ANNAMACHARYA INSTITUTE OF TECHNOLOGY & SCIENCES::KADAPA
(AUTONOMOUS)

(Approved by AICTE New Delhi & Affiliated to JNTUA, Anantapuramu)Accredited by NAAC with 'A' grade, Bangalore)

DATA STRUCTURES

(Common to CSE, IT & allied branches)

Course Objectives:

- To provide the knowledge of basic data structures and their implementations.
- To understand importance of data structures in context of writing efficient programs.
- To develop skills to apply appropriate data structures in problem solving.

Course Outcomes: At the end of the course, Student will be able to

CO1: Explain the role of linear data structures in organizing and accessing data efficientlyin algorithms.
CO2: Design, implement, and apply linked lists for dynamic data storage,demonstrating understanding of memory allocation.
CO3: Develop programs using stacks to handle recursive algorithms, manage program states, and solve related problems.
CO4: Apply queue-based algorithms for efficient task scheduling, distinguish between deques and priority queues, and apply them appropriately to solve data management challenges.
CO5: Explain the role of non-linear data structures in organizing and tree traversalsCO6: Recognize scenarios where hashing is advantageous, and design hash-based solutions forspecific problems.

UNIT I

Introduction to Linear Data Structures: Definition and importance of linear data structures, Abstract data types (ADTs), Overview of time and space complexity analysis for linear data structures. Searching Techniques: Linear & Binary Search, Sorting Techniques: Bubble sort, Selection sort, Insertion Sort.

UNIT II

Linked Lists: Singly linked lists: representation and operations, Doubly linked lists and circular linked lists, Comparing arrays and linked lists, Applications of linked lists.

UNIT III

Stacks: Introduction to stacks: properties and operations, Implementing stacks using arrays and linked lists, Applications of stacks in expression evaluation.

Queues: Introduction to queues: properties and operations, implementing queues using arrays and linked lists, Applications of queues ,scheduling, etc.

UNIT IV

Deques: Introduction to deques (double-ended queues),Operations on deques and their applications.

Hashing: Brief introduction to hashing and hash functions, Collision resolution techniques: chaining and open addressing, Hash tables: basic implementation and operations, Applicationsof hashing in unique identifier generation, caching, etc.

UNIT V

Trees: Introduction to Trees, Binary Search Tree – Insertion, Deletion & Traversal

Graphs: Introductions to Graphs,DFS&BFS

Textbooks:

1. Data Structures and algorithm analysis in C, Mark Allen Weiss, Pearson, 2nd Edition.
2. Fundamentals of data structures in C, Ellis Horowitz, Sartaj Sahni, Susan AndersonFreed,Silicon Press, 2008

Reference Books:

1. Algorithms and Data Structures: The Basic Toolbox by Kurt Mehlhorn and Peter Sanders
2. C Data Structures and Algorithms by Alfred V. Aho, Jeffrey D. Ullman, and John E. Hopcroft
3. Problem Solving with Algorithms and Data Structures" by Brad Miller and David Ranum
4. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein

Algorithms in C, Parts 1-5 (Bundle): Fundamentals, Data Structures, Sorting, Searching, and GraphAlgorithms by Robert Sedgewick.

UNIT I
INTRODUCTION TO LINEAR DATA STRUCUTRE

1.1.DATA STRUCTURES:

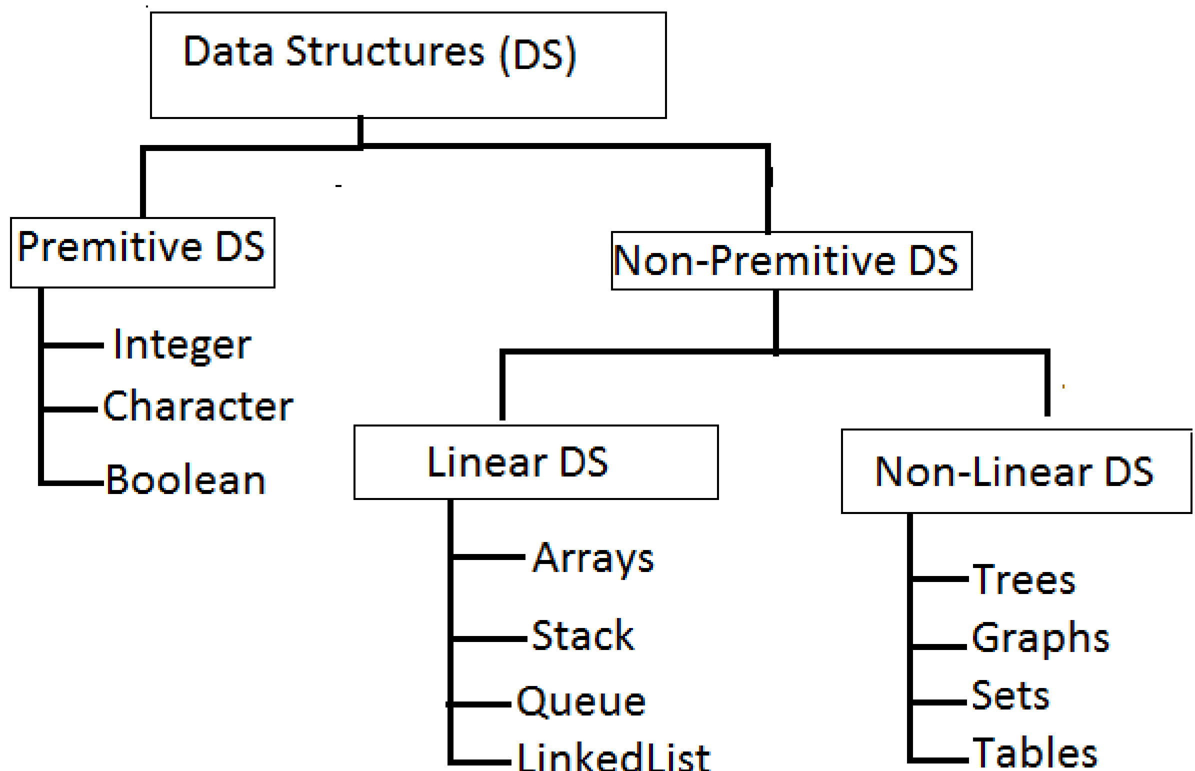
Data may be organized in many different ways logical or mathematical model of a program particularly organization of data. This organized data is called “Data Structure”.

Or

The organized collection of data is called a ‘Data Structure’.

Data Structure=Organized data +Allowed operations

Data Structure involves two complementary goals. The first goal is to identify and develop useful, mathematical entities and operations and to determine what class of problems can be solved by using these entities and operations. The second goal is to determine representation for those abstract entities to implement abstract operations on this concrete representation.



Primitive Data structures are directly supported by the language ie; any operation is directly performed in these data items.
Ex: integer, Character, Real numbers etc.
Non-primitive data types are not defined by the programming language, but are instead created by the programmer. Linear data structures organize their data elements in a linear fashion, where data elements are attached one after the other. Linear data structures are very easy to implement, since the memory of the computer is also organized in a linear fashion. Some commonly used linear data structures are arrays, linked lists, stacks and queues.
In nonlinear data structures, data elements are not organized in a sequential fashion. Data structures like multidimensional arrays, trees, graphs, tables and sets are some examples of widely used nonlinear data structures.

Operations on the Data Structures:

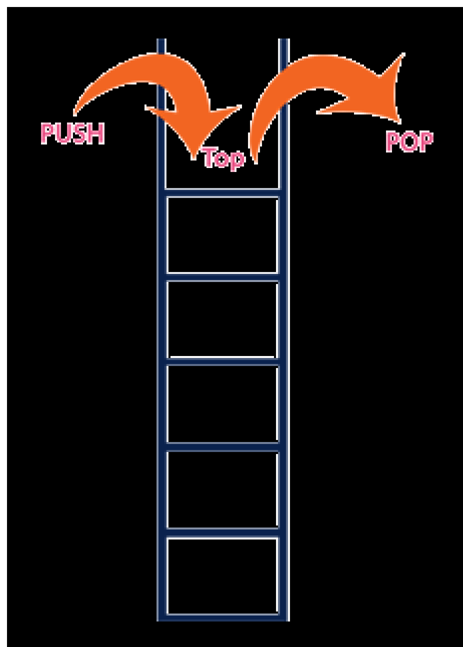
Following operations can be performed on the data structures:
1. Traversing- It is used to access each data item exactly once so that it can be processed.
2. Searching- It is used to find out the location of the data item if it exists in the given collection of data items.
3. Inserting- It is used to add a new data item in the given collection of data items.
4. Deleting- It is used to delete an existing data item from the given collection of data items.

Linear Data Structure

A linear data structure is a structure in which the elements are stored sequentially, and the elements are connected to the previous and the next element. As the elements are stored sequentially, so they can be traversed or accessed in a single run. The implementation of linear data structures is easier as the elements are sequentially organized in memory. The data elements in an array are traversed one after another and can access only one element at a time.
The types of linear data structures are Array, Queue, Stack, Linked List. Array is a type of data structure that stores data elements in adjacent locations. Array is considered as linear data structure that stores elements of same data types.

STACK:

Stack is a linear data structure in which the insertion and deletion operations are performed at only one end. In a stack, adding and removing of elements are performed at a single position which is known as "**top**".

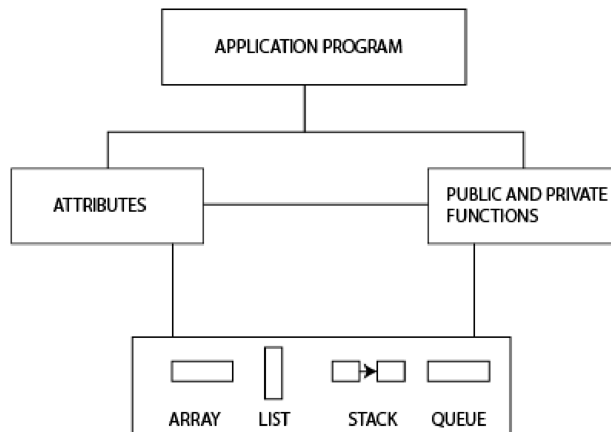


QUEUE:
Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends.

LINKED LIST:
A **linked list** is a way to store a collection of elements. Each element in a linked list is stored in the form of a **node**. A **data** part stores the element and a **next** part stores the link to the next node.

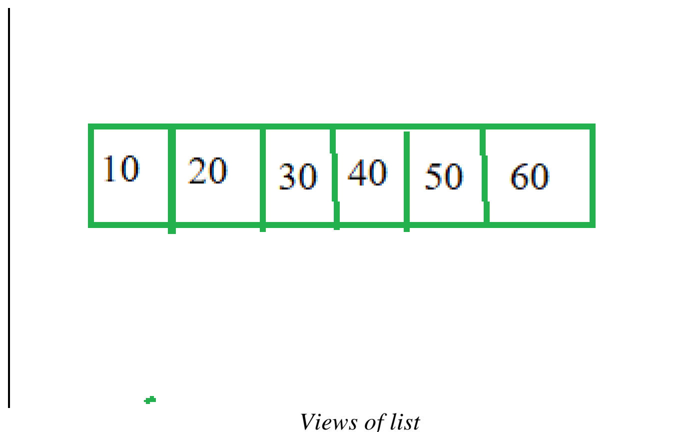
ARRAY:
An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations.

1.2.ABSTRACT DATA TYPE MODEL:
An Abstract Data Type (ADT) is a programming concept that defines a high-level view of a data structure, without specifying the implementation details. In other words, it is a blueprint for creating a data structure that defines the behavior and interface of the structure, without specifying how it is implemented.



ABSTRACT DATA TYPES:

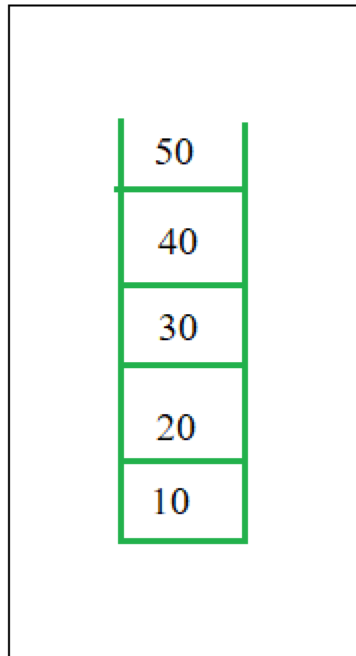
1. List ADT



Views of list

- The data is generally stored in key sequence in a list which has a head structure consisting of *count*, *pointers* and *address of compare function* needed to compare the data in the list.
- The data node contains the *pointer* to a data structure and a *self-referential pointer* which points to the next node in the list.

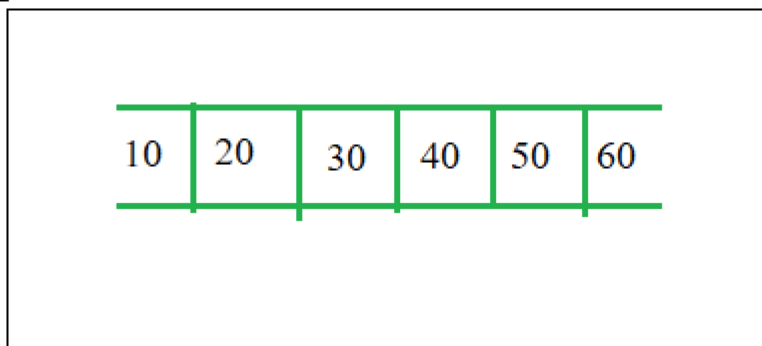
2. Stack ADT



View of stack

In Stack ADT Implementation instead of data being stored in each node, the pointer to data is stored.
The program allocates memory for the *data* and *address* is passed to the stack ADT.
The head node and the data nodes are encapsulated in the ADT. The calling function can only see the pointer to the stack.
The stack head structure also contains a pointer to *top* and *count* of number of entries currently in stack.

.
3. Queue ADT



View of Queue

The queue abstract data type (ADT) follows the basic design of the stack abstract data type.
Each node contains a void pointer to the *data* and the *link pointer* to the next element in the queue. The program's responsibility is to allocate memory for storing the data.

.

1.3.OVERVIEW OF TIME AND SPACE COMPLEXITY :

Analyzing an algorithm means determining the amount of resources (such as time and memory) needed to execute it. Algorithms are generally designed to work with an arbitrary number of inputs, so the efficiency or complexity of an algorithm is stated in terms of time and space complexity.
The *time complexity* of an algorithm is basically the running time of a program as a function of the input size. Similarly, the *space complexity* of an algorithm is the amount of computer memory that is required during the program execution as a function of the input size.
In other words, the number of machine instructions which a program executes is called its time complexity. This number is primarily dependent on the size of the program's input and the algorithm used.

Time Complexity:

The amount of time required for an algorithm to complete its execution is its time complexity. An algorithm is said to be efficient if it takes the minimum (reasonable) amount of time to complete its execution.

The number of steps any problem statement is assigned depends on the kind of statement.
For example, comments 0 steps.

1. We introduce a variable, count into the program statement to increment count with initial value 0.Statement to increment count by the appropriate amount are introduced into the program.
This is done so that each time a statement in the original program is executes count is incremented by the step count of that statement.
Algorithm:

```
Algorithm sum(a,n)
{
s= 0.0;
count = count+1;
for I=1 to n do 8
{
count =count+1;
s=s+a[I];
count=count+1;
```

```
}
count=count+1;
count=count+1;
return s;
}
```

If the count is zero to start with, then it will be 2n+3 on termination. So each invocation of sum execute a total of 2n+3 steps.

2. The second method to determine the step count of an algorithm is to build a table in which we list the total number of steps contributes by each statement.

First determine the number of steps per execution (s/e) of the statement and the total number of times (ie., frequency) each statement is executed.

By combining these two quantities, the total contribution of all statements, the step count for the entire algorithm is obtained.

Statement	S/e	Frequency	Total
1. Algorithm Sum(a,n) 2.{ 3. S=0.0; 4. for I=1 to n do 5. s=s+a[I]; 6. return s; 7. }	0 0 1 1 1 1 0	- - 1 n+1 n 1 -	0 0 1 n+1 n 1 0
Total		2n+3	

Space Complexity:

The amount of space occupied by an algorithm is known as Space Complexity. An algorithm is said to be efficient if it occupies less space and required the minimum amount of time to complete its execution.

