Items etc.
This gives useful information about the Linked list.

This data part of this header node is generally used to hold any global information about the entire linked list. The next part of the header node points to the first node in the list.

A header linked list can be divided into two types:

i) Grounded header linked list that stores NULL in the last node's next field.

ii) Circular header linked list that stores the address of the header node in the next part of the last node of the list.

Garbage Collection

The operating system of a computer may periodically collect all the deleted space on to the free storage list. Any technique which does these collections is called garbage collection.

When we delete a particular note from an existing linked list or delete the linked list the space occupied by it must be given back to the free pool. So that the memory can be the used by some other program that needs memory space.

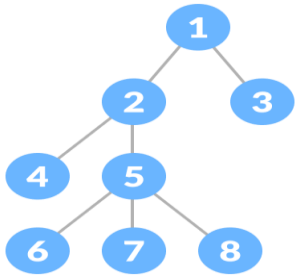
To the free pool is done.

The operating system will perform this operation whenever it finds the CPU idle or whenever the programs are falling short of memory space. The OS scans through the entire memory cell & marks those cells. That are being by some program then it collects the entire cell which are not being used & add to the free pool. So that this cells can be used by other programs. This process is called garbage collection. The garbage collection is invisible to the programmer.

Chapter-6

TREE

A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value.



Application:

The following are the applications of trees:

- **Storing naturally hierarchical data:** Trees are used to store the data in the hierarchical structure. For example, the file system.
- **Organize data:** It is used to organize data for efficient insertion, deletion and searching.
- **Trie:** It is a special kind of tree that is used to store the dictionary. It is a fast and efficient way for dynamic spell checking.
- **Heap:** It is also a tree data structure implemented using arrays. It is used to implement priority queues.
- **B-Tree and B+Tree:** B-Tree and B+Tree are the tree data structures used to implement indexing in databases.
- **Routing table:** The tree data structure is also used to store the data in routing tables in the routers.

Basic Tree Terminologies

1.Node

A node is an entity that contains a key or value .

The last nodes of each path are called leaf nodes or external nodes that do not contain a link/pointer to child nodes.

The node having at least a child node is called an internal node.

2. Root Node: The topmost node of a tree or the node which does not have any parent node is called the root node. {**1}** is the root node of the tree. A non-empty tree must contain exactly one root node .

3. Parent Node: The node which is a predecessor of a node is called the parent node of that node. {**2}** is the parent node of {**6, 7}**.

4.Child Node: The node which is the immediate successor of a node is called the child node of that node. Examples: {**6, 7}** are the child nodes of {**2}**.

5.Degree of a Node: The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be 0. The degree of a tree is the degree of its root. The degree of the node {3} is 3. Leaf Node or External Node: The nodes which do not have any child nodes are called leaf nodes. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} are the leaf nodes of the tree

6. Sibling: Children of the same parent node are called siblings. {8, 9, 10} are called siblings.

7.Depth of a node: The count of edges from the root to the node. Depth of node {14} is 3.

8.Height of a node: The number of edges on the longest path from that node to a leaf. Height of node {**3}** is **2**.

9.Height of a tree: The height of a tree is the height of the root node i.e the count of edges from the root to the deepest node. The height of the above tree is **3**.

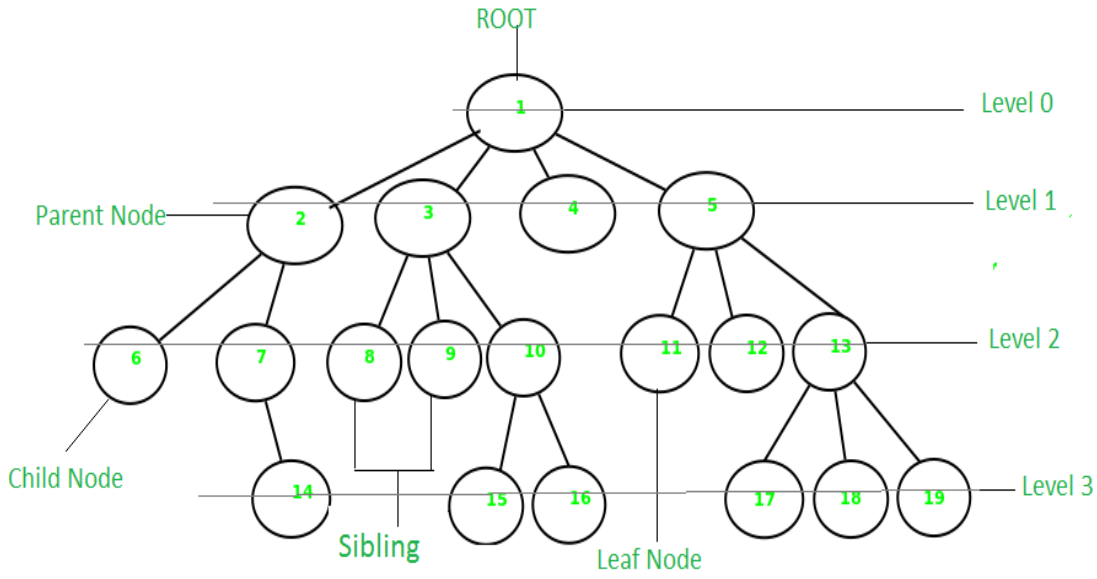
10.Level of a node: The count of edges on the path from the root node to that node. The root node has level **0**.

11.Internal node: A node with at least one child is called Internal Node.

12.Ancestor of a Node: Any predecessor nodes on the path of the root to that node are called Ancestors of that node. The node 1 is called ancestor of node7 as there is successive parent present from node 7 to node 1. {**1, 2}** are the parent nodes of the node {**7}**

13.Descendant: Any successor node on the path from the leaf node to that node. {**7, 14}** are the descendants of the node. {**2}**.

14. Path: A sequence of edges is called as path.



Binary Tree

Each node of a binary tree consists of three items:

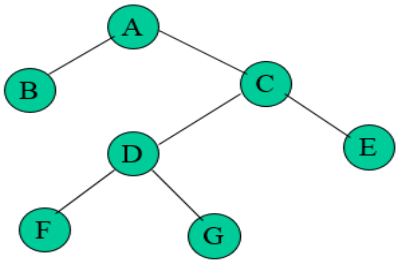
- data item
- address of left child
- address of right child

A binary tree is a tree-type non-linear data structure with a maximum of two children for each parent. Here, binary name itself suggests that 'two'; therefore, each node can have either 0, 1 or 2 children.

Types of Binary Tree

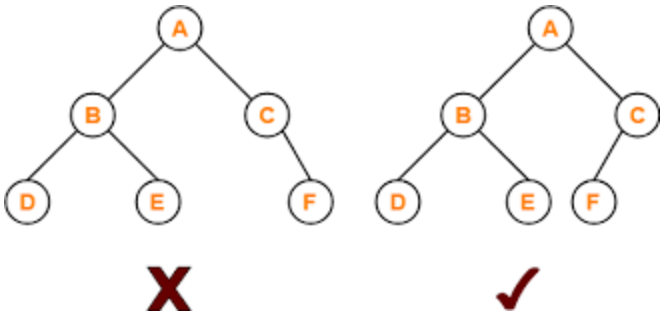
1. Full/ proper/ strict Binary tree

The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes.



2. Complete Binary tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.



If we number the nodes of the complete binary tree from left to right level by level then the node k has its left child at 2k position and right child at 2k+1 position.

Binary Tree Representation in memory

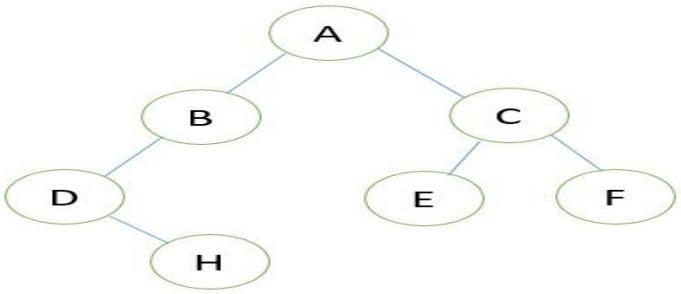
A tree must represent a hierarchical relationship between parent node and child node.
Let T be a Binary Tree. There are two ways of representing T in the memory as follow

1. Sequential or Linear Representation of Binary Tree.
2. Link Representation of Binary Tree.

1.Sequential or Linear Representation of Binary Tree.

Let us consider that we have a tree T. let our tree T is a binary tree that us complete binary tree. Then there is an efficient way of representing T in the memory called the sequential representation or array representation of T. This representation uses only a linear array TREE as follows:

1. The root N of T is stored in TREE [1].
2. If a node occupies TREE [k] then its left child is stored in TREE [2 * k] and its right child is stored into TREE [2 * k + 1].