***Fixed part*:**

It varies from problem to problem. It includes the space needed for storing instructions, constants, variables, and structured variables (like arrays and structures).

***Variable part*:**

It varies from program to program. It includes the space needed for recursion stack, and for structured variables that are allocated space dynamically during the runtime of a program.

The space requirement s(p) of any algorithm p may therefore be written as,

S(P) = c+ Sp(Instance characteristics)

Where 'c' is a constant.

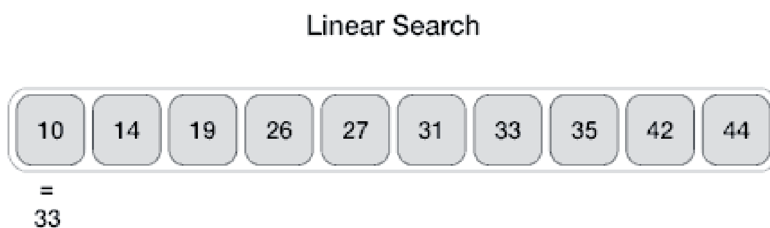
1.4. Searching:

Searching is a process of finding a particular record, which can be a single element or a small chunk, within a huge amount of data. The data can be in various forms: arrays, linked lists, trees, heaps, and graphs etc. With the increasing amount of data nowadays, there are multiple techniques to perform the searching operation.is a process of finding a particular record, which can be a single element or a small chunk, within a huge amount of data. The data can be in various forms: arrays, linked lists, trees, heaps, and graphs etc. With the increasing amount of data nowadays, there are multiple techniques to perform the searching operation.

1.4.1. Linear search

Linear search is a type of sequential searching algorithm. In this method, every element within the input array is traversed and compared with the key element to be found. If a match is found in the array the search is said to be successful; if there is no match found the search is said to be unsuccessful and gives the worst-case time complexity.

For instance, in the given animated diagram, we are searching for an element 33. Therefore, the linear search method searches for it sequentially from the very first element until it finds a match. This returns a successful search.



In the same diagram, if we have to search for an element 46, then it returns an unsuccessful search since 46 is not present in the input.

Algorithm: LINEAR(DATA, N,ITEM, LOC)
Here DATA is a linear Array with N elements. And ITEM is a given item of information. This algorithm finds the location LOC of an ITEM in DATA. LOC=-1 if the search is unsuccessful.
Step 1: Set DATA[N+1]=ITEM
Step 2: Set LOC=1
Step 3: Repeat while (DATA [LOC] != ITEM)
Set LOC=LOC+1

Step 4: if LOC=N+1 then
Set LOC= -1.
Step 5: Exit

Pseudocode

```
procedure linear_search (list, value)

   for each item in the list

      if match item == value

         return the item's location

      end if

   end for

end procedure
```

Analysis

Linear search traverses through every element sequentially therefore, the best case is when the element is found in the very first iteration. The best-case time complexity would be **O(1)**.

However, the worst case of the linear search method would be an unsuccessful search that does not find the key value in the array, it performs n iterations. Therefore, the worst-case time complexity of the linear search algorithm would be **O(n)**.

Example

Let us look at the step-by-step searching of the key element (say 47) in an array using the linear search method.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78

Step 1 : The linear search starts from the 0th index. Compare the key element with the value in the 0th index, 34.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78
=							
47							

However, 47 ≠ 34. So it moves to the next element.

Step 2 : Now, the key is compared with value in the 1st index of the array.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78
	=						
	47						

Still, 47 ≠ 10, making the algorithm move for another iteration.

Step 3 : The next element 66 is compared with 47. They are both not a match so the algorithm compares the further elements.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78
		=					
		47					

Step 4 : Now the element in 3rd index, 27, is compared with the key value, 47.

They are not equal so the algorithm is pushed forward to check the next element.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78
			=				
			47				

Step 5 : Comparing the element in the 4th index of the array, 47, to the key 47. It is figured that both the elements match. Now, the position in which 47 is present, i.e., 4 is returned.

0	1	2	3	4	5	6	7
34	10	66	27	47	8	55	78
				=			
				47			

The output achieved is “Element found at 4th index”.

Implementation

The Linear Search program can be seen implemented in four programming languages. The function compares the elements of input with the key value and returns the position of the key in the array or an unsuccessful search prompt if the key is not present in the array.

Program for Linear Search:

```
#include <stdio.h>

void linear_search(int a[], int n, int key){

   int i, count = 0;
```

```
  for(i = 0; i < n; i++) {
    if(a[i] == key) { // compares each element of the array
      printf("The element is found at %d position\n", i+1);
      count = count + 1;
    }
  }
  if(count == 0) // for unsuccessful search
    printf("The element is not present in the array\n");
}
int main(){
  int i, n, key;
  n = 6;
  int a[10] = {12, 44, 32, 18, 4, 10};
  key = 18;
  linear_search(a, n, key);
  key = 23;
  linear_search(a, n, key);
  return 0;
}
```

Output:

The element is found at 4 position

The element is not present in the array

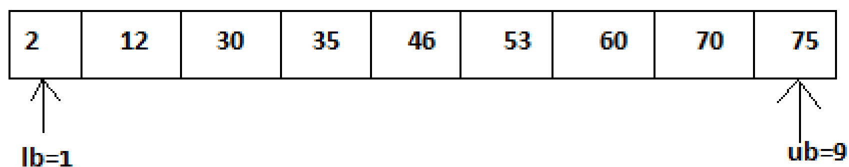
1.4.2. BINARY SEARCH:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer, since it

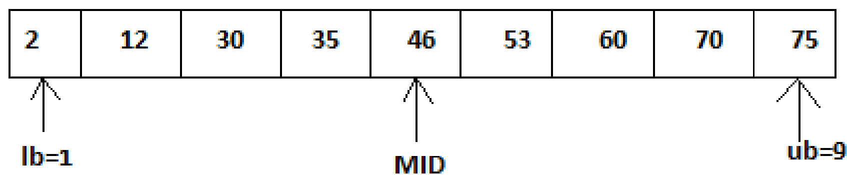
divides the array into half before searching. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular key value by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. But if the middle item has a value greater than the key value, the right sub-array of the middle item is searched. Otherwise, the left sub-array is searched. This process continues recursively until the size of a subarray reduces to zero.

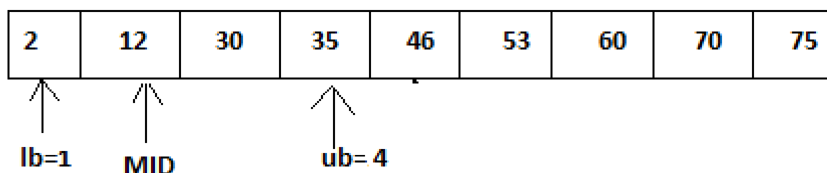
Ex: consider a list of sorted elements stored in an Array A is



Let the key element which is to be searched is 35.
Key=35
The number of elements in the list n=9.
Step 1: MID= [lb+ub]/2
=(1+9)/2
=5

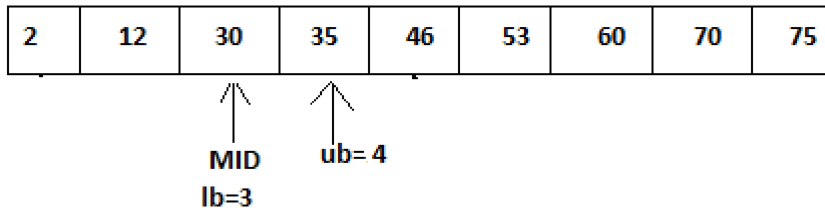


Key<A[MID]
i.e. 35<46.
So search continues at lower half of the array.
Ub=MID-1
=5-1 = 4.
Step 2: MID= [lb+ub]/2
=(1+4)/2
=2.

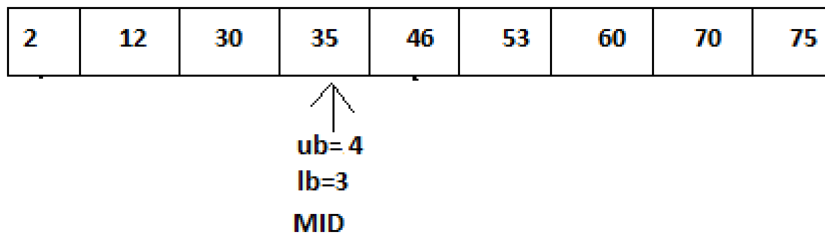


Key>A[MID]
i.e. 35>12.
So search continues at Upper Half of the array.

Lb=MID+1
=2+1
= 3. 50
Step 3: MID= [lb+ub]/2
=(3+4)/2
=3.



Key>A[MID]
i.e. 35>30.
So search continues at Upper Half of the array.
Lb=MID+1
=3+1
= 4.



Step 4: MID= [lb+ub]/2
=(4+4)/2
=4.

ALGORITHM:
BINARY SEARCH[A,N,KEY]
Step 1: begin
Step 2: [Initilization]
    Lb=1; ub=n;
Step 3: [Search for the ITEM]
    Repeat through step 4,while Lower bound is less than Upper Bound.
Step 4: [Obtain the index of middle value]
    MID=(lb+ub)/2
Step 5: [Compare to search for ITEM]
    If Key<A[MID] then
    Ub=MID-1
    Other wise if Key >A[MID] then
    Lb=MID+1
    Otherwise write “Match Found”
    Return Middle.

Step 6: [Unsuccessful Search]
write "Match Not Found"
Step 7: Stop.

Implementation:

Binary search is a fast search algorithm with run-time complexity of O(log n). This search algorithm works on the principle of divide and conquer. For this algorithm to work properly, the data collection should be in a sorted form.

Program for binary search:

```
#include<stdio.h>

void binary_search(int a[], int low, int high, int key){

  int mid;

  mid = (low + high) / 2;

  if (low <= high) {

    if (a[mid] == key)

      printf("Element found at index: %d\n", mid);

    else if(key < a[mid])

      binary_search(a, low, mid-1, key);

    else if (a[mid] < key)

      binary_search(a, mid+1, high, key);

  } else if (low > high)

    printf("Unsuccessful Search\n");

}

int main(){

  int i, n, low, high, key;

  n = 5;

  low = 0;
```

```
  high = n-1;

  int a[10] = {12, 14, 18, 22, 39};

  key = 22;

  binary_search(a, low, high, key);

  key = 23;

  binary_search(a, low, high, key);

  return 0;

}
```

Output

Element found at index: 3

Unsuccessful Search

Advantages: When the number of elements in the list is large, Binary Search executed faster than linear search. Hence this method is efficient when number of elements is large.

Disadvantages: To implement Binary Search method the elements in the list must be in sorted order, otherwise it fails.

1.5. SORTING

INTRODUCTION

Sorting is a technique of organizing the data. It is a process of arranging the records, either in ascending or descending order i.e. bringing some order lines in the data. Sort methods are very important in Data structures.

Sorting can be performed on any one or combination of one or more attributes present in each record. It is very easy and efficient to perform searching, if data is stored in sorting order. The sorting is performed according to the key value of each record. Depending up on the makeup of key, records can be stored either numerically or alphanumerically. In numerical sorting, the records arranged in ascending or descending order according to the numeric value of the key.

1.5.1. BUBBLE SORT

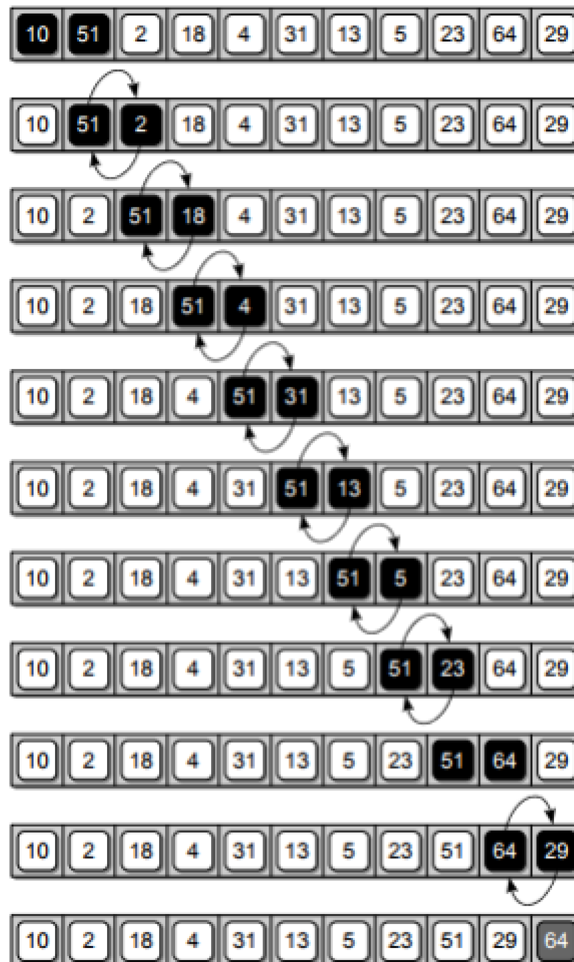
Bubble Sort: This sorting technique is also known as exchange sort, which arranges values by iterating over the list several times and in each iteration the larger value gets bubble up to the end of the list. This algorithm uses multiple passes and in each pass the first and second data items are compared. if the first data item is bigger than the second, then the two items are swapped. Next the items in second and third position are compared and if the first one is larger

than the second, then they are swapped, otherwise no change in their order. This process continues for each successive pair of data items until all items are sorted.

Bubble Sort Algorithm:

```
Step 1: Repeat Steps 2 and 3 for i=1 to 10
Step 2: Set j=1
Step 3: Repeat while j<=n
                    if a[i] < a[j] Then
                              interchange a[i] and a[j]
                    [End of if]
           Set j = j+1
          [End of Inner Loop]
[End of Step 1 Outer Loop]
Step 4: Exit
```

Example1:



Various Passes of Bubble Sort

Example 2:

To discuss bubble sort in detail, let us consider an arrayA[]that has the followingelements:

A[] = {30, 52, 29, 87, 63, 27, 19, 54}

Pass 1:

Compare 30 and 52. Since 30 < 52, no swapping is done.
Compare 52 and 29. Since 52 > 29, swapping is done. 30, **29, 52**, 87, 63, 27, 19, 54
Compare 52 and 87. Since 52 < 87, no swapping is done.
Compare 87 and 63. Since 87 > 63, swapping is done. 30, 29, 52, **63, 87**, 27, 19, 54
Compare 87 and 27. Since 87 > 27, swapping is done. 30, 29, 52, 63, **27, 87**, 19, 54
Compare 87 and 19. Since 87 > 19, swapping is done. 30, 29, 52, 63, 27, **19**, **87**, 54
Compare 87 and 54. Since 87 > 54, swapping is done. 30, 29, 52, 63, 27, 19, **54, 87**
Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Pass 2:

Compare 30 and 29. Since 30 > 29, swapping is done. **29, 30**, 52, 63, 27, 19, 54, 87
Compare 30 and 52. Since 30 < 52, no swapping is done.
Compare 52 and 63. Since 52 < 63, no swapping is done.
Compare 63 and 27. Since 63 > 27, swapping is done. 29, 30, 52, **27, 63**, 19, 54, 87
Compare 63 and 19. Since 63 > 19, swapping is done.
29, 30, 52, 27, **19, 63**, 54, 87
Compare 63 and 54. Since 63 > 54, swapping is done. 29, 30, 52, 27, 19, **54, 63**, 87
Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Pass 3:

Compare 29 and 30. Since 29 < 30, no swapping is done.
Compare 30 and 52. Since 30 < 52, no swapping is done.
Compare 52 and 27. Since 52 > 27, swapping is done. 29, 30, **27, 52**, 19, 54, 63, 87
Compare 52 and 19. Since 52 > 19, swapping is done. 29, 30, 27**, 19, 52**, 54, 63, 87
Compare 52 and 54. Since 52 < 54, no swapping is done.
Observe that after the end of the third pass, the third largest element is placed at

the third highest index of the array. All the other elements are still unsorted.

Pass 4:

Compare 29 and 30. Since 29 < 30, no swapping is done.
Compare 30 and 27. Since 30 > 27, swapping
is done. 29, **27, 30**, 19, 52, 54, 63, 87
Compare 30 and 19. Since 30 > 19, swapping
is done. 29, 27, **19, 30**, 52, 54, 63, 87
Compare 30 and 52. Since 30 < 52, no swapping is done.
Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

Pass 5:

Compare 29 and 27. Since 29 > 27, swapping
is done. **27, 29,** 19, 30, 52, 54, 63, 87
Compare 29 and 19. Since 29 > 19, swapping
is done. 27, **19, 29**, 30, 52, 54, 63, 87
Compare 29 and 30. Since 29 < 30, no swapping is done.
Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

Pass 6:

Compare 27 and 19. Since 27 > 19, swapping
is done. **19, 27,** 29, 30, 52, 54, 63, 87
Compare 27 and 29. Since 27 < 29, no swapping is done.
Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

Pass 7:

(a) Compare 19 and 27. Since 19 < 27, no swapping is done.
Observe that the entire list is sorted now.

Advantages :

- Simple and easy to implement
- In this sort, elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

Disadvantages :

- It is slowest method . $O(n^2)$
- Inefficient for large sorting lists.

Program

```
#include<stdio.h>
void main ()
{
int i, j,temp;
int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
for(i = 0; i<10; i++)
{
```

```
for(j = i+1; j<10; j++)
{
if(a[j] > a[i])
{
temp = a[i];
a[i] = a[j];
a[j] = temp;
}
}
}
printf("Printing Sorted Element List ...\n");
for(i = 0; i<10; i++)
{
printf("%d\n",a[i]);
}
}
```

Output:

Printing Sorted Element List . . .
7
9
10
12
23
34
34
44
78
101

1.5.2. SELECTION SORT

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array. The array with n elements is sorted by using n-1 pass of selection sort algorithm.

Algorithm for selection sort

SELECTION SORT(ARR, N)

Step 1: Repeat Steps 2 and 3 for K = 1 to N-1
Step 2: CALL SMALLEST(ARR, K, N, POS)
Step 3: SWAP A[K] with ARR[POS]
[END OF LOOP]
Step 4: EXIT

SMALLEST (ARR, K, N, POS)

Step 1: [INITIALIZE] SET SMALL = ARR[K]
Step 2: [INITIALIZE] SET POS = K

Step 3: Repeat for J = K+1 to N
IF SMALL > ARR[J]
SET SMALL = ARR[J]
SET POS = J
[END OF IF]
[END OF LOOP]
Step 4: RETURN POS

Example 1: 3, 6, 1, 8, 4, 5

Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

Example2 :

Example: Consider the following array with 6 elements. Sort the elements of the array by using selection sort.

A = {10, 2, 3, 90, 43, 56}.

Pass	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
1	2	10	3	90	43	56
2	2	3	10	90	43	56
3	2	3	10	90	43	56
4	2	3	10	43	90	56
5	2	3	10	43	56	90

Sorted A = {2, 3, 10, 43, 56, 90}

Advantages:

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

Disadvantages:

- Running time of Selection sort algorithm is very poor of 0 (n2).

 However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

Program

```
#include<stdio.h>
int smallest(int[],int,int);
void main ()
{
int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
int i,j,k,pos,temp;
for(i=0;i<10;i++)
{
pos = smallest(a,10,i);
temp = a[i];
a[i]=a[pos];
a[pos] = temp;
}
printf("\nprinting sorted elements...\n");
for(i=0;i<10;i++)
{
printf("%d\n",a[i]);
}
}
int smallest(int a[], int n, int i)
{
int small,pos,j;
small = a[i];
pos = i;
for(j=i+1;j<10;j++)
{
if(a[j]<small)
{
small = a[j];
pos=j;
}
}
return pos;
}
```

Output:

```
printing sorted elements...
7
9
10
12
23
```

23
34
44
78
101

INSERTION SORT

Insertion sort is one of the best sorting techniques. It is twice as fast as Bubble sort. In Insertion sort the elements comparisons are as less as compared to bubble sort. In this comparison the value until all prior elements are less than the compared values is not found. This means that all the previous values are lesser than compared value. Insertion sort is good choice for small values and for nearly sorted values.

Working of Insertion sort:

The Insertion sort algorithm selects each element and inserts it at its proper position in a sub list sorted earlier. In a first pass the elements A_1 is compared with A_0 and if A[1] and A[0] are not sorted they are swapped.

In the second pass the element[2] is compared with A[0] and A[1]. And it is inserted at its proper position in the sorted sub list containing the elements A[0] and A[1]. Similarly doing i_{th} iteration the element A[i] is placed at its proper position in the sorted sub list, containing the elements A[0],A[1],A[2],…………A[i-1].

To understand the insertion sort consider the unsorted Array A={7,33,20,11,6}.

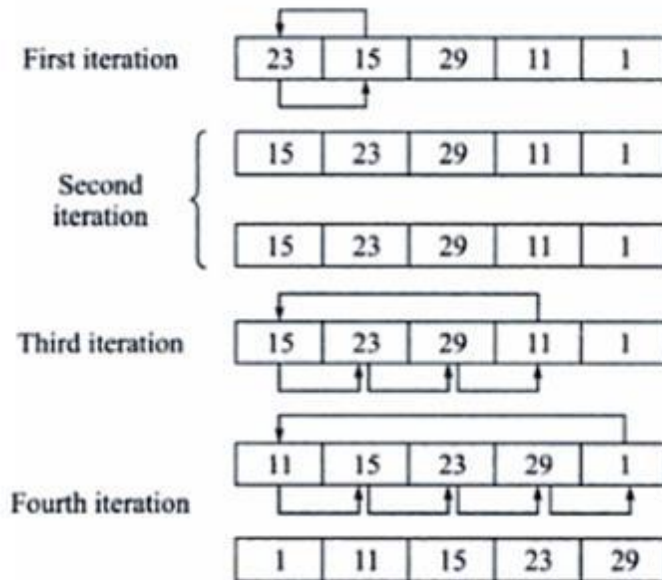
Algorithm for insertion sort

INSERTION-SORT (ARR, N)

Step 1: Repeat Steps 2 to 5 for K = 1 to N – 1
Step 2: SET TEMP = ARR[K]
Step 3: SET J = K - 1
Step 4: Repeat while TEMP <= ARR[J]
SET ARR[J + 1] = ARR[J]
SET J = J - 1
[END OF INNER LOOP]
Step 5: SET ARR[J + 1] = TEMP
[END OF LOOP]
Step 6: EXIT

Example 1:

Consider an array of integers given below. We will sort the values in the
Array using insertion sort
23 15 29 11 1



Example2:

The steps to sort the values stored in the array in ascending order using Insertion sort are given below:

7	33	20	11	6

Step 1: The first value i.e; 7 is trivially sorted by itself.

Step 2: the second value 33 is compared with the first value 7. Since 33 is greater than 7, so no changes are made.

Step 3: Next the third element 20 is compared with its previous element (towards left).Here 20 is less than 33.but 20 is greater than 7. So it is inserted at second position. For this 33 is shifted towards right and 20 is placed at its appropriate position.

7	33	20	11	6

7	20	33	11	6

Step 4: Then the fourth element 11 is compared with its previous elements. Since 11 is less than 33 and 20 ; and greater than 7. So it is placed in between 7 and 20. For this the elements 20 and 33 are shifted one position towards the right.

7	20	33	11	6

7	11	20	33	6

Step5: Finally the last element 6 is compared with all the elements preceding it. Since it is smaller than all other elements, so they are shifted one position towards right and 6 is inserted at the first position in the array. After this pass, the Array is sorted.

7	11	20	33	6

6	7	11	20	33

Step 6: Finally the sorted Array is as follows:

6	7	11	20	33

Advantages of Insertion Sort:

- It is simple sorting algorithm, in which the elements are sorted by considering one item at a time. The implementation is simple.
- It is efficient for smaller data set and for data set that has been substantially sorted before.
- It does not change the relative order of elements with equal keys
- It reduces unnecessary travels through the array
- It requires constant amount of extra memory space.

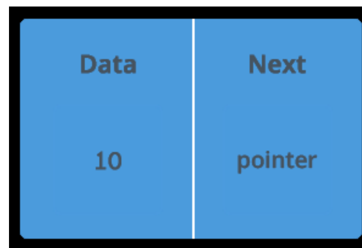
Disadvantages:-

- It is less efficient on list containing more number of elements.
- As the number of elements increases the performance of program would be slow .

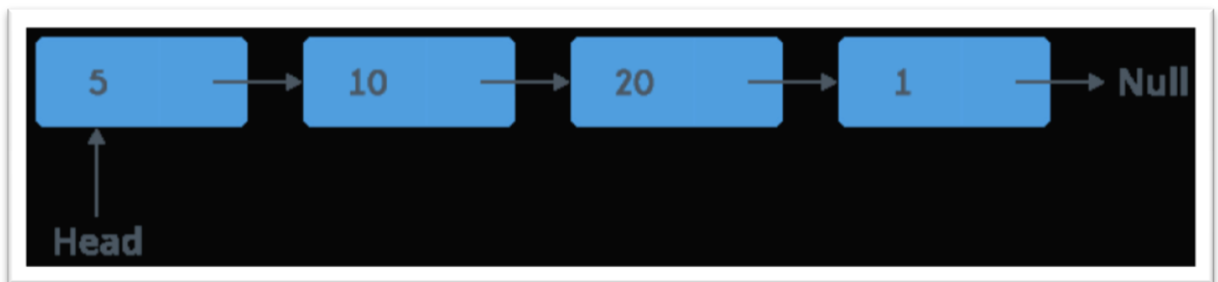
UNIT II
LINKED LISTS

1.1. Linked lists

A **linked list** is a way to store a collection of elements. Each element in a linked list is stored in the form of a **node**. A **data** part stores the element and a **next** part stores the link to the next node.



Linked List:



Advantages of linked lists:

Linked lists have many advantages. Some of the very important advantages are:

1. Linked lists are dynamic data structures. i.e., they can grow or shrink during the execution of a program.
2. Linked lists have efficient memory utilization.
3. Insertion and Deletions are easier and efficient. Linked lists provide flexibility in inserting a data item at a specified position and deletion of the data item from the given position.
4. Many complex applications can be easily carried out with linked lists.

Disadvantages of linked lists:

1. It consumes more space because every node requires a additional pointer to store address of the next node.
2. Searching a particular element in list is difficult and also time consuming.

Types of Linked list:

- Single linked list
- double linked list
- circular linked list
- double circular linked lists

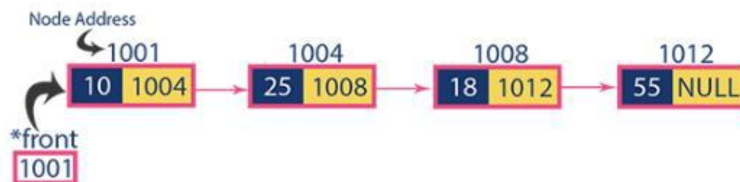
Single linked list:

Single linked list is a sequence of elements in which every element has link to its next element in the sequence.

In any single linked list, the individual element is called as **"Node"**. Every **"Node"** contains two fields, **data** and **next**. The **data** field is used to store actual value of that node and next field is used to store the address of the next node in the sequence.

The graphical representation of a node in a single linked list is as follows...

Example



Operations on single linked list:

In a single linked list we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Traverse
5. Searching

Creation of a node:
Step 1: Include all the **header files** and user defined functions.
Step 2: Define a **Node** structure with two members **data** and **next**
Step 3: Define a Node pointer '**head**' and set it to **NULL**.
Step 4: Implement the **main** method by displaying operations menu

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

Insertion:

In a single linked list, the insertion operation can be performed in three ways. They are as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

Step 1: Start
Step 2: Create a newnode with given value.
Step 3: Check whether list is Empty (head == NULL)
Step 4: If it is **Empty**:
  set newNode→next = NULL
  set head = newNode.
Step 5: If it is **Not Empty:**
  set newNode→next = head
  set head = newNode.
Step 6: Stop

Inserting At End of the list

Step 1: Start
Step 2: Create a newnode with given value.
Step 3: Check whether list is Empty (head == NULL)
Step 4: If it is **Empty**:
  set newNode→next = NULL
  set head = newNode.
Step 5**:** If it is **Not Empty** then define a node pointer **temp**
  temp = head (initialize temp with **head**).
Step 6**:** move **temp** to its next node until it reaches to the last node in the list
  (until **temp → next** = **NULL**).
Step 7**:** Set **temp → next** = **newNode**.

Step 8: Stop

Inserting At Specific location in the list (After a Node)

Step 1: Start

Step 2: Create a newnode with given value.

Step 3: Check whether list is Empty (head == NULL)

Step 4: If it is **Empty**:

set newNode→next = NULL

set head = newNode.

Step 5: If it is **Not Empty**, then define a node pointer **temp**

temp = head (initialize temp with **head**).

Step 6: move **temp** to its next node until it reaches specific location to insert

(until **temp → data** = **location**).

Step 7: Set **newNode → next** = **temp → next**

Set temp → next = **newNode**

Step 8: Stop

Deletion: In a single linked list, the deletion operation can be performed in three ways. They are as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

Step 1: Start

Step 2: Check whether list is **Empty** (**head** == **NULL**)

Step 3: If it is **Empty**:

display **'List is Empty!!! Deletion is not possible'**.

Step 4: If it is Not Empty then define a Node pointer 'temp'

Set temp = head (initialize temp with **head**).

Step 5: Set head = **temp → next**

delete **temp**.

free (temp)

Deleting from End of the list

Step 1: start

Step 2: Check whether list is **Empty** (**head** == **NULL**)

Step 3: If it is **Empty**:

display **'List is Empty!!! Deletion is not possible'**

Step 4: If it is **Not Empty** then define two Node pointers **'temp1'** and '**temp2'**

Set temp1 = head (initialize '**temp1**' with **head**).

Step 5: set '**temp2 = temp1** ' and move **temp1** to its next node.

Step 6: Repeat the same until it reaches to the last node in the list.

(until **temp1 → next** == **NULL**)

Step 7: Finally, Set **temp2 → next** = **NULL**
delete **temp1**.
free (temp1).

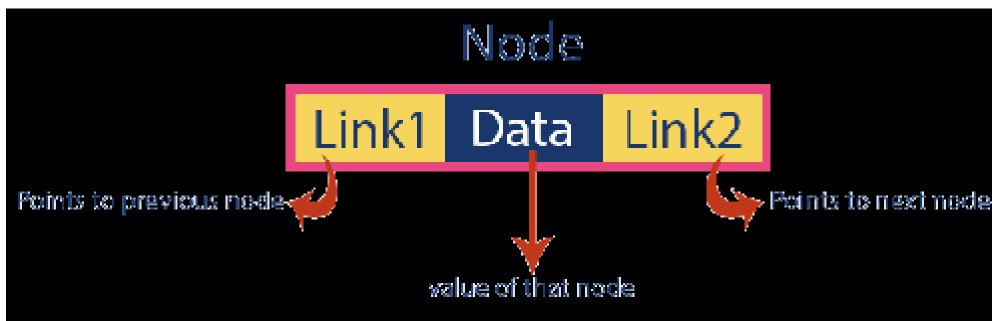
Deleting a Specific Node from the list
Step 1: start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is **Empty**:
display **'List is Empty!!! Deletion is not possible'**
Step 4**:** If it is **Not Empty**, then define two Node pointers **'temp1'** and '**temp2'**
Set temp1 = head (initialize '**temp1**' with **head**).
Step 5: set '**temp2 = temp1** ' and move **temp1** to its next node.
Step 6: Repeat the same until it reaches specific node which we want to delete.
Step 7: set **temp2 → next** = **temp1 → next**
delete **temp1**
free(temp1)

Traverse
Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is Empty:
display List is Empty!!!
Step 4: If it is Not Empty, then define a Node pointer 'temp'
Set temp = head (initialize temp with head).
Step 5: Keep displaying temp-->□data until temp reaches last node
temp = temp-->next
Step 6: Stop

Searching
Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is **Empty**:
display **List is Empty. Searching is not possible.**
Step 4: If it is **Not Empty:** define a Node pointer **'temp'**
Set temp = head (initialize temp with **head**).
Step 5: Enter item to search i.e., key
Step 6: move temp until it reaches key.
temp = temp →next
Step 7: if(temp →data = key) then
print “search is successful”
else
print “search is unsuccessful”

1.8 Double linked list:

In double linked list, every node has link to its previous node and next node. So, we can traverse forward by using next field and can traverse backward by using previous field. Every node in a double linked list contains three fields and they are shown in the following figure...