Its sequential representation is as follow:

A	B	C	D	-	E	F	-	H		

2. Linked Representation of Binary Tree

Consider a Binary Tree T. T will be maintained in memory by means of a linked list representation which uses three parallel arrays; INFO, LEFT, and RIGHT pointer variable ROOT as follows. In Binary Tree each node N of T will correspond to a location k such that

1. LEFT [k] contains the location of the left child of node N.
2. INFO [k] contains the data at the node N.
3. RIGHT [k] contains the location of right child of node N.

Representation of a node:

LEFT [k]	INFO [k]	RIGHT [k]

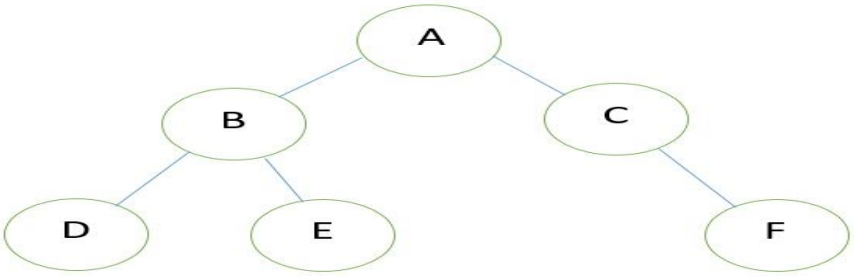
In this representation of binary tree root will contain the location of the root R of T. If any one of the

subtree is empty, then the corresponding pointer will contain the null value if the tree T itself is empty, the ROOT will contain the null value.

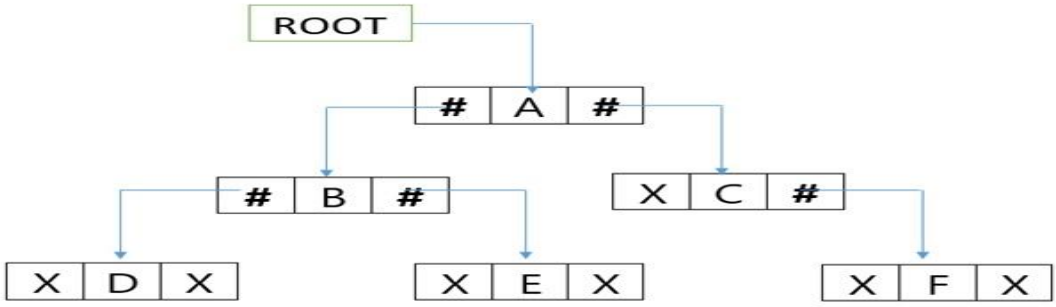
Example

Consider the binary tree T in the figure. A schematic diagram of the linked list representation of T appears in the following figure. Observe that each node is pictured with its three fields, and that the empty subtree is pictured by using x for null entries.

Binary Tree



Linked Representation of the Binary Tree:



Binary Tree Traversal

Traversal is a process to visit all the nodes of a tree and may print their values too. Because, all nodes are connected via edges (links) we always start from the root (head) node. That is, we cannot randomly access a node in a tree.

There are three ways which we use to traverse a tree −

- In-order Traversal
- Pre-order Traversal
- Post-order Traversal

1.In-order Traversal

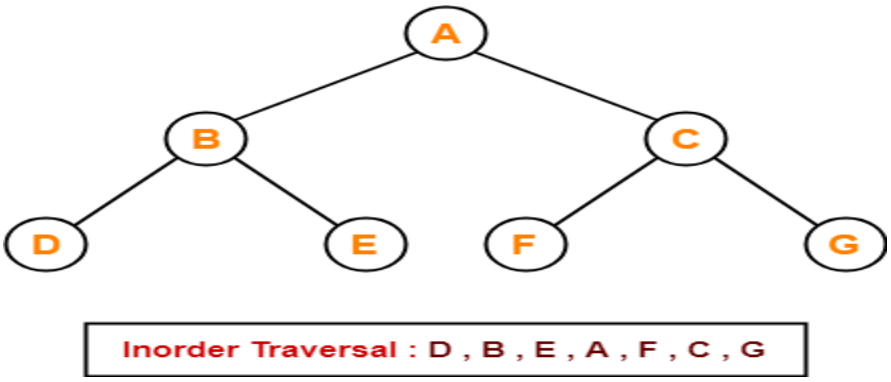
This is also known as symmetric order.

This traversal is referred as LNR.

Algorithm-

1. First, visit all the nodes in the left subtree
2. Then the root node

3. Visit all the nodes in the right subtree
   Left → Root → Right



Application-

Inorder traversal is used to get infix expression of an expression tree.

2.Preorder Traversal-

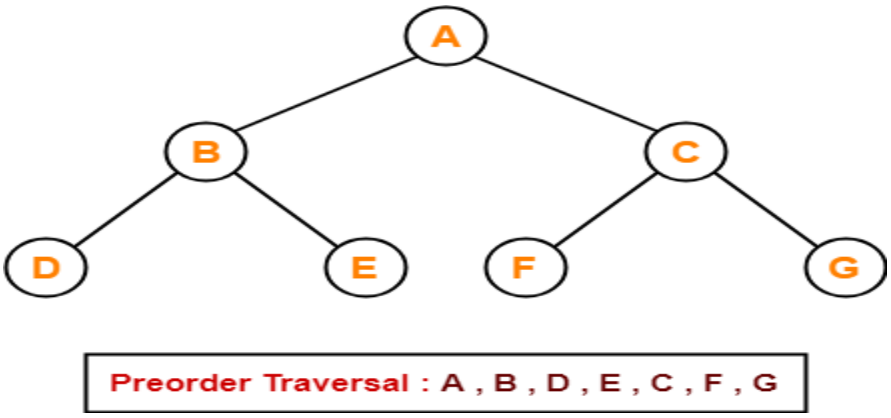
This is also known as depth fist order.

This traversal is referred as NLR.

Algorithm-

1. Visit the root
2. Traverse the left sub tree i.e. call Preorder (left sub tree)
3. Traverse the right sub tree i.e. call Preorder (right sub tree)

Root → Left → Right



Applications-

Preorder traversal is used to get prefix expression of an expression tree.Preorder traversal is used to create a copy of the tree.

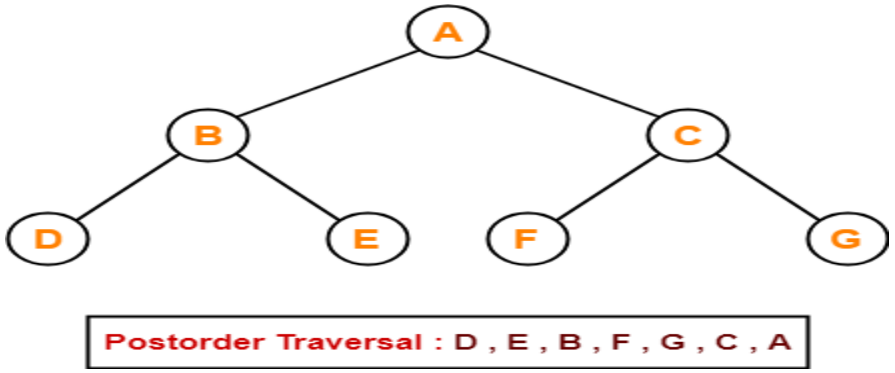
3.Postorder Traversal-

This traversal is referred as LRN.

Algorithm-

1. Traverse the left sub tree i.e. call Postorder (left sub tree)
2. Traverse the right sub tree i.e. call Postorder (right sub tree)
3. Visit the root

Left → Right → Root



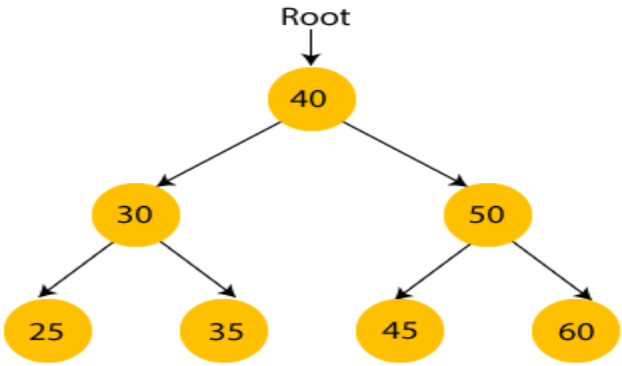
Applications-

- Postorder traversal is used to get postfix expression of an expression tree.
- Postorder traversal is used to delete the tree.
- This is because it deletes the children first and then it deletes the parent.

Binary Search tree

A binary tree T is termed as Binary Search Tree if each node n of T satisfies the following property.