- In double linked list, the first node must be always pointed by head.
- Always the previous field of the first node must be NULL.
- Always the next field of the last node must be NULL.

In a Double linked list we perform the following operations...

1. Creation
2. Insertion
3. Deletion
4. Traverse
5. Searching

Creation of a node:

Step 1: Include all the **header files** and user defined functions.
Step 2: Define a **Node** structure with two members **data** and **next**
Step 3: Define a Node pointer '**head**' and set it to **NULL**
Step 4: Implement the **main** method by displaying operations menu

```
struct node
{
int data;
struct node *prev, *next;
};
```

Insertion

In a double linked list, the insertion operation can be performed in three ways as follows...

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

Inserting At Beginning of the list

Step 1: Start
Step 2: Create a **newNode** with given value
Step 3: Check whether list is **Empty** (**head** == **NULL**)
Step 4: If it is **Empty** then
Set newnode→prev =null
Set newNode → next = null

Set newnode→data = value
Set head = newnode
Step 5: If it is **not Empty** then
Set newNode → next = head
Set head→prev = newNode
Set head = newnode

Inserting At Specific location in the list

Step 1: Start
Step 2: Create a **newNode** with given value
Step 3: Check whether list is **Empty** (**head** == **NULL**)
Step 4: If it is **Empty** then
Set newnode→prev =null
Set newNode → next = null
Set newnode→data = value
Set head = newnode
Step 5: If it is **not Empty**: (define a node pointer temp)
Set temp = head (initialize temp with head)
Step 6**:** move **temp** to its next node until it reaches specific location to insert
(until **temp** → **data** = **location**).
Step 7: Set newnode→next = temp→next
Set newnode→prev=temp
Set (temp→next)→prev = newnode
Set temp →next = newnode

Inserting At End of the list

Step 1: Start
Step 2: Create a **newNode** with given value
Step 3: Check whether list is **Empty** (**head** == **NULL**)
Step 4: If it is **Empty** then
Set newnode→prev =null
Set newNode → next = null
Set newnode→data = value
Set head = newnode
Step 5: If it is **not Empty**: (define a node pointer temp)
Set temp = head (initialize temp with head)
Step 6**:** move **temp** to its next node until it reaches last node to insert.
(until temp →next = null)
Step 7: Set temp-->ext = newnode
Set newnode→rev = temp
Set newnode →next =null

Deletion:

In a double linked list, the deletion operation can be performed in three ways as follows...

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting at Specific location

Deleting from Beginning of the list

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then
display **'List is Empty!!! Deletion is not possible'**.
Step 3: If it is not Empty: then define a Node pointer **'temp'**
Set temp = head (initialize temp with **head)**.
Step 4: Set head = temp→next
Set head→prev = null
Set temp →next = null
delete temp
free (temp)

Deleting at Specific location

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then
display **'List is Empty!!! Deletion is not possible'**.
Step 3: If it is not Empty, then define a Node pointer **'temp'**
Set temp =head (initialize temp with head**)**.
Step 4: Keep moving the **temp** until it reaches specific node to delete.
Step 5: Set (temp→prev)→next = temp→next
Set (temp→next)→prev = temp→prev
delete temp
free(temp)

Deleting from End of the list

Step 1: Check whether list is **Empty** (**head** == **NULL**)
Step 2: If it is **Empty** then
display **'List is Empty!!! Deletion is not possible'**.
Step 3: If it is **not Empty**, then define a Node pointer **'temp'**
Set temp =head (initialize temp with head).
Step 4: Keep moving the temp until it reaches last node to delete.
(until temp → next = NULL)

Step 5: set (temp → prev) → next = null
delete temp
free(temp)

Traverse (forward):

Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is Empty:
display List is Empty!!!
Step 4: If it is Not Empty, then define a Node pointer 'temp'
Set temp = head (initialize temp with head).
Step 5: Keep moving temp forward
temp = temp→next
Step 6: Keep displaying temp→data until temp reaches last node
(until temp→next=null)
Step 6: Stop

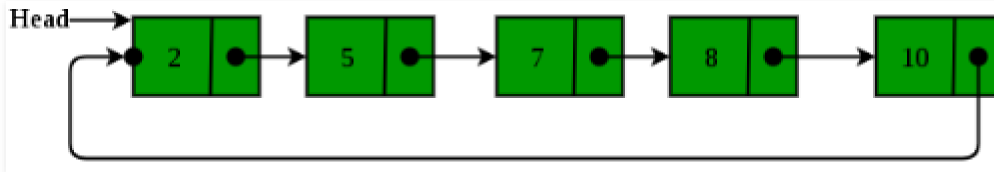
Traverse (backward):

Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is Empty:
display List is Empty!!!
Step 4: If it is Not Empty, then define a Node pointer 'temp'
Set temp = tail (initialize temp with tail).
Step 5: Keep moving temp backward
temp = temp→prev
Step 6: Keep displaying temp→data until temp reaches head node
(until temp→prev=null)
Step 7: Stop

Searching:

Step 1: Start
Step 2: Check whether list is **Empty** (**head** == **NULL**)
Step 3: If it is **Empty**:
display "**List is Empty. Searching is not possible".**
Step 4: If it is **Not Empty** then define a Node pointer **'temp'**
Set temp = head (initialize temp with **head**).
Step 5: Enter item to search i.e., key
Step 6: move temp until it reaches key.
temp = temp →next
Step 7: if(temp →data = key) then
print "search is successful"
else
print "search is unsuccessful"

1.9 Circular linked list:
Circular linked list is a sequence of elements in which every element has link to its next element in the sequence and the last element has a link to the first element in the sequence.



Operations:

- Insertion
- Deletion
- Traverse
- Searching

Insertion (begin)
Step 1: Start
Step 2: Create a new node with a given value
Step 3: Check whether list is Empty **(head == NULL)**
Step 4: If list is empty then
    Set Newnode→data = value
    Set Newnode→next=newnode
    Set head = newnode
    Set tail = newnode
Step 5: If list is non-empty then
    Set newnode→next=head
    Set head=newnode
    Set tail→next=newnode
Step 6: Stop
Insertion (End)
Step 1: Start
Step 2: Create a new node with a given value
Step 3: Check whether list is Empty (**head == NULL**)
Step 4: If list is empty:
    Set Newnode→data = value
    Set Newnode→next=newnode
    Set head = newnode
    Set tail = newnode
Step 5: If list is non-empty:
    Set tail→next=newnode
    Set tail=newnode

Set tail→next=head

Step 6: Stop

Insertion (Specific location)

Step 1: Start

Step 2: Create a new node with a given value

Step 3: Check whether list is Empty (**head == NULL**)

Step 4: If list is empty:

  Set Newnode→data = value

  Set Newnode→next=newnode

  Set head = newnode

  Set tail = newnode

Step 5: If list is not empty: Define pointer temp.

  Set **temp = head** (initialize temp with head)

Step 6: move **temp** to its next node until it reaches the location to insert new node

  Set temp=temp→next

  Set temp→data=location

Step 7: when location is reached

  Set newnode→next=temp→next

  Set temp→next=newnode

Step 8: Stop

Deletion (begin)

Step 1: Start

Step 2: Check whether list is Empty **(head == NULL)**

Step 3: If it is Empty:

  Display "**List is Empty. Deletion is not possible**"

Step 4: If it is Not Empty: Define pointer temp.

  Set **temp = head** (initialize temp with head)

Step 5**: Set head=temp-->ext**

  Set tail→next=head

Step 6**: Delete temp**

  free (temp)

Step 7: Stop

Deletion (end)

Step 1: Start

Step 2: Check whether list is Empty **(head == NULL)**

Step 3: If it is Empty:

  Display "**List is Empty. Deletion is not possible**"

Step 4: If it is Not Empty: define pointers 'temp1' and 'temp2'

  temp1 = head (initialize temp1 with head).

Step 5: set **temp2 = temp1** and move temp1 to its next node

Step 6: Repeat the same until **temp1 → next == head**

Step 7: **set temp2→next=head**

Step 8: **delete temp1**

  free (temp1)

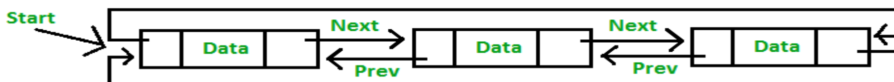
Deletion (specific location)

Step 1: Start

Step 2: Check whether list is Empty **(head == NULL)**
Step 3: If it is Empty:
Display "**List is Empty. Deletion is not possible**"
Step 4: If it is Not Empty: define pointers 'temp1' and 'temp2'
temp1 = **head** (initialize temp1 with head).
Step 5: set **temp2** = **temp1** and move temp1 to its next node
Step 6: Repeat the same until temp1 reaches the node to delete at specific position in the list
Step 7: **Set temp2→next = temp1→next**
Step 8: **delete temp1**
free (temp1)
Step 9: stop.

1.10 .Circular Double linked list:

Circular Doubly Linked List has properties of both doubly linked list and circular linked list in which two consecutive elements are linked or connected by previous and next pointer and the last node points to first node by next pointer and also the first node points to last node by previous pointer.



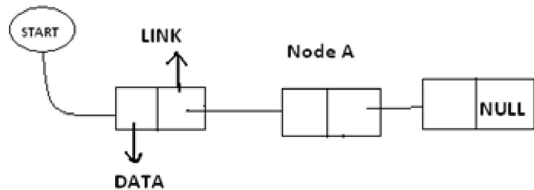
Difference between Arrays and Linked List?

Arrays	**Linked List**
1. Arrays are used in the predictable storage requirement ie; exert amount of data storage required by the program can be determined. 2. In arrays the operations such as insertion and deletion are done in an inefficient manner. 3. The insertion and deletion are done by moving the elements either up or down. 4. Successive elements occupy adjacent space on memory. 5. In arrays each location contain DATA only 6. The linear relation ship between the data elements of an array is reflected by the physical relation ship of data in the memory. 7. In array declaration a block of memory space is required. 8.There is no need of storage of pointer or lines 9.The Conceptual view of an Array is as follows:	1. Linked List are used in the unpredictable storage requirement ie; exert amount of data storage required by the program can't be determined. 2. In Linked List the operations such as insertion and deletion are done more efficient manner ie; only by changing the pointer. 3. The insertion and deletion are done by only changing the pointers. 4. Successive elements need not occupy adjacent space. 5. In linked list each location contains data and pointer to denote whether the next element present in the memory. 6. The linear relation ship between the data elements of a Linked List is reflected by the Linked field of the node. 7. In Linked list there is no need of such thing. 8. In Linked list a pointer is stored along into the element. 9. The Conceptual view of Linked list is as

<table>
<tr>
<td>

A	10	20	30	40	50
	0	1	2	3	4

10.In array there is no need for an element to specify whether the next is stored</td>
<td>follows:



10. There is need for an element (node) to specify whether the next node is formed.</td>
</tr>
</table>

Applications of linked lists:
1. Sparse matrix Representation
2. Polynomial representation
3. Dynamic storage management

Polynomial Representation

Array implementation:

•Array Implementation:
- $p1(x) = 8x^3 + 3x^2 + 2x + 6$
- $p2(x) = 23x^4 + 18x - 3$

p1(x)

6	2	3	8

p2(x)

-3	18	0	0	23

•This is why arrays aren't good to represent polynomials:

- $p3(x) = 16x^{21} - 3x^5 + 2x + 6$

6	2	0	0	-3	0	…………	0	16

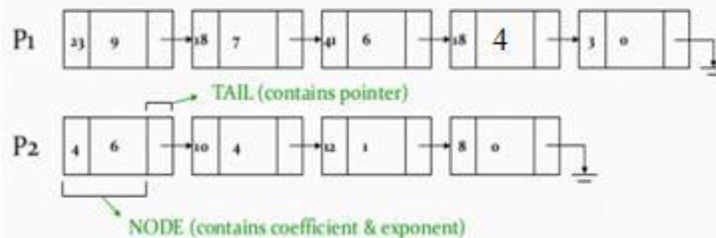
WASTE OF SPACE!

Linked list implementation:

Procedure

- Linked list Implementation:

- $p1(x) = 23x^9 + 18x^7 + 41x^6 + 18x^4 + 3$
- $p2(x) = 4x^6 + 10x^4 + 12x + 8$



Procedure to add polynomials using linked list

- Adding polynomials using a Linked list representation: (storing the result in p3)

To do this, we have to break the process down to cases:

- Case 1: exponent of p1 > exponent of p2
  - Copy node of p1 to end of p3.

[go to next node]

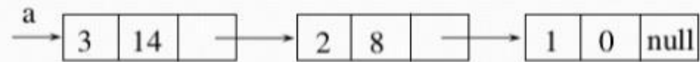
- Case 2: exponent of p1 < exponent of p2
  - Copy node of p2 to end of p3.

[go to next node]

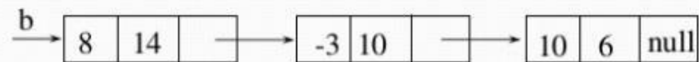
- Case 3: exponent of p1 = exponent of p2
  - Create a new node in p3 with the same exponent and with the sum of the coefficients of p1 and p2.

Example

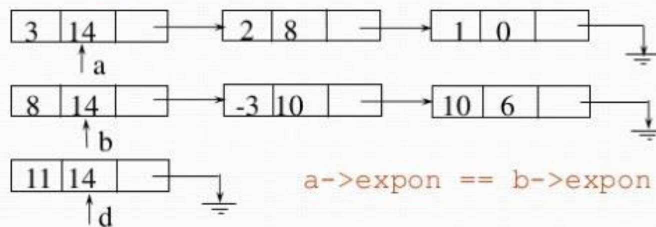
$$a = 3x^{14} + 2x^{8} + 1$$

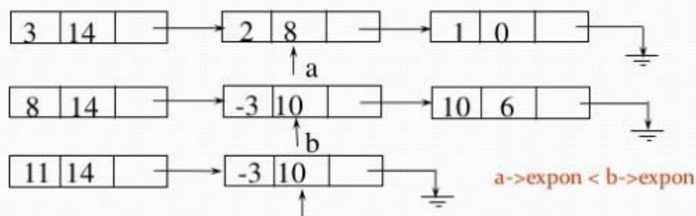


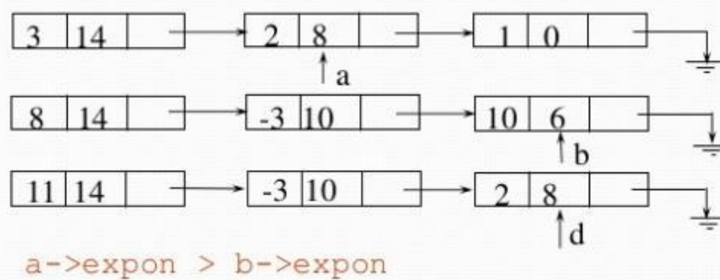
$$b = 8x^{14} - 3x^{10} + 10x^{6}$$

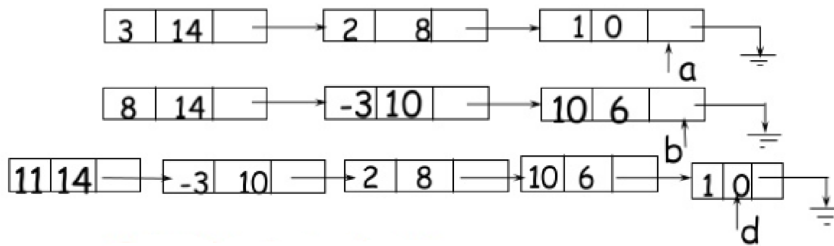


Adding Polynomials









Sparse matrix Representation

In computer programming, a matrix can be defined with a 2-dimensional array. Any array with 'm' columns and 'n' rows represents a mXn matrix. There may be a situation in which a matrix contains more number of ZERO values than NON-ZERO values. Such matrix is known as sparse matrix.

Sparse matrix is a matrix which contains very few non-zero elements.

A sparse matrix can be represented by using TWO representations, those are as follows...

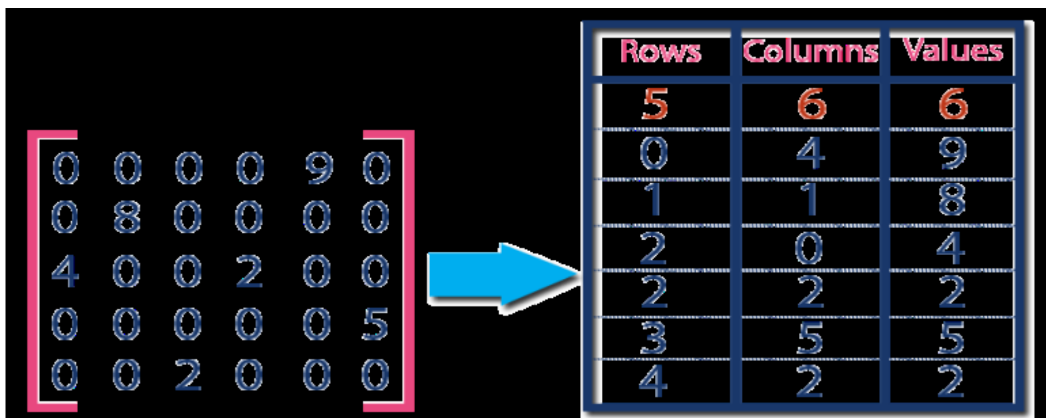
Triplet Representation

Linked Representation

Triplet Representation

In this representation, we consider only non-zero values along with their row and column index values. In this representation, the 0th row stores total rows, total columns and total non-zero values in the matrix.

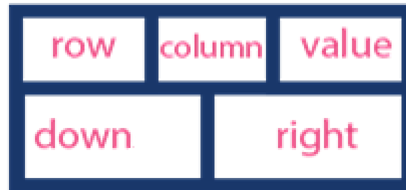
For example, consider a matrix of size 5 X 6 containing 6 number of non-zero values. This matrix can be represented as shown in the image...



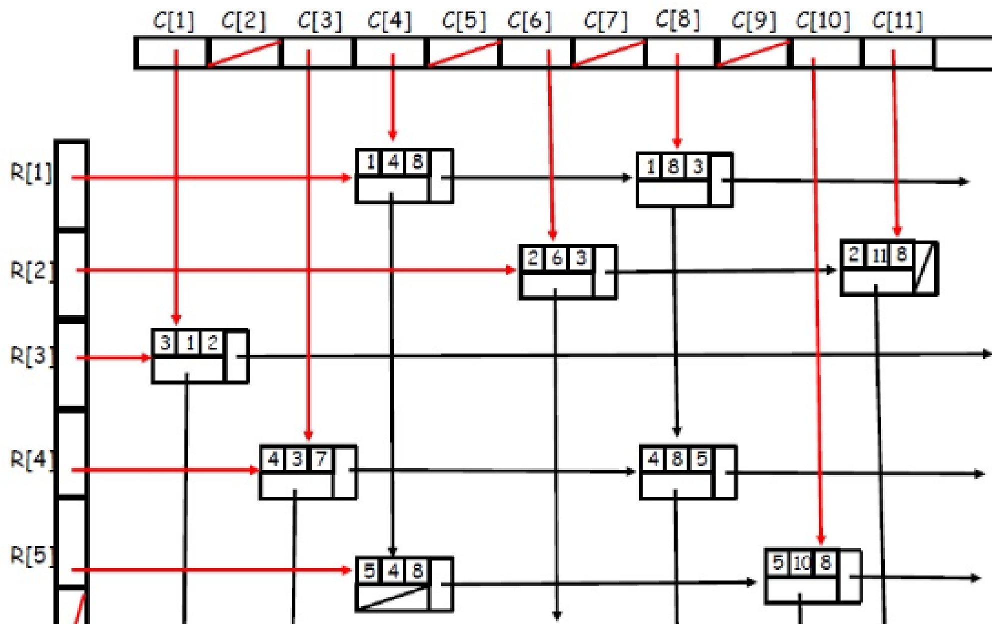
Rows	Columns	Values
5	6	6
0	4	9
1	1	8
2	0	4
2	2	2
3	5	5
4	2	2

Linked Representation

In linked representation, we use linked list data structure to represent a sparse matrix. In this linked list, we use two different nodes namely header node and element node. Header node consists of three fields and element node consists of five fields as shown in the image...



Example: There are two arrays **of pointers** that are the row array and column array. Each cell of the array is pointing to the respective line/column. It is as in the picture below:



Being a sparse matrix, where there are zeroes, you see those arrows as those nodes that have zeroes with them do not exist.

Dynamic memory management:
Dynamic memory management scheme is based on these principles:
- • Allocation schemes
- • Deallocation schemes

Allocation schemes: how request for a node will be serviced:
- • Fixed block allocation
- • Variable block allocation
- o First fit
- o Next fit

- o Best fit
- o Worst fit

Deallocation schemes: how to return a node to memory bank whenever it is no more required.

- • Random deallocation
- • Ordered deallocation

UNIT-III

STACKS AND QUEUES

STACKS

A Stack is linear data structure. A stack is a list of elements in which an element may be inserted or deleted only at one end, called the **top of the stack**. Stack principle is **LIFO (last in, first out)**. Which element inserted last on to the stack that element deleted first from the stack. As the items can be added or removed only from the top i.e. the last item to be added to a stack is the first item to be removed.

Operations on stack:

The two basic operations associated with stacks are:

1. Push
2. Pop

While performing push and pop operations the following test must be conducted on the stack.

a) Stack is empty or not b) stack is full or not

1. Push: Push operation is used to add new elements in to the stack. At the time of addition first check the stack is full or not. If the stack is full it generates an error message "stack overflow".

2. Pop: Pop operation is used to delete elements from the stack. At the time of deletion first check the stack is empty or not. If the stack is empty it generates an error message "stack underflow".

All insertions and deletions take place at the same end, so the last element added to the stack will be the first element removed from the stack. When a stack is created, the stack base remains fixed while the stack top changes as elements are added and removed. The most accessible element is the top and the least accessible element is the bottom of the stack.

Representation of Stack (or) Implementation of stack:

The stack should be represented in two ways:

1. Stack using array
2. Stack using linked list

1. Stack using array:

Let us consider a stack with 6 elements capacity. This is called as the size of the stack. The number of elements to be added should not exceed the maximum size of the stack. If we attempt to add new element beyond the maximum size, we will encounter a ***stack overflow*** condition. Similarly, you cannot remove elements beyond the base of the stack. If such is the case, we will reach a ***stack underflow*** condition.

1.push():When an element is added to a stack, the operation is performed by push(). Below Figure shows the creation of a stack and addition of elements using push().

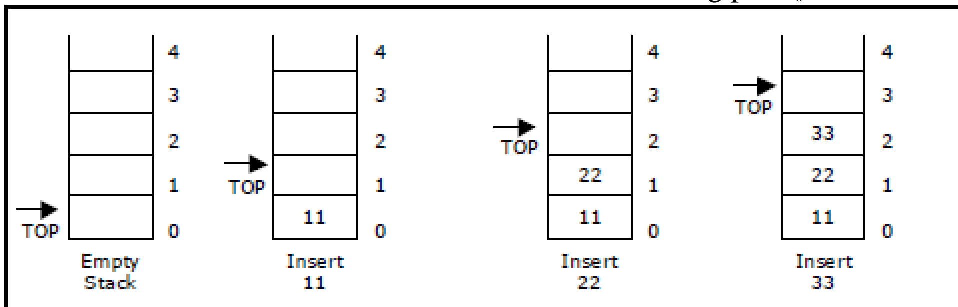


Figure . Push operations on stack

Initially top=-1, we can insert an element in to the stack, increment the top value i.e top=top+1. We can insert an element in to the stack first check the condition is stack is full or not. i.e top>=size-1. Otherwise add the element in to the stack.

Code	Algorithm

void push()
{
int x;
if(top >= n-1)
{
printf("\n\nStack Overflow..");
return;
}
else
{
printf("\n\nEnter data: ");
scanf("%d", &x);
stack[top] = x;
top = top + 1;
printf("\n\nData Pushed into the stack");
}
}
	Algorithm: Procedure for push():
	Step 1: START
	Step 2: if top>=size-1 then
	Write " Stack is Overflow"
	Step 3: Otherwise
	3.1: read data value 'x'
	3.2: top=top+1;
	3.3: stack[top]=x;
	Step 4: END

2.Pop(): When an element is taken off from the stack, the operation is performed by pop(). Below figure shows a stack initially with three elements and shows the deletion of elements using pop().

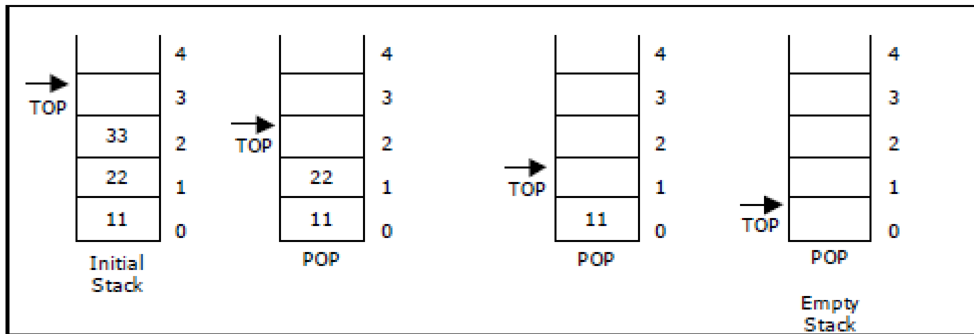
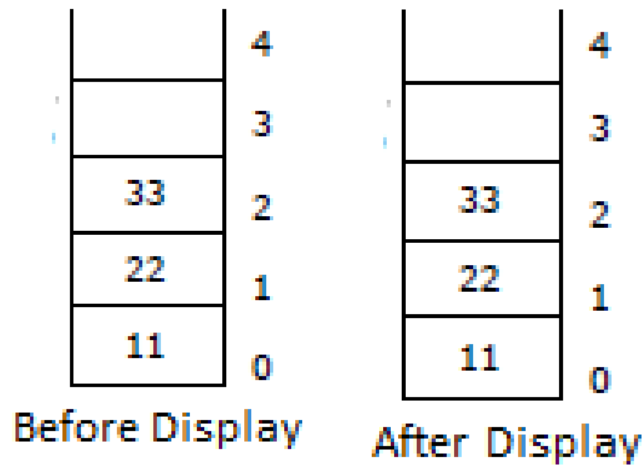


Figure Pop operations on stack

We can insert an element from the stack, decrement the top value i.e top=top-1. We can delete an element from the stack first check the condition is stack is empty or not. i.e top==-1. Otherwise remove the element from the stack.

Void pop()
{
If(top==-1)
{
Printf("Stack is Underflow");
}
else
{
printf("Delete data %d",stack[top]);
top=top-1;
}
}
	Algorithm: procedure pop():
	Step 1: START
	Step 2: if top==-1 then
	Write "Stack is Underflow"
	Step 3: otherwise
	3.1: print "deleted element"
	3.2: top=top-1;
	Step 4: END

3.display(): This operation performed display the elements in the stack. We display the element in the stack check the condition is stack is empty or not i.e top==-1.Otherwise display the list of elements in the stack.



void display()	**Algorithm: procedure pop():**

{
If(top==-1)
{
Printf(“Stack is Underflow”);
}
else
{
printf(“Display elements are:);
for(i=top;i>=0;i--)
printf(“%d”,stack[i]);
}
}
	Step 1: START
	Step 2: if top==-1 then
	Write “Stack is Underflow”
	Step 3: otherwise
	3.1: print “Display elements are”
	3.2: for top to 0
	Print ‘stack[i]’
	Step 4: END

Source code for stack operations, using array:

```
#include<stdio.h>
#inlcude<conio.h>
int stack[100],choice,n,top,x,i;
void push(void);
void pop(void);
void display(void);
int main()
{
//clrscr();
top=-1;
printf("\n Enter the size of STACK[MAX=100]:");
scanf("%d",&n);
printf("\n\t STACK OPERATIONS USING ARRAY");
printf("\n\t--------------------------------");
```

```
printf("\n\t 1.PUSH\n\t 2.POP\n\t 3.DISPLAY\n\t 4.EXIT");
do
{
printf("\n Enter the Choice:");
scanf("%d",&choice);
switch(choice)
{
case 1:
{
push();
break;
}
case 2:
{
pop();
break;
}
case 3:
{
display();
break;
}
case 4:
{
printf("\n\t EXIT POINT ");
break;
}
default:
{
printf ("\n\t Please Enter a Valid Choice(1/2/3/4)");
}
}
}
while(choice!=4);
return 0;
}
void push()
{
if(top>=n-1)
{
printf("\n\tSTACK is over flow");
}
else
{
printf(" Enter a value to be pushed:");
scanf("%d",&x);
```

```
top++;
stack[top]=x;
}
}
void pop()
{
if(top<=-1)
{
printf("\n\t Stack is under flow");
}
else
{
printf("\n\t The popped elements is %d",stack[top]);
top--;
}
}
void display()
{
if(top>=0)
{
 printf("\n The elements in STACK \n");
for(i=top; i>=0; i--)
printf("\n%d",stack[i]);
printf("\n Press Next Choice");
}
else
{
printf("\n The STACK is empty");
}
}
```

2. Stack using Linked List:

We have seen how a stack is created using an array. This technique of creating a stack is easy, but the drawback is that the array must be declared to have some fixed size. In case the stack is a very small one or its maximum size is known in advance, then the array implementation of the stack gives an efficient implementation. But if the array size cannot be determined in advance, then the other alternative, i.e., linked representation, is used.

The storage requirement of linked representation of the stack with nelements is O(n), and the typical time requirement for the operations is O(1).

In a linked stack, every node has two parts—one that stores data and another that stores the address of the next node. The START pointer of the linked list is used as TOP. All insertions and deletions are done at the node pointed by TOP. If TOP = NULL, then it indicates that the stack is empty.

The linked representation of a stack is shown in Fig. 7.13.

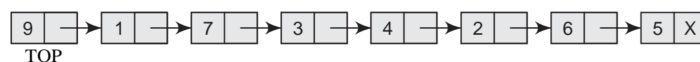


Figure 7.13 Linked stack

OPERATIONS ON A LINKED STACK

A linked stack supports all the three stack operations, that is, push, pop, and peek.

Push Operation

The push operation is used to insert an element into the stack. The new element is added at the topmost position of the stack. Consider the linked stack shown in Fig. 7.14.

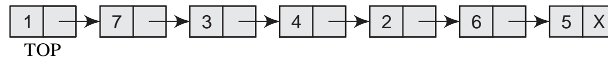


Figure 7.14 Linked stack

To insert an element with value 9, we first check if TOP=NULL. If this is the case, then we allocatememory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called TOP. However, if TOP!=NULL, then we insert the new node at the beginning of the linked stack and name this new node as TOP. Thus, the updated stack becomes as shown in Fig. 7.15.

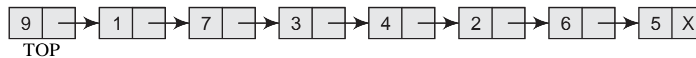


Figure 7.15 Linked stack after inserting a new node

Figure 7.16 shows the algorithm to push an element into a linked stack. In Step 1, memory is allocated for the new node. In Step 2, the DATApart of the new node is initialized with the value tobe stored in the node. In Step 3, we check if the new node is the first node of the linked list.

```
Step 1: Allocate memory for the new node and name it as NEW_NODE

Step 2: SET NEW_NODE -> DATA = VALStep 3: IF TOP = NULL

            SET NEW_NODE -> NEXT = NULLSET TOP = NEW_NODE

        ELSE

            SET NEW_NODE -> NEXT = TOPSET TOP = NEW_NODE

        [END OF IF]

Step 4: END
```

Figure 7.16 Algorithm to insert an element in alinked stack

This is done by checking if TOP = NULL. In case the IFstatementevaluates to true, then NULLis stored in the NEXTpart of thenode and the new node is called TOP. However, if the newnode is not the first node in the list, then it is added beforethe first node of the list (that is, the TOP node) and termedas TOP.