- The value of n is greater than the value of all nodes in its left subtree.
- The value of n is less than the value of all nodes in its right subtree.



Advantages of Binary search tree

- Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Example of creating a binary search tree

Suppose the data elements are - 45, 15, 79, 90, 10, 55, 12, 20, 50

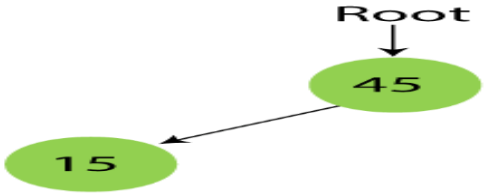
- First, we have to insert 45 into the tree as the root of the tree.
- Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
- Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.

Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below -

Step 1 - Insert 45.

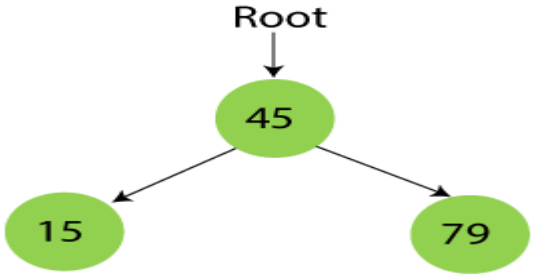
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree



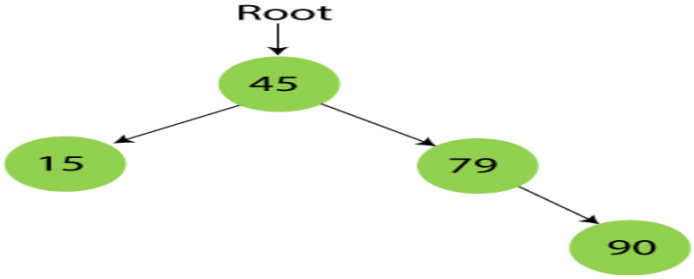
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



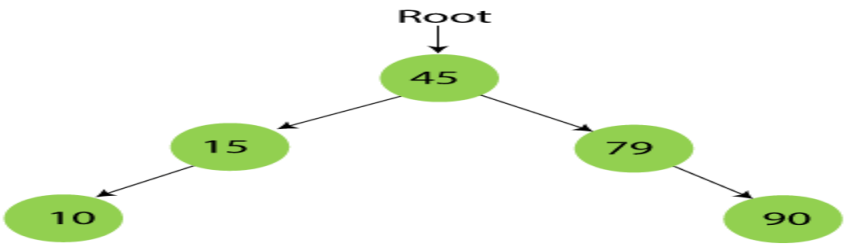
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79**.**



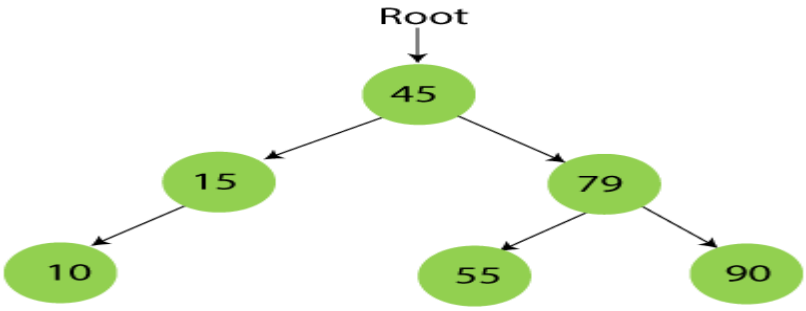
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



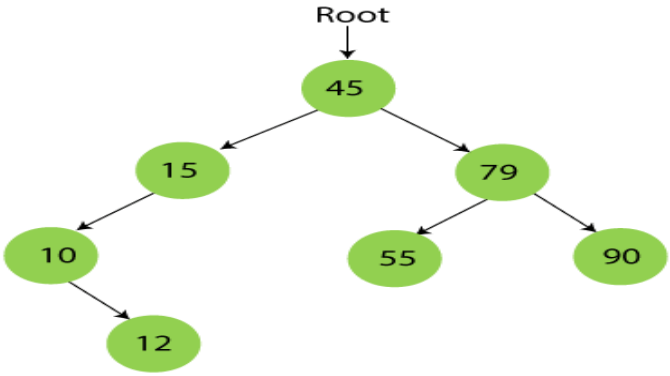
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



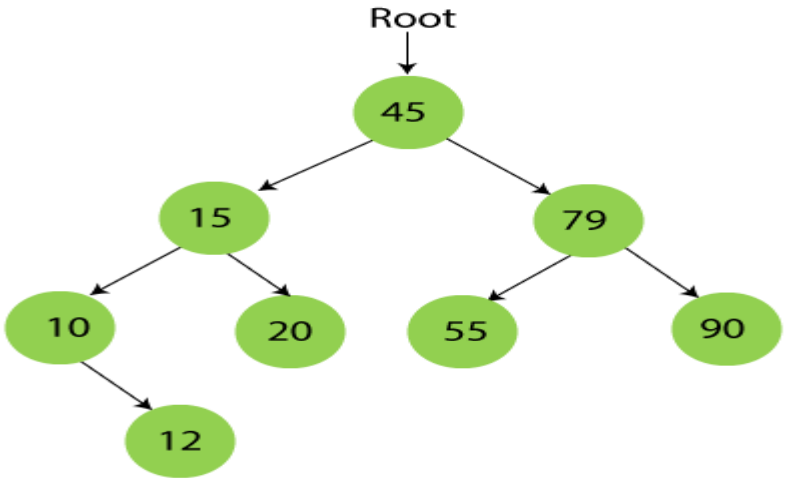
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



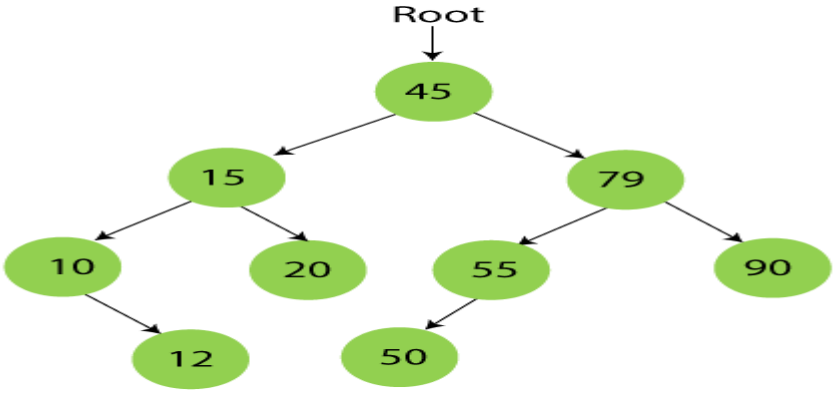
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Now, the creation of binary search tree is completed.

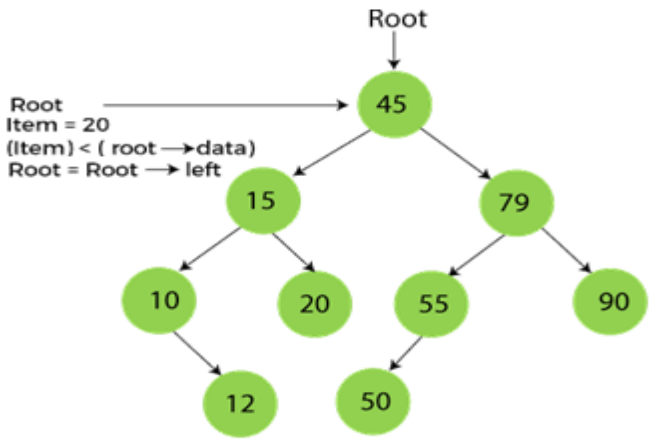
Searching in Binary search tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows –

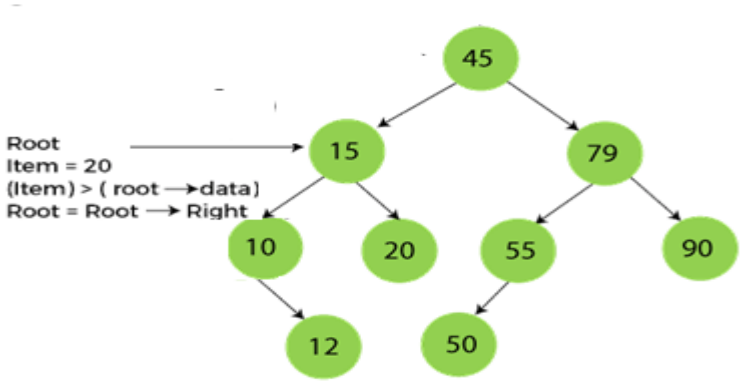
1. First, compare the element to be searched with the root element of the tree.
2. If root is matched with the target element, then return the node's location.
3. If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
4. If it is larger than the root element, then move to the right subtree.
5. Repeat the above procedure recursively until the match is found.
6. If the element is not found or not present in the tree, then return NULL.