7.1.1 Pop Operation

The pop operation is used to delete the topmost element from a stack. However, before deleting the value, we must first check if TOP=NULL, because if this is the case, then it means that the stack is empty and no more deletions can be done. If an attempt is made to delete a value from a stack that is already empty, an UNDERFLOW message is printed. Consider the stack shown in Fig. 7.17.

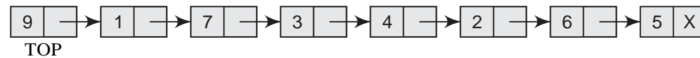


Figure 7.17 Linked stack

In case TOP!=NULL, then we will delete the node pointed by TOP, and make TOP point to the second element of the linked stack. Thus, the updated stack becomes as shown in Fig. 7.18.

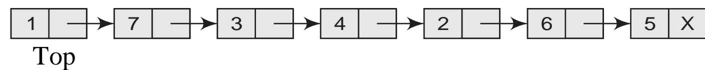
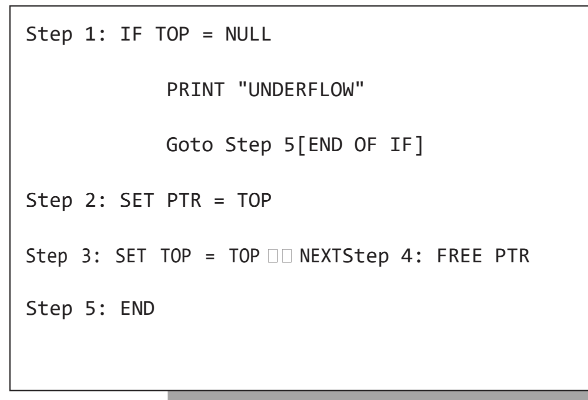


Figure 7.18 Linked stack after deletion of the topmost element



```
Step 1: IF TOP = NULL

            PRINT "UNDERFLOW"

            Goto Step 5[END OF IF]

Step 2: SET PTR = TOP

Step 3: SET TOP = TOP □□ NEXTStep 4: FREE PTR

Step 5: END
```

Figure 7.19 Algorithm to delete an element from a linked stack

Figure 7.19 shows the algorithm to delete an element from a stack. In Step 1, we first check for the UNDERFLOW condition. In Step 2, we use a pointer PTR that points to TOP. In Step 3, TOP is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

Applications of stack:

1. Stack is used by compilers to check for balancing of parentheses, brackets and braces.
2. Stack is used to evaluate a postfix expression.
3. Stack is used to convert an infix expression into postfix/prefix form.
4. In recursion, all intermediate arguments and return values are stored on the processor's stack.
5. During a function call the return address and arguments are pushed onto a stack and on return they are popped off.

QUEUE:

A queue is linear data structure and collection of elements. A queue is another special kind of list, where items are inserted at one end called the **rear** and deleted at the other end called the **front**. The principle of queue is a **"FIFO" or "First-in-first-out"**.

Queue is an abstract data structure. A queue is a useful data structure in programming. **It is similar to the ticket queue outside a cinema hall**, where the first person entering the queue is the first person who gets the ticket.

A real-world example of queue can be a single-lane one-way road, where the vehicle enters first, exits first.

More real-world examples can be seen as queues at the ticket windows and bus-stops and our college library.

The operations for a queue are analogues to those for a stack; the difference is that the insertions go at the end of the list, rather than the beginning.

Operations on QUEUE:

A queue is an object or more specifically an abstract data structure (ADT) that allows the following operations:

- **Enqueue or insertion:** which inserts an element at the end of the queue.
- **Dequeue or deletion:** which deletes an element at the start of the queue.

Queue operations work as follows:

1. Two pointers called FRONT and REAR are used to keep track of the first and last elements in the queue.
2. When initializing the queue, we set the value of FRONT and REAR to 0.
3. On enqueing an element, we increase the value of REAR index and place the new element in the position pointed to by REAR.
4. On dequeueing an element, we return the value pointed to by FRONT and increase the FRONT index.
5. Before enqueing, we check if queue is already full.
6. Before dequeuing, we check if queue is already empty.
7. When enqueing the first element, we set the value of FRONT to 1.
8. When dequeing the last element, we reset the values of FRONT and REAR to 0.

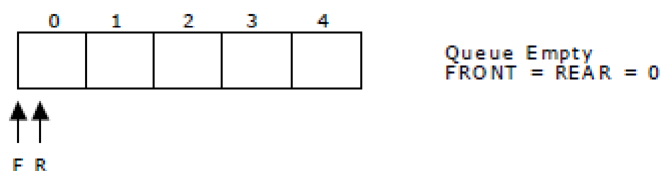
Representation of Queue (or) Implementation of Queue:

The queue can be represented in two ways:

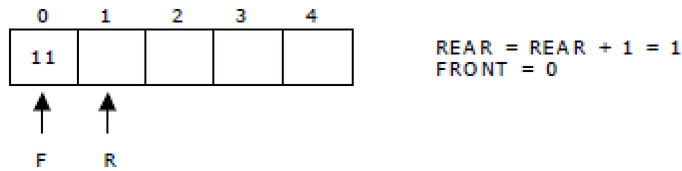
1. Queue using Array
2. Queue using Linked List

1.Queue using Array:

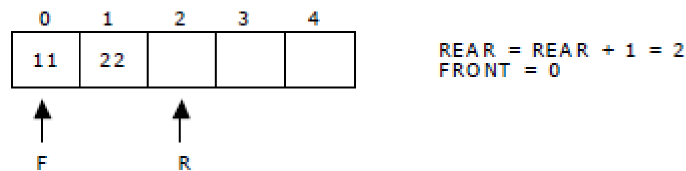
Let us consider a queue, which can hold maximum of five elements. Initially the queue is empty.



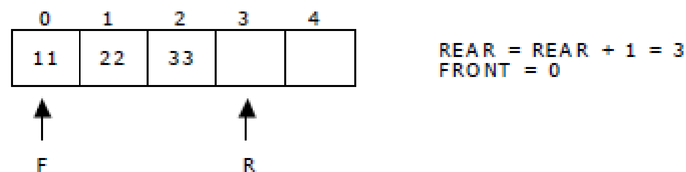
Now, insert 11 to the queue. Then queue status will be:



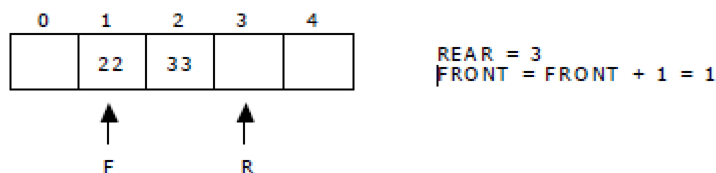
Next, insert 22 to the queue. Then the queue status is:



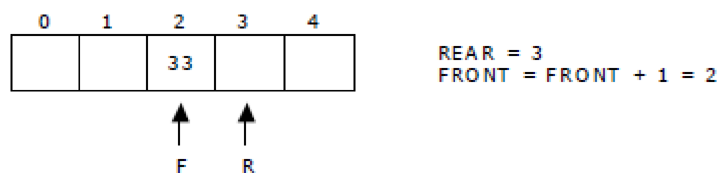
Again insert another element 33 to the queue. The status of the queue is:



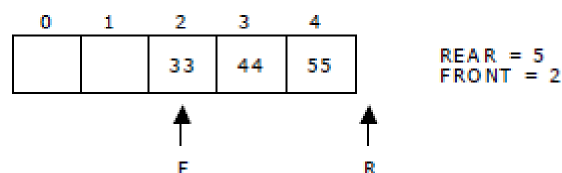
Now, delete an element. The element deleted is the element at the front of the queue.So the status of the queue is:



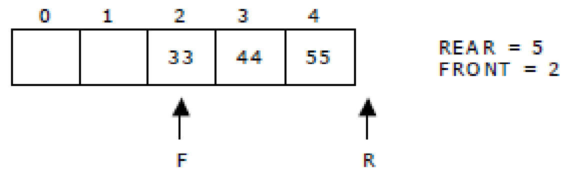
Again, delete an element. The element to be deleted is always pointed to by the FRONT pointer. So, 22 is deleted. The queue status is as follows:



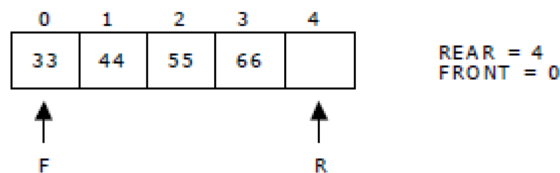
Now, insert new elements 44 and 55 into the queue. The queue status is:



Next insert another element, say 66 to the queue. We cannot insert 66 to the queue as the rear crossed the maximum size of the queue (i.e., 5). There will be queue full signal. The queue status is as follows:



Now it is not possible to insert an element 66 even though there are two vacant positions in the linear queue. To overcome this problem the elements of the queue are to be shifted towards the beginning of the queue so that it creates vacant position at the rear end. Then the FRONT and REAR are to be adjusted properly. The element 66 can be inserted at the rear end. After this operation, the queue status is as follows:



This difficulty can overcome if we treat queue position with index 0 as a position that comes after position with index 4 i.e., we treat the queue as a **circular queue.**

Queue operations using array:

a.enqueue() or insertion():which inserts an element at the end of the queue.

void insertion()
{
if(rear==max)
printf("\n Queue is Full");
else
{
printf("\n Enter no %d:",j++);
scanf("%d",&queue[rear++]);
}
}
	Algorithm: Procedure for insertion():
	Step-1:START
	Step-2: if rear==max then
	Write 'Queue is full'
	Step-3: otherwise
	3.1: read element 'queue[rear]'
	Step-4:STOP

b.dequeue() or deletion(): which deletes an element at the start of the queue.

void deletion()
{
if(front==rear)
{
printf("\n Queue is empty");
}
else
{
printf("\n Deleted Element is
%d",queue[front++]);
x++;
} }
	Algorithm: procedure for deletion():
	Step-1:START
	Step-2: if front==rear then
	Write' Queue is empty'
	Step-3: otherwise
	3.1: print deleted element
	Step-4:STOP

Queue using Linked list:

We have seen how a queue is created using an array. Although this technique of creating a queue

is easy, its drawback is that the array must be declared to have some fixed size. If we allocate space for 50 elements in the queue and it hardly uses 20–25 locations, then half of the space will

be wasted. And in case we allocate less memory locations for a queue that might end up growing large and large, then a lot of re-allocations will have to be done, thereby creating a lot of overhead and consuming a lot of time.

In case the queue is a very small one or its maximum size is known in advance, then the array implementation of the queue gives an efficient implementation. But if the array size cannot be determined in advance, the other alternative, i.e., the linked representation is used.

The storage requirement of linked representation of a queue with n elements is O(n) and the typical time requirement for operations is O(1).

In a linked queue, every element has two parts, one that stores the data and another that stores the address of the next element. The START pointer of the linked list is used as FRONT. Here, we will also use another pointer called REAR, which will store the address of the last element in the queue. All insertions will be done at the rear end and all the deletions will be done at the front end. If FRONT=REAR=NULL, then it indicates that the queue is empty.

The linked representation of a queue is shown in Fig. 8.6.

Operations on Linked Queues

A queue has two basic operations: insert and delete. The insert operation adds an element to the end of the queue, and the delete operation removes an element from the front or the start ofthe queue. Apart from this, there is another operation peek which returns the value of the first element of the queue.

Insert *Operation*

The insert operation is used to insert an element into a queue. The new element is added as the last element of the queue. Consider the linked queue shown in Fig. 8.7.

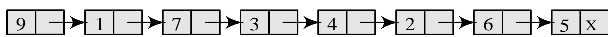


Figure 8.6 Linked queue

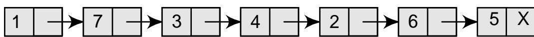
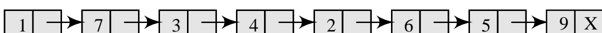


Figure 8.7 Linked queue

To insert an element with value 9, we first check if FRONT=NULL. If the condition holds, then the queue is empty. So, we allocate memory for a new node, store the value in its DATA part and NULL in its NEXT part. The new node will then be called both FRONT and REAR. However, if FRONT != NULL, then we will insert the new node at the rear end of the linked queue and name this new node as REAR. Thus, the updated queue becomes



Front Rear

Figure 8.8 Linked queue after inserting a new node

```
Step 1: Allocate memory for the new node and nameit as PTR

Step 2: SET PTR -> DATA = VALStep 3: IF FRONT =
NULL

            SET FRONT = REAR = PTR

            SET FRONT -> NEXT = REAR -> NEXT = NULL

                                    ELSE

            SET REAR -> NEXT = PTRSET REAR = PTR

            SET REAR -> NEXT = NULL[END OF IF]

Step 4: END
```

Figure 8.9 Algorithm to insert an element in a linked queue

Figure 8.9 shows the algorithm to insertan element in a linked queue. In Step 1, the memory is allocated for the new node. In Step2, the DATA part of the new node is initialized with the value to be stored in the node. In Step3, we check if the new node is the first node of the linked queue. This is done by checking if FRONT = NULL. If this is the case, then the newnode is tagged as FRONTas well as REAR. Also NULLis stored in the NEXT part of the node (which is also the FRONT and the REAR node). However, ifthe new node is not the first node in the list, thenit is added at the REAR end of the linked queue (or the last node of the queue).

Delete Operation:

The delete operation is used to delete the element that is first inserted in a queue, i.e., the element whose address is stored in FRONT. However, before deleting the value, we must first check if FRONT=NULL because if this is the case, then the queue is empty and no more deletions can be done. If an attempt is made to delete a value from a queue that is already empty, an underflow message is printed. Consider the queue shown in Fig. 8.10.

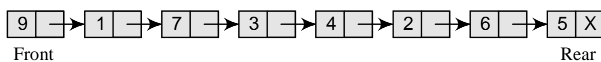


Figure 8.10 Linked queue

To delete an element, we first check if FRONT=NULL. If the condition is false, then we delete the first node pointed by FRONT. The FRONT will now point to the second element of the

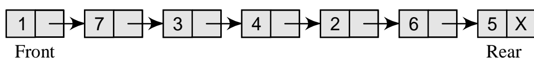
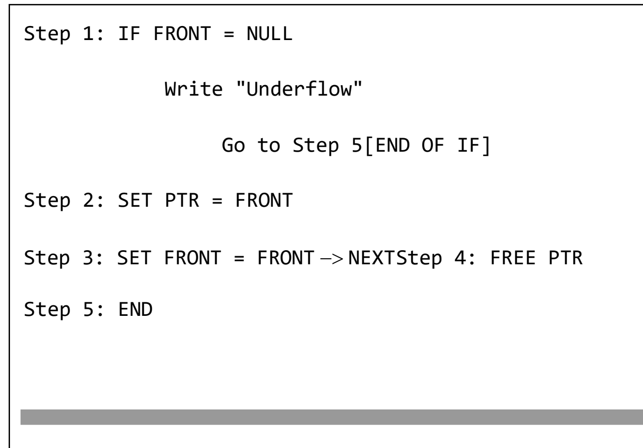


Figure 8.11 Linked queue after deletion of an element



```
Step 1: IF FRONT = NULL
            Write "Underflow"
                 Go to Step 5[END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXTStep 4: FREE PTR
Step 5: END
```

Figure 8.12 Algorithm to delete an element from a linked queue

linked queue. Thus, the updated queue becomes as shown in Fig. 8.11.

Figure 8.12 shows the algorithm to delete an element froma linked queue. In Step 1, we first check for the underflow condition. If the condition is true, then an appropriate message is displayed, otherwise in Step 2, we use a pointer PTR that points to FRONT. In Step 3, FRONT is made to point to the next node in sequence. In Step 4, the memory occupied by PTR is given back to the free pool.