Now, let's understand the searching in binary tree using an example. We are taking the binary search tree formed above. Suppose we have to find node 20 from the below tree.

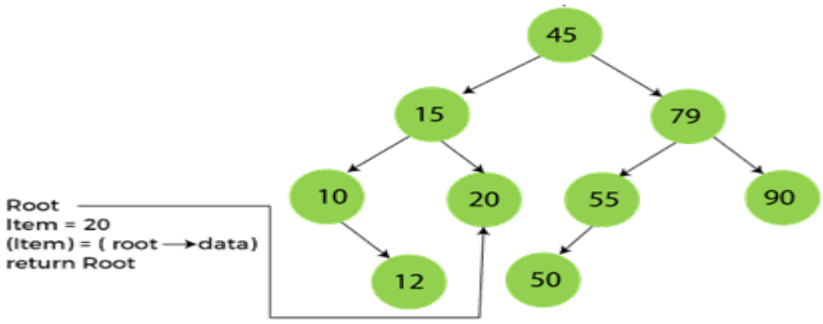
Step1:



Step2:



Step3:



Now, let's see the algorithm to search an element in the Binary search tree.

Algorithm to search an element in Binary search tree

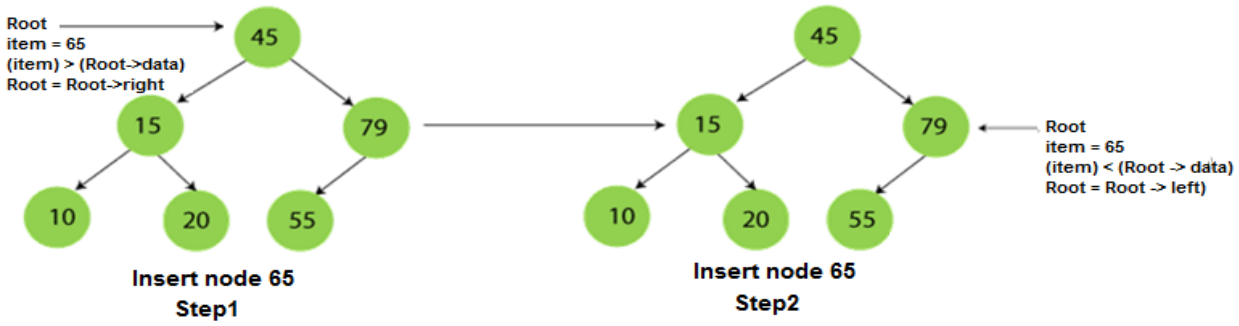
1. Search (root, item)
2. Step 1 - if (item = root → data) or (root = NULL)
3. return root
4. else if (item < **root** → data)
5. return Search(root → left, item)
6. else
7. return Search(root → right, item)
8. END if
9. Step 2 - END

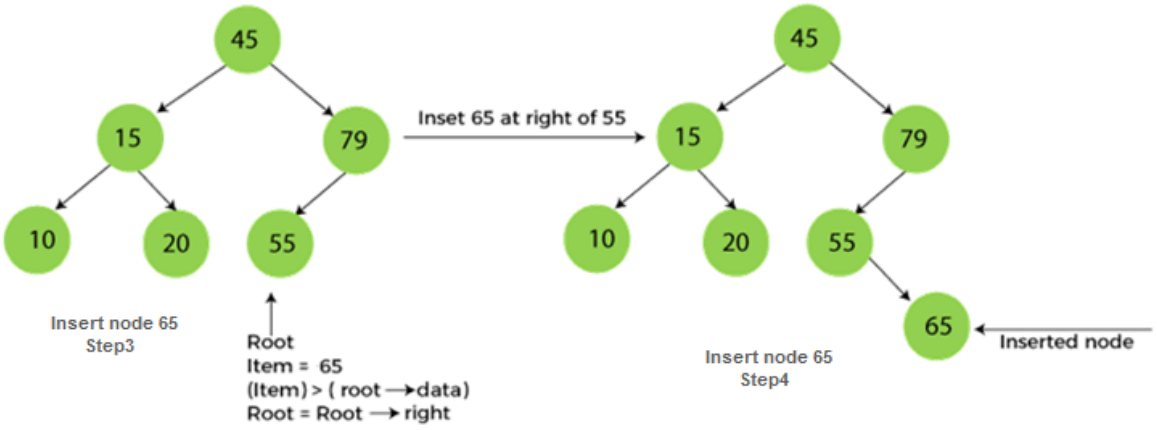
Insertion in Binary Search tree

A new key in BST is always inserted at the leaf.

To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data.

Now, let's see the process of inserting a node into BST using an example.





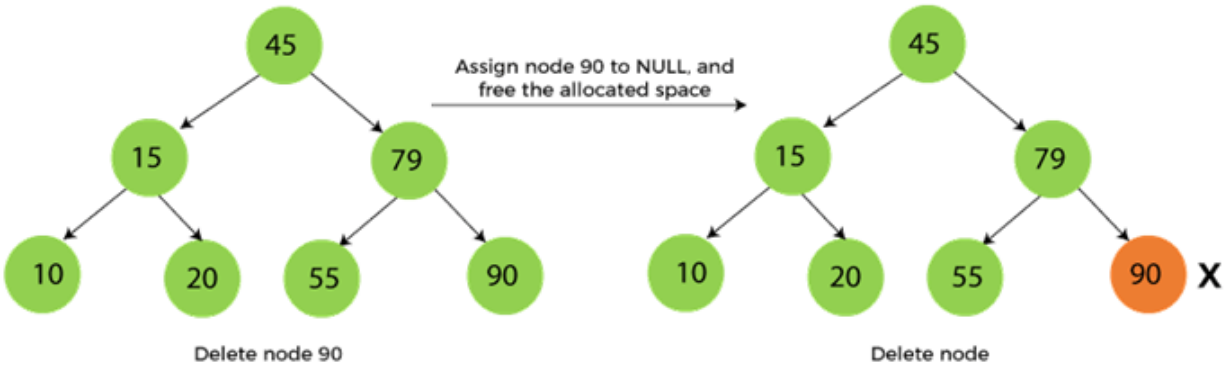
Deletion in Binary Search tree

In a binary search tree, we must delete a node from the tree by keeping in mind that the property of BST is not violated. To delete a node from BST, there are three possible situations occur -

- The node to be deleted is the leaf node, or,
- The node to be deleted has only one child, and,
- The node to be deleted has two children

1.When the node to be deleted is the leaf node

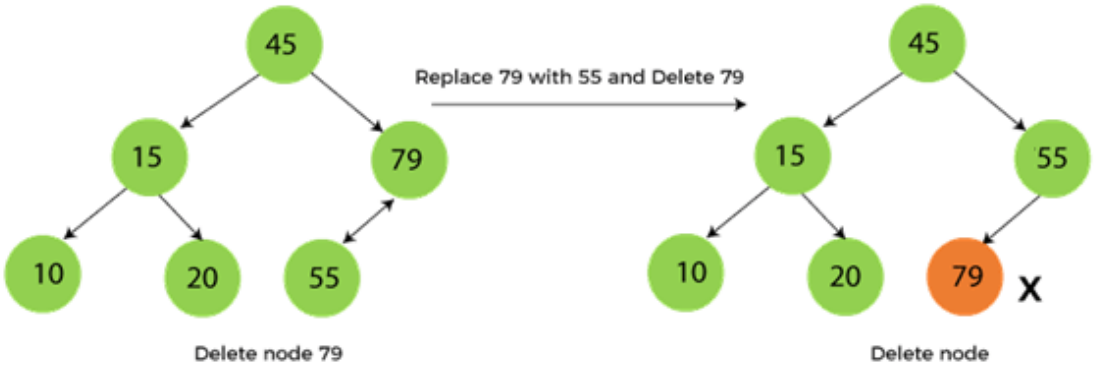
- It is the simplest case to delete a node in BST. Here, we have to replace the leaf node with NULL and simply free the allocated space.
- In below image, suppose we have to delete node 90, as the node to be deleted is a leaf node, so it will be replaced with NULL, and the allocated space will free.



2.When the node to be deleted has only one child

In this case, we have to replace the target node with its child, and then delete the child node. It means that after replacing the target node with its child node, the child node will now contain the value to be deleted. So, we simply have to replace the child node with NULL and free up the allocated space.