Applications of Queue:

1. It is used to schedule the jobs to be processed by the CPU.
2. When multiple users send print jobs to a printer, each printing job is kept in the printing queue. Then the printer prints those jobs according to first in first out (FIFO) basis.
3. Breadth first search uses a queue data structure to find an element from a graph.

Scheduling :

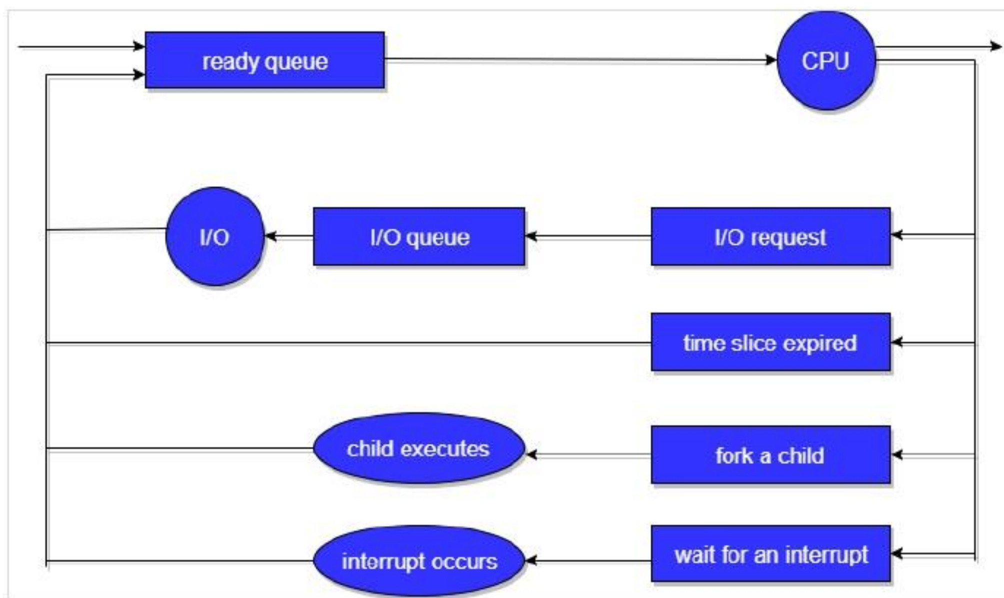
The processes that are entering into the system are stored in the **Job Queue**. Suppose if the processes are in the Ready state are generally placed in the **Ready Queue**.

The processes waiting for a device are placed in **Device Queues**. There are unique device queues which are available for every I/O device.

First place a new process in the **Ready queue** and then it waits in the ready queue till it is selected for execution.

Once the process is assigned to the CPU and is executing, any one of the following events occur −

- The process issue an I/O request, and then placed in the I/O queue.
- The process may create a new sub process and wait for termination.
- The process may be removed forcibly from the CPU, which is an interrupt, and it is put back in the ready queue.



In the first two cases, the process switches from the waiting state to the ready state, and then puts it back in the ready queue. A process continues this cycle till it terminates, at which time it is removed from all queues and has its PCB and resources deallocated.

Types of Schedulers

There are three types of schedulers available which are as follows −

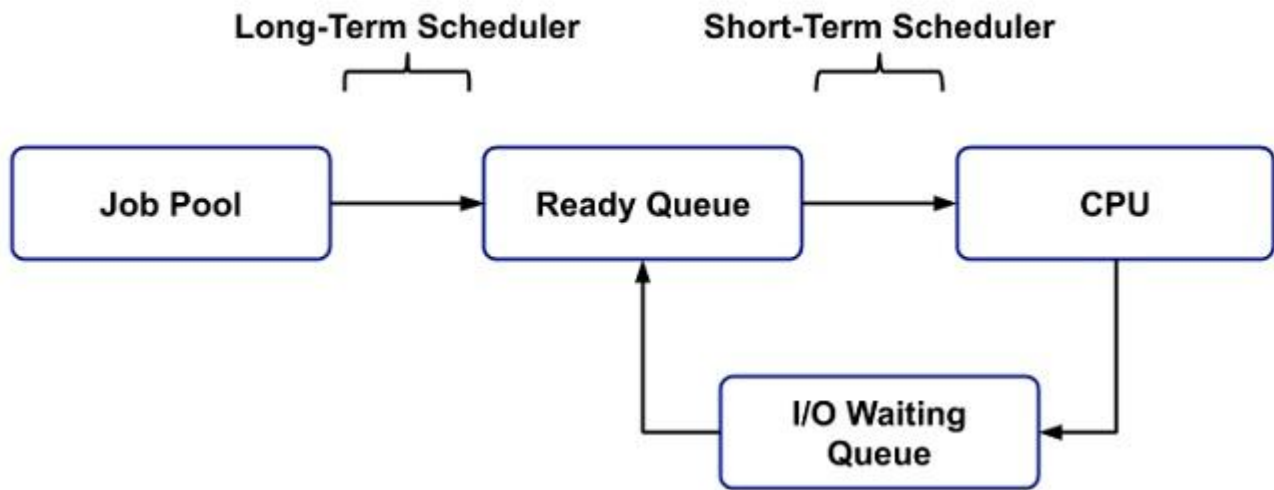
Long Term Scheduler

Long term scheduling is performed when a new process is created, if the number of ready processes in the ready queue becomes very high. Then, there is an overhead on the operating system, for maintaining long lists, containing switching and dispatching increases. Therefore, allowing only a limited number of processes into the ready queue, the long term scheduler manages this.

Long term scheduler runs less frequently. It decides which program must get into the job queue. From the job queue, the job processor selects processes and loads them into the memory for execution.

The main aim of the Job Scheduler is to maintain a good degree of Multiprogramming. The degree of Multiprogramming means the average rate of process creation is equal to the average departure rate of processes from the execution memory.

The diagram of long term and short term scheduler is as follows −



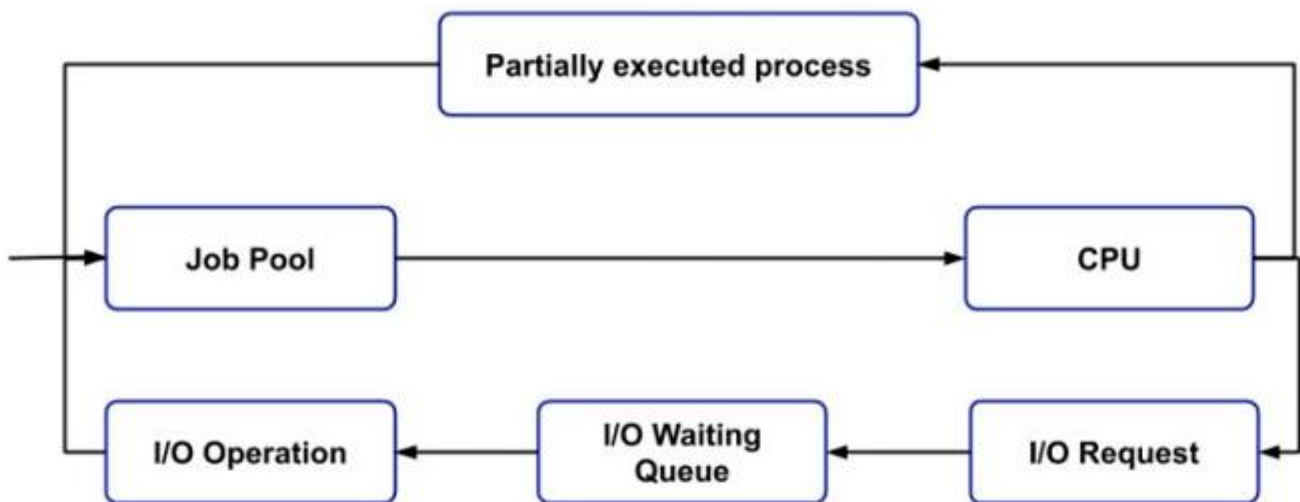
Short Term Scheduler

Short term scheduler is called a CPU Scheduler and runs very frequently. The aim of the scheduler is to enhance CPU performance and increase process execution rate.

Medium Term Scheduler

This type of scheduling removes the processes from memory and thus reduces the degree of multiprogramming. Later, the process is reintroduced into memory and its execution is continued where it left off. This is called **swapping**. The process is swapped out, and is later swapped in, by the medium term scheduler.

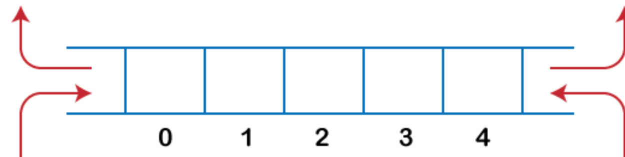
The diagram of medium term scheduler is as follows −



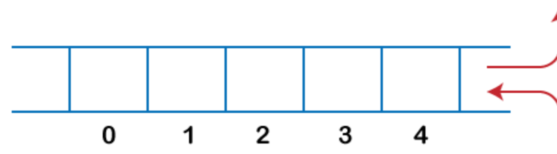
UNIT - 4
DEQUES AND HASHING

Deque:

The dequeue represents Double Ended Queue. In the queue, the inclusion happens from one end while the erasure happens from another end. The end at which the addition happens is known as the backside while the end at which the erasure happens is known as front end.

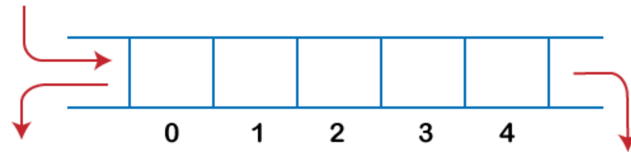


Deque is a direct information structure in which the inclusion and cancellation tasks are performed from the two finishes. We can say that deque is a summed up form of the line. How about we take a gander at certain properties of deque. Deque can be utilized both as stack and line as it permits the inclusion and cancellation procedure on the two finishes. In deque, the inclusion and cancellation activity can be performed from one side. The stack adheres to the LIFO rule in which both the addition and erasure can be performed distinctly from one end; in this way, we reason that deque can be considered as a stack.



In deque, the addition can be performed toward one side, and the erasure should be possible on another end. The queue adheres to the FIFO rule in which the component is embedded toward one side and erased from another end. Hence, we reason that the deque can likewise be considered as the queue.

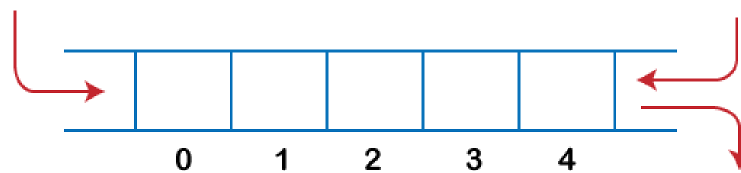
There are two types of Queues, Input-restricted queue, and output-restricted queue. Information confined queue: The info limited queue implies that a few limitations areapplied to the inclusion. In info confined queue, the addition is applied to one end while the erasure is applied from both the closures.



Yield confined queue: The yield limited line implies that a few limitations are applied

to the erasure activity. In a yield limited queue, the cancellation can be applied

uniquely from one end, while the inclusion is conceivable from the two finishes.



Operations on Deque
The following are the operations applied on deque:

- Insert at front
- Delete from end
- insert at rear
- delete from rear

Other than inclusion and cancellation, we can likewise perform look activity in deque. Through look activity, we can get the front and the back component of the dequeue.

We can perform two additional procedure on dequeue:

isFull(): This capacity restores a genuine worth if the stack is full; else, it restores a bogus worth.

isEmpty(): This capacity restores a genuine worth if the stack is vacant; else it restores a bogus worth.

Memory Representation

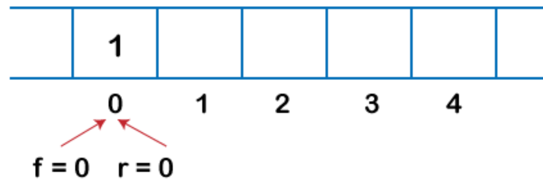
The deque can be executed utilizing two information structures, i.e., round exhibit, and doubly connected rundown. To actualize the deque utilizing round exhibit, we initially should realize what is roundabout cluster.

Implementation of Deque using a circular array:

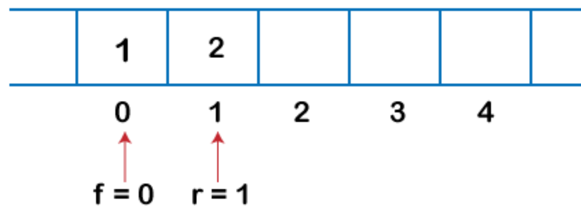
The following are the steps to perform the operations on the Deque:

Enqueue operation

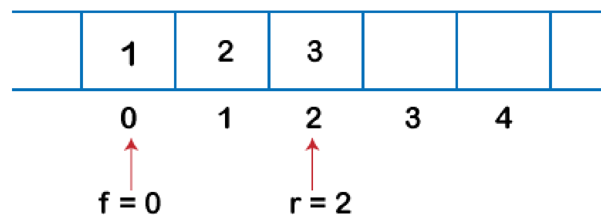
1. At first, we are thinking about that the deque is unfilled, so both front and back are set to - 1, i.e., f = - 1 and r = - 1.
2. As the deque is vacant, so embeddings a component either from the front or backside would be something very similar. Assume we have embedded component 1, at that point front is equivalent to 0, and the back is likewise equivalent to 0.



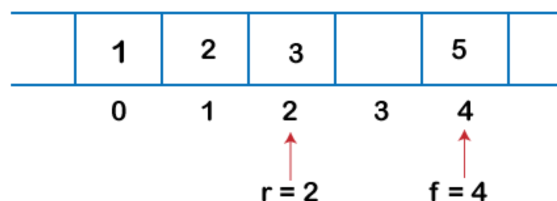
3. Assume we need to embed the following component from the back. To embed the component from the backside, we first need to augment the back, i.e., rear=rear+1. Presently, the back is highlighting the subsequent component, and the front is highlighting the main component.



4. Assume we are again embeddings the component from the backside. To embed the component, we will first addition the back, and now back focuses to the third component.

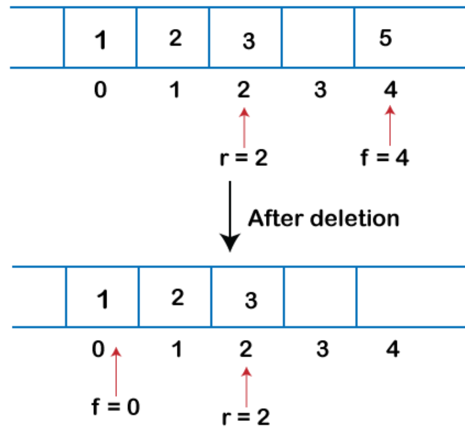


5. In the event that we need to embed the component from the front end, and addition a component from the front, we need to decrement the estimation of front by 1. In the event that we decrement the front by 1, at that point the front focuses to - 1 area, which isn't any substantial area in an exhibit. Thus, we set the front as (n - 1), which is equivalent to 4 as n is 5. When the front is set, we will embed the incentive as demonstrated in the beneath figure:

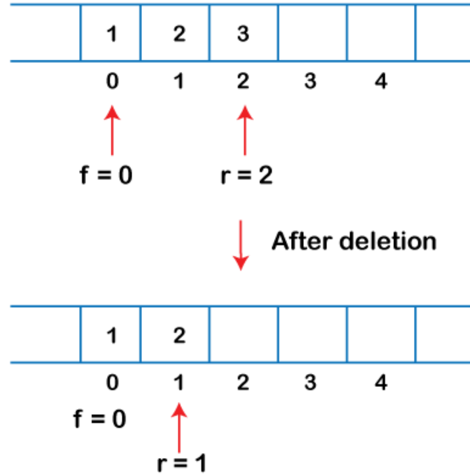


7.12.Dequeue Operation

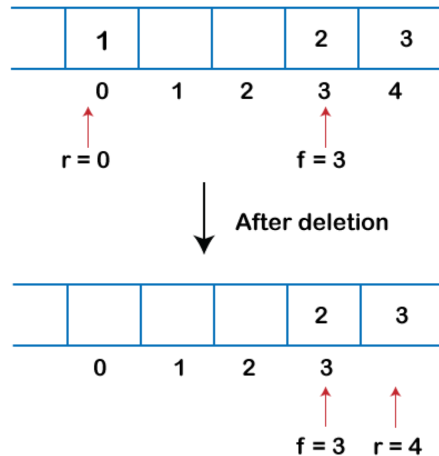
1. On the off chance that the front is highlighting the last component of the exhibit, and we need to play out the erase activity from the front. To erase any component from the front, we need to set front=front+1. At present, the estimation of the front is equivalent to 4, and in the event that we increase the estimation of front, it becomes 5 which is definitely not a substantial list. Thusly, we presume that in the event that front focuses to the last component, at that point front is set to 0 if there should be an occurrence of erase activity.