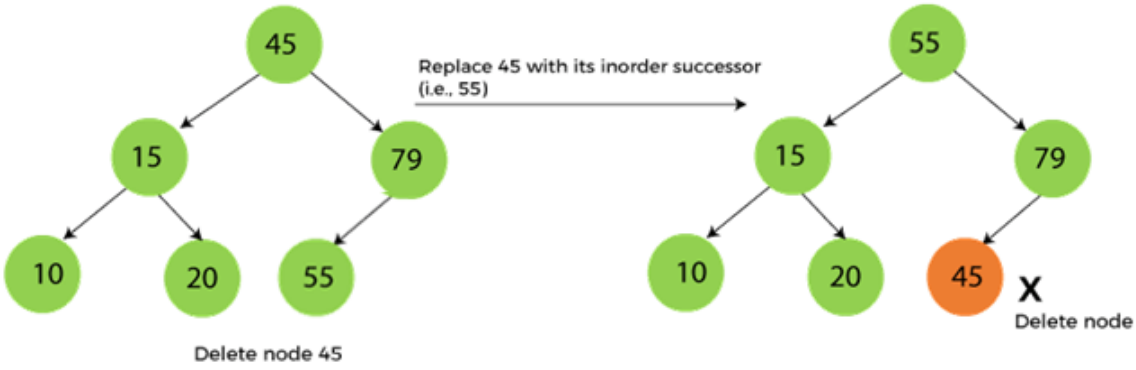
In the below image, suppose we have to delete the node 79, as the node to be deleted has only one child, so it will be replaced with its child 55.
So, the replaced node 79 will now be a leaf node that can be easily deleted.



3.When the node to be deleted has two children

If the node has 2 children ,the strategy is to place the data of this node with the smallest data of the right subtree and recursively delete that node.

In the below image, suppose we have to delete node 45 that is the root node, as the node to be deleted has two children, so it will be replaced with its inorder successor. Now, node 45 will be at the leaf of the tree so that it can be deleted easily.

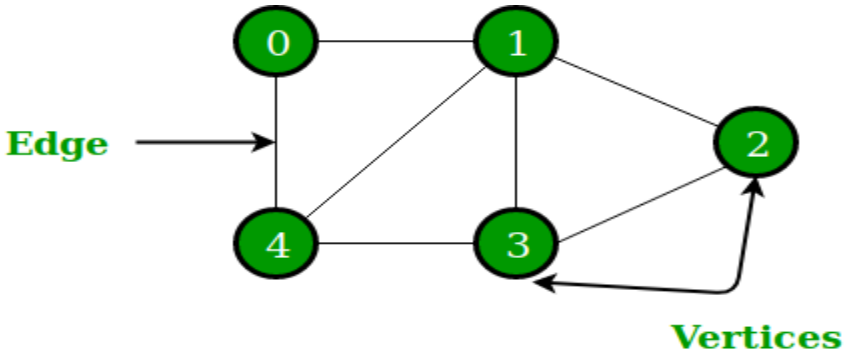


Chapter-7

GRAPH

A graph is a non-linear kind of data structure made up of nodes or vertices and edges.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices.



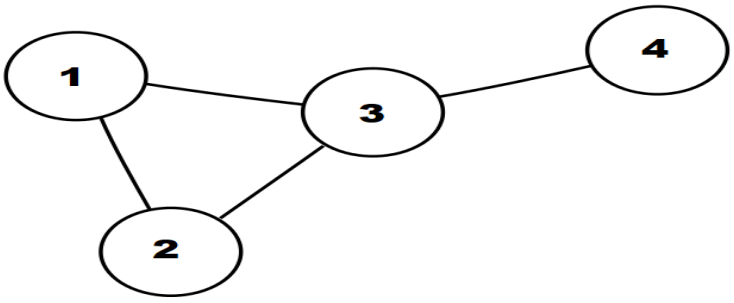
In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {01, 12, 23, 34, 04, 14, 13}.

Applications

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender etc.

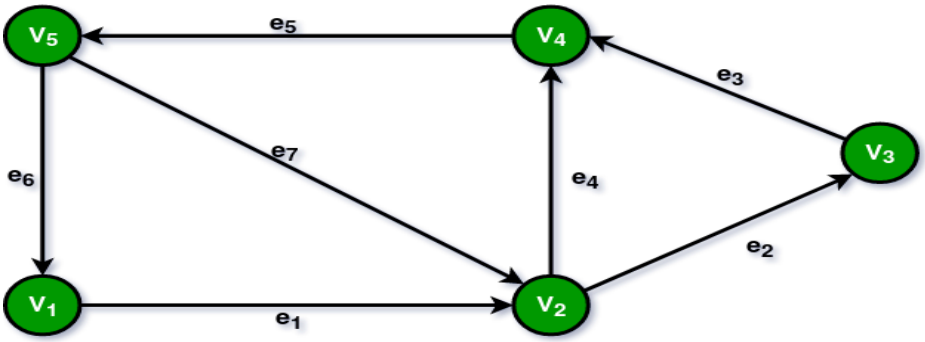
Graph Terminologies

1.Undirected: A graph in which all the edges are bi-directional. The edges do not point in a specific direction.

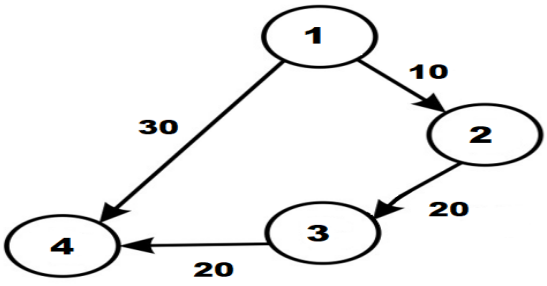


Undirected Graph

2. Directed Graph or Digraph: A graph G = (V, E) with a mapping f such that every edge maps onto some ordered pair of vertices (Vi, Vj) is called Digraph. It is also called *Directed Graph*. Ordered pair (Vi, Vj) means an edge between Vi and Vj with an arrow directed from Vi to Vj.



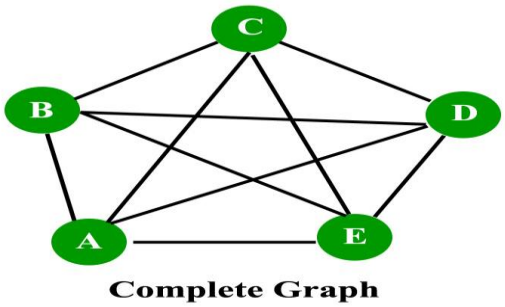
3. Weighted Graph: A graph that has a value associated with every edge. The values corresponding to the edges are called weights. A value in a weighted graph can represent quantities such as cost, distance, and time, depending on the graph. Weighted graphs are typically used in modeling computer networks.



Weighted Graph

4. Unweighted Graph: A graph in which there is no value or weight associated with the edge. All the graphs are unweighted by default unless there is a value associated.

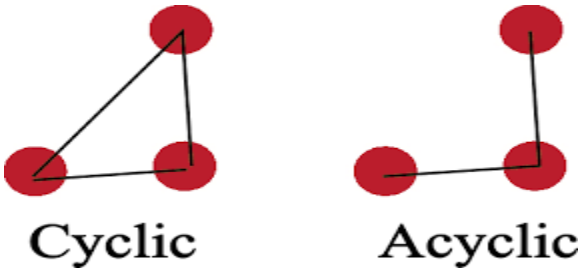
5. Complete Graph: A simple graph with n vertices is called a complete graph if the degree of each vertex is n-1, that is, one vertex is attach with n-1 edges. A complete graph is also called Full Graph.



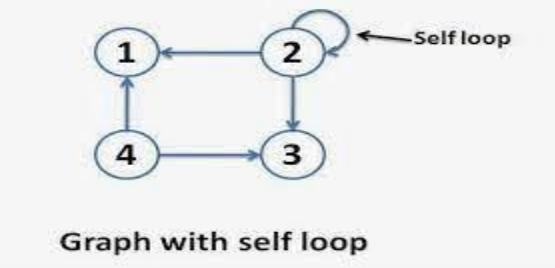
Complete Graph

6.Cyclic Graph: For a graph to be called a cyclic graph, it should consist of at least one cycle. If a graph has a minimum of one cycle present, it is called a cyclic graph.

7.Acyclic Graph: A graph is called an acyclic graph if zero cycles are present, and an acyclic graph is the complete opposite of a cyclic graph.



8 .Vertex: Every individual data element is called a vertex or a node.
9 .Edge It is a connecting link between two nodes or vertices.
10. Degree: The total number of edges connected to a vertex in a graph.
Indegree: The total number of incoming edges connected to a vertex.
Outdegree The total number of outgoing edges connected to a vertex.
11. Self-loop: If there is an edge whose stat and end vertices are same i.e.(vi,vi) is an edge ,then it is called self loop.



Graph with self loop

12. Adjacent Nodes: Two nodes are called adjacent if they are connected through an edge.
13. Path: sequence of vertices in which each pair of successive vertices is connected by an edge
14. Isolated Vertex: A vertex with degree zero is called an isolated vertex.
Example:

Here, the vertex 'a' and vertex 'b' has a no connectivity between each other and also to any other vertices. So the degree of both the vertices 'a' and 'b' are zero. These are also called as isolated vertices.

Representation of Graph

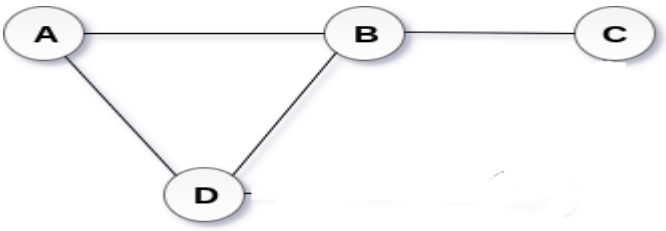
The graph can be represented as matrices.

Adjacency Matrix

Adjacency Matrix is used to represent a graph. We can represent directed as well as undirected graphs using adjacency matrices..
If a graph has n number of vertices, then the adjacency matrix of that graph is n x n, and each entry of the matrix represents the number of edges from one vertex to another.

Adjacency Matrix Representation for undirected graph

If an Undirected Graph G consists of n vertices then the adjacency matrix of a graph is n x n matrix A = [aij] and defined by -
aij = 1 {if there is a path exists from Vi to Vj}
aij = 0 {Otherwise}
In an undirected graph, if there is an edge exists between Vertex A and Vertex B, then the vertices can be transferred from A to B as well as B to A.



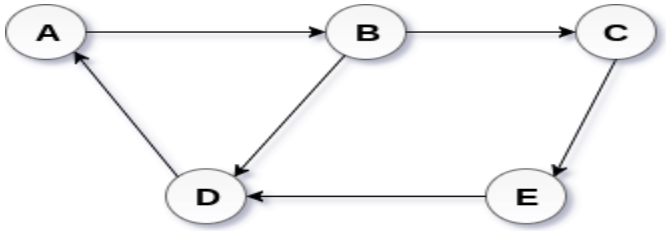
Undirected Graph

The adjacency matrix of the above graph will be -

	A	B	C	D
A	0	1	0	1
B	1	0	1	1
C	0	1	0	1
D	1	1	0	0

Adjacency matrix representation for a directed graph

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called the initial node, while node B is called the terminal node.



Directed Graph

The adjacency matrix of the above graph will be -

	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Properties of the adjacency matrix

Some of the properties of the adjacency matrix are listed as follows:

- An adjacency matrix is a matrix that contains rows and columns used to represent a simple labeled graph with the numbers 0 and 1 in the position of (V_I, V_j), according to the condition of whether or not the two V_i and V_j are adjacent.
- For a directed graph, if there is an edge exists between vertex i or V_i to Vertex j or V_j, then the value of $A[V_i][V_j] = 1$, otherwise the value will be 0.
- For an undirected graph, if there is an edge that exists between vertex i or V_i to Vertex j or V_j, then the value of $A[V_i][V_j] = 1$ and $A[V_j][V_i] = 1$, otherwise the value will be 0.

Chapter-8

SORTING SEARCHING & MERGING

Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order. Sorting can be done in ascending and descending order such as such as increasing or decreasing with numerical data or alphabetically with character data.

BUBBLE SORT

Bubble sort is a simple sorting algorithm. Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

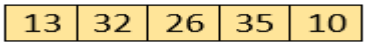
Algorithm:

a is the list of elements stored in memory to be sorted.
m is the size of the list.

```
bubbleSort(a[ ],m)
1. for(pass=0;pass<m-1;pass++)
2.         for (i=0; i<m-pass-1;i++)
3.                 if (a[i]>a[i+1])
4.                         temp = a[i];
5.                         a[i] =a[i+1];
6.                         a[i+1]=temp;
End
```

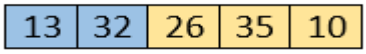
Working of Bubble sort Algorithm

To understand the working of bubble sort algorithm, let's take an unsorted array.
Let the elements of array are -

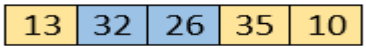


First Pass

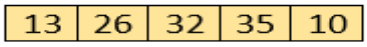
Sorting will start from the initial two elements. Let compare them to check which is greater.



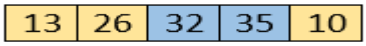
Here, 32 is greater than 13 (32 > 13), so it is already sorted. Now, compare 32 with 26.



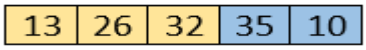
Here, 26 is smaller than 36. So, swapping is required. After swapping new array will look like -



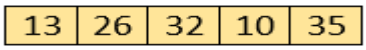
Now, compare 32 and 35.



Here, 35 is greater than 32. So, there is no swapping required as they are already sorted.
Now, the comparison will be in between 35 and 10.



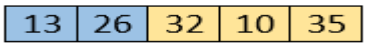
Here, 10 is smaller than 35 that are not sorted. So, swapping is required. Now, we reach at the end of the array. After first pass, the array will be -

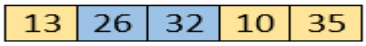


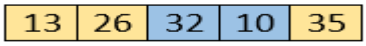
Now, move to the second iteration.

Second Pass

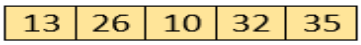
The same process will be followed for second iteration.

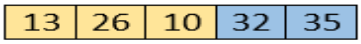






Here, 10 is smaller than 32. So, swapping is required. After swapping, the array will be -

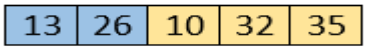


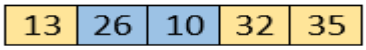


Now, move to the third iteration.

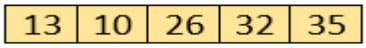
Third Pass

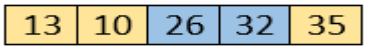
The same process will be followed for third iteration.

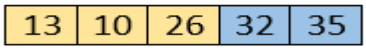




Here, 10 is smaller than 26. So, swapping is required. After swapping, the array will be -



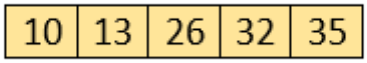




Now, move to the fourth iteration.

Fourth pass

Similarly, after the fourth iteration, the array will be -



Hence, there is no swapping required, so the array is completely sorted.

Time Complexity

Case	Time Complexity
Best Case	O(n)
Average Case	O(n2)
Worst Case	O(n2)

Space Complexity

Space Complexity	O(1)

Quick sort

Quick sort is a faster and highly efficient sorting algorithm which follows the divide and conquers approach.
Divide and conquer is a technique of breaking down the algorithms into sub problems, then solving the sub problems, and combining the results back together to solve the original problem.

Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.
Conquer: Recursively, sort two subarrays with Quicksort.
Combine: Combine the already sorted array.
Choosing the pivot
Pivot can be random, i.e. select the random pivot from the given array.
Pivot can either be the rightmost element of the leftmost element of the given array.
Select median as the pivot element.

Algorithm:

a is the list of elements stored in memory to be sorted.
beg is the initial index of the list.
end is the last index of the list.
Pivot is the position in the list where partition takes place.

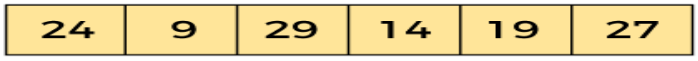
```
quickSort(a[],beg,end)
1. if(beg< end)
2.      pivot = partition(a, beg, end)
3.      quickSort(a, beg, pivot-1)
4.      quickSort(a, pivot+1, end)
End
partition(a, beg, end)
1.pivotItem = a[beg]
2.i= beg
```

```
3. for (j=beg+1; j<=end; j++)
4.         if (a[j]<= pivotItem)
5.                   i=i+1
6.                   if(i!=j)
7.                             swap(a[i],a[j])
8. swap(a[beg], a[i])
9. return i
End
```

Working of Quick Sort Algorithm

To understand the working of quick sort, let's take an unsorted array. It will make the concept more clear and understandable.

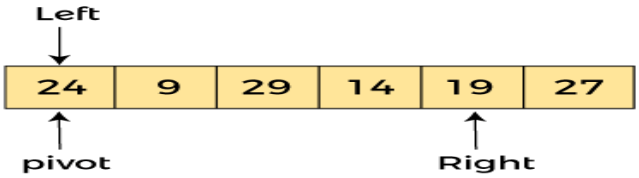
Let the elements of array are -



In the given array, we consider the leftmost element as pivot. So, in this case, a[left] = 24, a[right] = 27 and a[pivot] = 24.

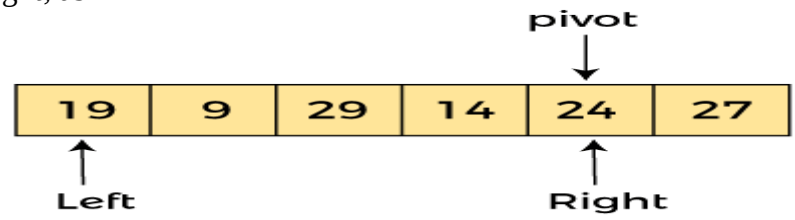
Since, pivot is at left, so algorithm starts from right and move towards left.

Now, a[pivot] < a[right], so algorithm moves forward one position towards left, i.e. –



Now, a[left] = 24, a[right] = 19, and a[pivot] = 24.

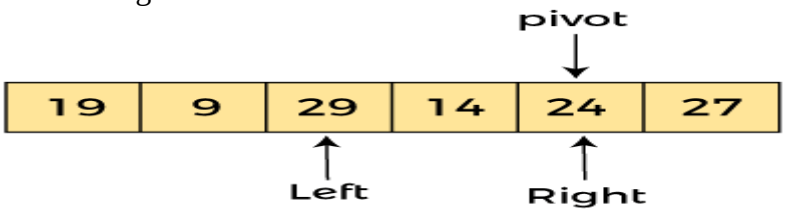
Because, a[pivot] > a[right], so, algorithm will swap a[pivot] with a[right], and pivot moves to right, as -



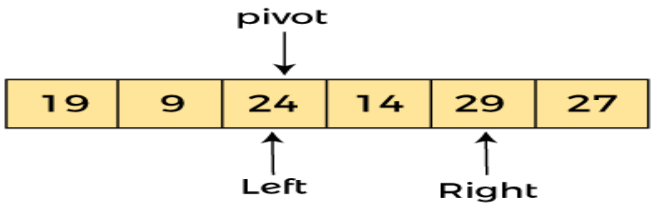
Now, a[left] = 19, a[right] = 24, and a[pivot] = 24. Since, pivot is at right, so algorithm starts from left and moves to right.

As a[pivot] > a[left], so algorithm moves one position to right as -

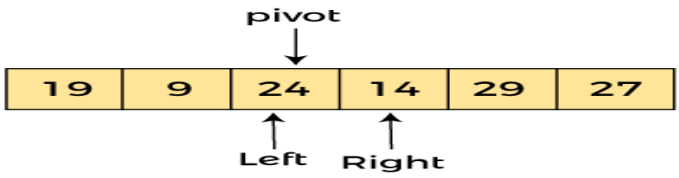
Now, a[left] = 9, a[right] = 24, and a[pivot] = 24. As a[pivot] > a[left], so algorithm moves one position to right as -



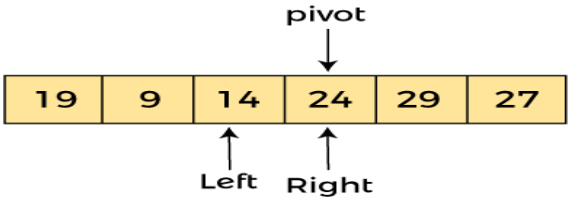
Now, a[left] = 29, a[right] = 24, and a[pivot] = 24. As a[pivot] < a[left], so, swap a[pivot] and a[left], now pivot is at left, i.e. –



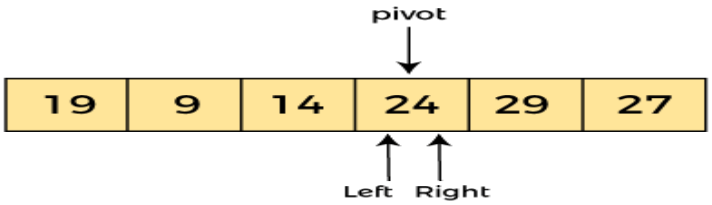
Since, pivot is at left, so algorithm starts from right, and move to left. Now, a[left] = 24, a[right] = 29, and a[pivot] = 24. As a[pivot] < a[right], so algorithm moves one position to left, as –



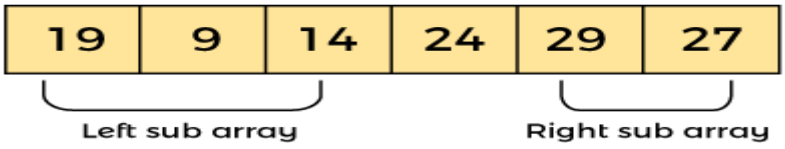
Now, a[pivot] = 24, a[left] = 24, and a[right] = 14. As a[pivot] > a[right], so, swap a[pivot] and a[right], now pivot is at right, i.e. –



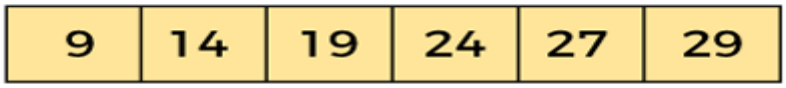
Now, a[pivot] = 24, a[left] = 14, and a[right] = 24. Pivot is at right, so the algorithm starts from left and move to right.



Now, a[pivot] = 24, a[left] = 24, and a[right] = 24. So, pivot, left and right are pointing the same element. It represents the termination of procedure.
Element 24, which is the pivot element is placed at its exact position.
Elements that are right side of element 24 are greater than it, and the elements that are left side of element 24 are smaller than it.



Now, in a similar manner, quick sort algorithm is separately applied to the left and right sub-arrays. After sorting gets done, the array will be –



1.**Time Complexity**

Case	Time Complexity
Best Case	O(n*logn)
Average Case	O(n*logn)
Worst Case	O(n2)

2. **Space Complexity**

Space Complexity	O(n*logn)

MERGING:

The operation of sorting is closely related to the process of merging. The merging of two order table which can be combined to produce a single sorted table.
This process can be accomplishes easily by successively selecting the record with the smallest key occurring by either of the table and placing this record in a new table.

SIMPLE MERGE

SIMPLE MERGE [FIRST,SECOND,THIRD,K]
Given two orders in table sorted in a vector K with FIRST, SECOND, THIRD

The variable I & J denotes the curser associated with the FIRST & SECOND table respectively. L is the index variable associated with the vector TEMP.

Algorithm

Step 1: [Initialize]
Set I = FIRST
Set J = SECOND
Set L = 0
Step 2: [Compare corresponding elements and output the smallest]
Repeat while I < SECOND & J < THIRD
If K[I] <= K[J],then L=L+1
TEMP [L]=K[I]
I=I+1
Else L=L+1
TEMP[L]=K[J]
J=J+1
Step 3: [Copy remaining unprocessed element in output area]
If I>=SECOND
Then repeat while J<=THIRD
L=L+1
TEMP[L]=K[J]
J=J+1
Else
Repeat while I< SECOND
L=L+1
TEMP[L]=K[I]
I=I+1
Step 4: [Copy the element into temporary vector into original area]
Repeat for I=1,2,…..L
K[FIRST-I+1]=TEMP[I]
Step 5: Exit

SEARCHING

Finding the location of a given item in a particular array is called as searching.
These are 2 types Searching

1. Linear Search
2. Binary Search

Linear Search

- Suppose ‘a’ is an array with n element.
- To find an item we have to search the array ‘a’ from the first element sequentially.
- This sequential search is known as Linear Search.

Algorithm:

```
void linearsearch (int a[], int n, int no)
{
int loc = -1, i ; for (i=0; i<n; i++)
{
if (a[i] == no) loc = i ;
}
if (loc == -1)
Printf ("Data not found");
else
Printf("Data found at %d position/location",loc);
}
```

Where a[] = name of the array
n = no. of element present in array 'a'
no = number to be searched loc = location to be searched

Binary Search

- Sequential Search is a simple method for searching an element in array.
- But this is not efficient for large list because we will have to make n comparison to search for the last element in the list.
- The binary search technique is very much efficient and applied only in sorted array.

Ex: Searching a name in telephone directory or searching a word in dictionary.

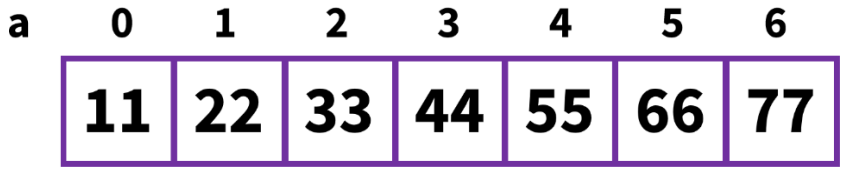
- In this method we make a comparison between the key and the middle element of the array.
- Since the array is sorted this comparison results either in a match between key and the middle element of the array or identifying the left half or result half of the array.
- When the current element is not equal to the middle element of the array, the procedure is repeated on the half in which the desired element is likely to be present.
- Proceeding in this way, either the element is detected or final division leads to a half consisting of no element.

Algorithm:

```
int BinarySearch (int key, int a[], int n)
{
int low, hi, mid;
low = 0 ;
hi = n-1;
while (low<=hi)
{
mid = (low+hi)/2;
if (key = = a[mid])
```

```
 return(mid);
if (key<a[mid])
 hi = mid-1;
else
low = mid+1;
}
return(-1);
}
```

Example:
Suppose the array elements are
11, 22, 33, 44, 55, 66, 77



We wish to search for 33 So key = 33
low = 0
hi = 6
n = 7
Is low <= hi Yes
mid = (low+hi)/2=(0+6)/2=3

Is 33 == a[3] No
Is 33 < a[3] yes
the steps will be repeated for lower half
hi = mid-1=3-1=2
low = 0
Is low <= hi
Mid=(0+2)/2=1

If key (33) == a[1] No
If 33 < a[1] No
low = 1+1 =2
hi = 2
mid=(2+2)/2=2

Is 33 == a[2]
Yes the search is successful at index 2.

Chapter-9
FILE ORGANIZATION

File:

- A file is an external collection of related data treated as a unit.
- Files are stored in auxiliary/secondary storage devices.
  - Disk
  - Tapes
- A file is a collection of data records with each record consisting of one or more fields.
- A file is a collection of data records grouped together for purpose of access control andmodification.
- It is the primary resource in which we can store information and can retrieve the information when it is required.
- The absolute file name consists of:
  - drive name
  - directory name(s)
  - file name
  - Extension

  For example: d:/network/LAN.doc

Terms Used with Files

- **Field:** This is the basic element of data which contains a single value and characterized by itslength and data type.
- **Record:** This is the collection of related fields that can be treated as a unit by some applicationprogram.

Extension	Type of Document	Application
.doc	Word-processing document	Microsoft Word 2003
.docx	Word-processing document	Microsoft Word 2007
.wks	Word-processing document	Microsoft Works word processing
.wpd	Word-processing document	Corel WordPerfect
.xls	Spreadsheet	Microsoft Excel
.slr	Spreadsheet	Microsoft Works spreadsheet
.mdb	Database	Microsoft Access

.ppt	PowerPoint Presentation	Microsoft PowerPoint
.pdf	Portable Document Format	Adobe Acrobat or Adobe Reader

File Operations:

Different types of operations can be performed on the file.

- Create
- Delete
- Open
- Close
- Read
- Write

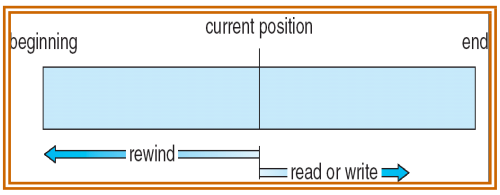
FILE ACCESSING METHODS

Information is stored in files and files reside on secondary storage. When this information is to be used, it has to be accessed and brought into primary main memory. Information in files could be accessed in many ways. There are 3 different file access methods.

- **Sequential access method**
- **Direct access method**
- **Indexed sequential access method**

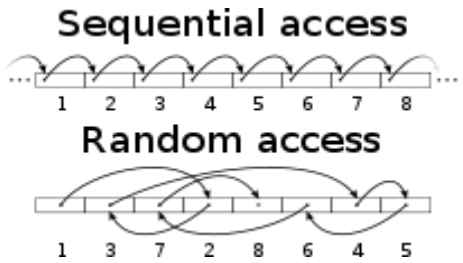
Sequential Access Method:

- This method is simplest among all methods. Information in the file is processed in order, one record after the other.
- Magnetic tapes are supporting this type of file accessing.
- Ex: A file consisting of 100 records, the current position
- of read/write head is 45th record, suppose we want to read the 75th record, then it access sequentially from 45,46,. 74,75.
- So, the read/write head traverse all the records between 45 to 75.
- Sequential files are typically used in batch application and payroll application.



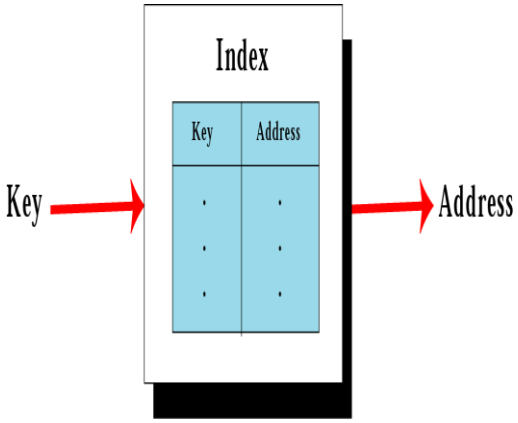
Direct access/ Random access method:

- Direct access is also called as relative access.
- In this method records can read/write randomly without any order.
- The direct access method is based on disk model of a file, because disks allow random access to any file block.
- A direct access file allows arbitrary blocks to be read or written.
- For ex. A disk consisting of 256 blocks, the current position of
- read/write head is at 95th block. The block to be read or write is 250th block. Then we can access the 250th block directly without any direction.
- Best example for direct access is a CD consisting of 10 songs, at present we are listening the song no.3, suppose we want listening the song no.9, then we can shift from song no.3 to 9 without any restriction.



Indexed sequential access method:

- The main disadvantage in the sequential file is, it takes more time to access a record, to overcome this problem, we are using this method.
- In this method, the records are stored sequentially for efficient processing. But they can be accessed directly using index or key field. That's why this method is said to be the indexed sequential file.
- Keys are pointer which contains address of various blocks.
- Records are organized in sequence based on a key field.
- Generally indexed files are used in airline reservation system and payroll packages



ADDRESS CALCULATION OR HASHING

Hashing is the function to generate a physical address from the record key value.

Key Value → Hashing → Unique physical address

So hashing is a process to generate the physical address from key value in direct access file.

H(k) = L

Where H -> function
K -> key
L -> address

Characteristics of hashing function

- It should be very easy and quick to compute.
- The hash function should be as far as possible uniformly distribute the hash address throughout the set L, so that there will be minimum no. of collision.

Types of hash function

1) Division Remainder method
2) Mid square method
3) Folding method

Division- Remainder Method

choose a number m larger than the number n of keys in K. (m is usually either a prime numberor a number without small divisor) the hash function H is defined by

H(k) = k (mod m) or **H(k) = k (mod m) + 1**.

here k (mod m) denotes the reminder when k is divided by m. the second formula is used whenwe want a hash address to range from 1 to m rather than 0 to m-1.

The remainder is calculated physical address for that record.
Suppose key value of the record is 2345.
i.e. k =2345 and m = 97 then H(k) = 2345 % 97=17
So that particular record will be sorted in 17 location of memory.
Generally m should be a prime number.

Mid Square method

The key value is squared. Then the mid digits are the address for that key.
Suppose key value = 2345
k^2 = 5499025
If the memory is of two bit then mid will be 99.

Folding Method

In folding method the key value is folded into no. of parts, where each part contains same no. of digits except possibly the last.
$H(k) = k_1 + k_2 + k_3 \ldots . k_n$
Suppose key value = 2356

k = 2356 $k_1 = 23, k_2 = 56$
23+56 = 79
So the address for that record is 79.

Collision

Suppose we have calculated an address L for a given key value k. But if that location L is previously occupied by some other record then that situation is known as collision.
So if the hash function will generate same address for two different key values. Then it is said tobe collision.
Consider the division remainder method.Let $k_1 = 150$, $k_2 = 184$, m =17
$H(k_1) = 150 \mod 17 = 14$
$H(k_2) = 184 \mod 17 = 14$
So for key values 150 and 184 the calculated address is 14. This is collision.

There are various collision resolution technique.

- Linear probing (also called open addressing or array method)
- Quadratic Probing
- Double hashing
- Separate Chaining (also called closed hashing or linked list method)

Advantages

- Accessing a record is faster.
- No need to arrange the record on a sorted order.
- If required, records can be processed sequentially.
- Insertion, deletion is easier

Disadvantage

- Expensive I/O device as compared to sequential file structure.
- Address generation overhead due to hashing function.
- Updation of all record is inefficient as compared to sequential file.

Application

Interactive online application such as airline and railway reservation system.

References

1. Book Reference: Data Structure by S. Lipschutz Schaum Series
2. Book Reference: Data Structures Using C
   by Amiya Kumar Rath (Author), Alok Kumar Jagadev (Author)
3. Book Reference: Data Structures Using C by Udit Agrawal
4. https://nptel.ac.in
5. https://www.javatpoint.com
6. https://www.geeksforgeeks.org