2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., rear=rear-1 as shown in the below figure:



3. In the event that the back is highlighting the principal component, and we need to erase the component from the backside then we need to set rear=n-1 where n is the size of the exhibit as demonstrated in the beneath figure:



Applications of Deque

 The deque can be utilized as a stack and line; subsequently, it can perform both re-try and fix activities.

- It tends to be utilized as a palindrome checker implies that in the event that we read the string from the two closures, at that point the string would be the equivalent.
- It tends to be utilized for multiprocessor planning. Assume we have two processors, and every processor has one interaction to execute. Every processor is appointed with an interaction or a task, and each cycle contains numerous strings. Every processor keeps a deque that contains strings that are prepared to execute. The processor executes an interaction, and on the off chance that a cycle makes a kid cycle, at that point that cycle will be embedded at the front of the deque of the parent interaction. Assume the processor P2 has finished the execution of every one of its strings then it takes the string from the backside of the processor P1 and adds to the front finish of the processor P2. The processor P2 will take the string from the front end; thusly, the erasure takes from both the closures, i.e., front and backside. This is known as the A-take calculation for planning.

Hash Tables :

Introduction:

We've seen searches that allow you to look through data in *O*(*n*) time, and searches that allow you to look through data in *O*(*logn*) time, but imagine a way to find exactly what you want in *O*(1) time. Think it's not possible? Think again! Hash tables allow the storage and retrieval of data in an average time

At its most basic level, a hash table data structure is just an array. Data is stored into this array at specific indices designated by a hash function. A hash function is a mapping between the set of input data and a set of integers.

With hash tables, there always exists the possibility that two data elements will hash to the same integer value. When this happens, a collision results (two data members try to occupy the same place in the hash table array),and methods have been devised to deal with such situations. In this guide, we will cover two methods, linear probing and separate chaining, focusing on the latter.

A hash table is made up of two parts: an array (the actual table where the data to be searched is stored) and a mapping function, known as a hash function. The hash function is a mapping from the input space to the integer space that defines the indices of the array. In other words, the hash function provides a way for assigning numbers to the input data such that the data can then be stored at the array index corresponding to the assigned number.

Let's take a simple example. First, we start with a hash table array of strings (we'll use strings as the data being stored and searched in this example). Let's say the hash table size is 12:

	Hash Table(strings)
0	(null)
1	(null)
2	(null)
3	(null)
4	(null)
5	(null)
6	(null)
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Next we need a hash function. There are many possible ways to construct a hash function. We'll discuss these possibilities more in the next section. For now, let's assume a simple hash function that takes a string as input. The returned hash value will be the sum of the ASCII characters that make up the string mod the size of the table:*int hash(char *str, int table_size) { int sum; /* Make sure a valid string passed in */ if (str==NULL) return -1; /* Sum up all the characters in the string */ for(; *str; str++) sum += *str; /* Return the sum mod the table size */ return sum % table_size; }* We run "Steve" through the hash function, and find that hash("Steve",12) yields 3:

Figure %: The hash table after inserting "Steve"

Let's try another string: "Spark". We run the string through the hash function and find that hash("Spark",12) yields 6. Fine. We insert it into the hash table:

	Hash Table(strings)
0	(null)
1	(null)
2	(null)
3	"Steve"
4	(null)
5	(null)
6	"Spark"
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

	Hash Table(strings)
0	(null)
1	(null)
2	(null)
3	"Steve"
4	(null)
5	(null)
6	(null)
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure %: The hash table after inserting "Spark"

Let's try another: "Notes". We run "Notes" through the hash function and find that hash("Notes",12) is 3. Ok. We insert it into the hash table:

	Hash Table(strings)
0	(null)
1	(null)
2	(null)
3	"Steve" "Notes"
4	(null)
5	(null)
6	"Spark"
7	(null)
8	(null)
9	(null)
10	(null)
11	(null)

Figure %: A hash table collision

What happened? A hash function doesn't guarantee that every input will map to a different output. There is always the chance that two inputs will hash to the same output. This indicates that both elements should be

inserted at the same place in the array, and this is impossible. This phenomenon is known as a collision. There are many algorithms for dealing with collisions, such as linear probing an d separate chaining. While each of the methods has its advantages, we will only discuss separate chaining here.

Separate chaining requires a slight modification to the data structure. Instead of storing the data elements right into the array, they are stored in linked lists. Each slot in the array then points to one of these linked lists. When an element hashes to a value, it is added to the linked list at that index in the array. Because a linked list has no limit on length, collisions are no longer a problem. If more than one element hashes to the same value, then both
are stored in that linked list.

Let's look at the above example again, this time with our modified data structure:

Figure %: Modified table for separate chaining

Again, let's try adding "Steve" which hashes to 3:

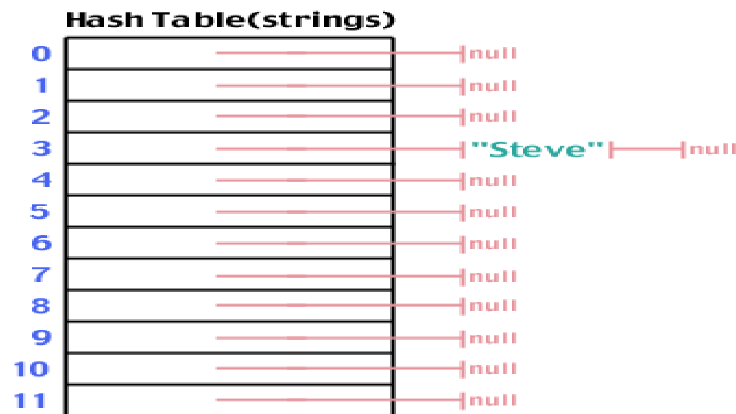


Figure %: After adding "Steve" to the table And "Spark" which hashes to 6:

Problem : How does a hash table allow for $O(1)$ searching? What is the worst case efficiency of a look up in a hash table using separate chainging?

A hash table uses hash functions to compute an integer value for data. This integer value can then be used as an index into an array, giving us a constant time access to the requested data. However, using separate chaining, we won't always achieve the best and average case efficiency of $O(1)$. If we have too small a hash table for the data set size and/or a bad hash function, elements can start to build in one index in

the array. Theoretically, all *n* element could end up in the same linked list. Therefore, to do a search in the worst case is equivalent to looking up a data element in a linked list, something we already know to be $O(n)$ time. However, with a good hash function and a well created hash table, the chances of this happening are, for all intents and purposes, ignorable. **Problem :** The bigger the ratio between the size of the hash table and the number of data elements, the less chance there is for collision. What is a drawback to making the hash table big enough so the chances of collision is ignorable?

Wasted memory space

Problem : How could a linked list and a hash table be combined to allow someone to run through the list from item to item while still maintaining the ability to access an individual element in $O(1)$ time?

Hash Functions

As mentioned briefly in the previous section, there are multiple ways for constructing a hash function. Remember that hash function takes the data as input (often a string), and return s an integer in the range of possible indices into the hash table. Every hash function must do that, including the bad ones. So what makes for a good hash function?

Characteristics of a Good Hash Function

There are four main characteristics of a good hash function:

1) The hash value is fully determined by the data being hashed.
2) The hash function uses all the input data.
3) The hash function "uniformly" distributes the data across the entire set of possible hash values.
4) The hash function generates very different hash values for similar strings. Let's examine why each of these is important:

Rule 1: If something else besides the input data is used to determine the hash, then the hash value is not as dependent upon the input data, thus allowing for a worse distribution of the hash values.

Rule 2: If the hash function doesn't use all the inp5u5t data, then slight variations to the input data would cause an inappropriate number of similar hash values resulting in too many collisions.

Rule 3: If the hash function does not uniformly distribute the data across the entire set of possible hash values, a large number of collisions will result, cutting down on the efficiency of the hash table.

Rule 4: In real world applications, many data sets contain very similar data elements.

Hash Table is a data structure which stores data in an associative manner. In a hash table, data is stored in an array format, where each data value has its own unique index value. Access of data becomes very fast if we know the index of the desired data.

Thus, it becomes a data structure in which insertion and search operations are very fast irrespective of the size of the data. Hash Table uses an array as a storage medium and uses hash technique to generate an index

where an element is to be inserted or is to be located from.

Hashing

Hashing is a technique to convert a range of key values into a range of indexes of an array. We're going to use modulo operator to get a range of key values. Consider an example of hash table of size 20, and the following items are to be stored. Item are in the (key,value) format.

- (1,20)
- (2,70)
- (42,80)
- (4,25)
- (12,44)
- (14,32)
- (17,11)
- (13,78)
- (37,98)

Sr.No.	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

Linear Probing

As we can see, it may happen that the hashing technique is used to create an already used index of the array. In such a case, we can search the next empty location in the array by looking into the next cell until we find an empty cell. This technique is called linear probing.

Sr.No.	Key	Hash	Array Inex	After Linear Probing, Array Index
1	1	1 % 20 = 1	1	1
2	2	2 % 20 = 2	2	2

3	42	42 % 20 = 2	2	3
4	4	4 % 20 = 4	4	4
5	12	12 % 20 = 12	12	12
6	14	14 % 20 = 14	14	14
7	17	17 % 20 = 17	17	17
8	13	13 % 20 = 13	13	13
9	37	37 % 20 = 17	17	18

Basic Operations

Following are the basic primary operations of a hash table.

Search − Searches an element in a hash table. **Insert** − inserts an element in a hash table. **delete** − Deletes an element from a hash table. DataItem

Define a data item having some data and key, based on which the search is to be conducted in a hash table.

```
struct
DataItem {
int data;
  int key;
};
```

Hash Method

Define a hashing method to compute the hash code of the key of the data item.

```
int hashCode(int key){
   return key % SIZE;
}
```

Search Operation

Whenever an element is to be searched, compute the hash code of the key passed and locate the element using that hash code as index in the array. Use linear probing to get the element ahead if the element is not found at the computed hash code.

Insert Operation

Whenever an element is to be inserted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing for empty location, if an element is found at the computed hash code.

Delete Operation

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

OpenAddressing

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, the size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot's key doesn't become equal to k or an empty slot is reached.

Delete(k): ***Delete operation is interesting***. If we simply delete a key, then the search may fail. So slots of deleted keys are marked specially as "deleted". The insert can insert an item in a deleted slot, but the search doesn't stop at a deleted slot.

Open Addressing is done in the following ways:

a) Linear Probing: In linear probing, we linearly probe for next slot. For example, the typical gap between two probes is 1 as seen in the example below. Let **hash(x)** be the slot index computed using a hash function and **S** be the table size

If slot hash(x) % S is full, then we try (hash(x) + 1) % S

If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) %

S If (hash(x) + 2) % S is also full, then we try (hash(x) + 3)

% S

Let us consider a simple hash function as "key mod 7" and a sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Challenges in Linear Probing :

1. **Primary Clustering:** One of the problems with linear probing is Primary clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search for an element.
2. **Secondary Clustering*:*** Secondary clustering is less severe, two records only have the same collision chain (Probe Sequence) if their initial position is the same.

b) Quadratic Probing We look for i^2'th slot in i'th iteration. let hash(x) be the slot index computed using hash function. If slot hash(x) % S is full,

then we try (hash(x) + 1*1) % S

If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) %

S If (hash(x) + 2*2) % S is also full, then we try (hash(x) + 3*3)

% S

c) **Double Hashing** We use another hash function hash2(x) and look for i*hash2(x) slot in i'th rotation.

let hash(x) be the slot index computed using hash function.

If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S

If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S

If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

Comparison:

Linear probing has the best cache performance but suffers from clustering. One more advantage of Linear probing is easy to compute. Quadratic probing lies between the two in terms of cache performance and clustering. Double hashing has poor cache performance but no clustering. Double hashing requires more computation time as two hash functions need to be computed.

S.No.	**Separate Chaining**	**Open Addressing**
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	Cache performance of chaining is not good as keys are stored using linked list.	Open addressing provides better cache performance as everything is stored in the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.

7.	Chaining uses extra space for links.	No links in Open addressing

PerformanceofOpenAddressing:

Like Chaining, the performance of hashing can be evaluated under the assumption that each key is equally likely to be hashed to any slot of the table (simple uniform hashing)

Applications of hashing:

1. Database indexing: Hashing is used to index and retrieve data efficiently in databases and other data storage systems.
2. Password storage: Hashing is used to store passwords securely by applying a hash function to the password and storing the hashed result, rather than the plain text password.
3. Data compression: Hashing is used in data compression algorithms, such as the Huffman coding algorithm, to encode data efficiently.
4. Search algorithms: Hashing is used to implement search algorithms, such as hash tables and bloom filters, for fast lookups and queries.
5. Cryptography: Hashing is used in cryptography to generate digital signatures, message authentication codes (MACs), and key derivation functions.
6. Load balancing: Hashing is used in load-balancing algorithms, such as consistent hashing, to distribute requests to servers in a network.
7. Blockchain: Hashing is used in blockchain technology, such as the proof-of-work algorithm, to secure the integrity and consensus of the blockchain.
8. Image processing: Hashing is used in image processing applications, such as perceptual hashing, to detect and prevent image duplicates and modifications.
9. File comparison: Hashing is used in file comparison algorithms, such as the MD5 and SHA-1 hash functions, to compare and verify the integrity of files.
10. Fraud detection: Hashing is used in fraud detection and cybersecurity applications, such as intrusion detection and antivirus software, to detect and prevent malicious activities.

Hashing provides constant time search, insert and delete operations on average. This is why hashing is one of the most used data structure, example problems are, distinct elements, counting frequencies of items, finding duplicates, etc.

There are many other applications of hashing, including modern-day cryptography hash functions. Some of these applications are listed below:

- Message Digest
- Password Verification
- Data Structures(Programming Languages)
- Compiler Operation
- Rabin-Karp Algorithm
- Linking File name and path together
- Game Boards
- Graphics

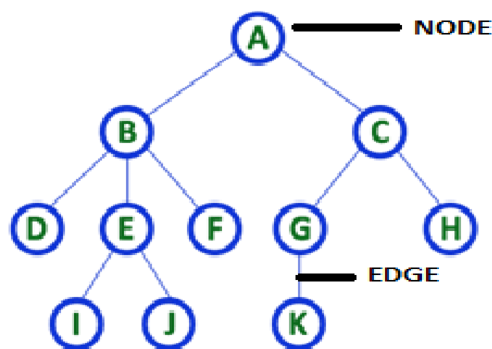
UNIT 5
TREES AND GRAPHS

INTRODUCTION

In linear data structure data is organized in sequential order and in non-linear data structure data is organized in random order. A tree is a very popular non-linear data structure used in a wide range of applications. Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

DEFINITION OF TREE:

Tree is collection of nodes (or) vertices and their edges (or) links. In tree data structure, every individual element is called as **Node**. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.



TREE with 11 nodes and 10 edges

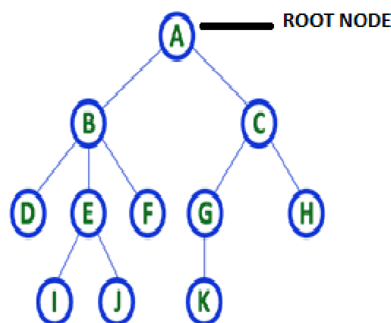
- In any tree with 'N' nodes there will be maximum of 'N-1' edges
- In a tree every individual element is called as 'NODE'

Note: 1. In a **Tree**, if we have **N** number of nodes then we can have a maximum of **N-1** number of links or edges.

2. **Tree** has no cycles.

TREE TERMINOLOGIES:

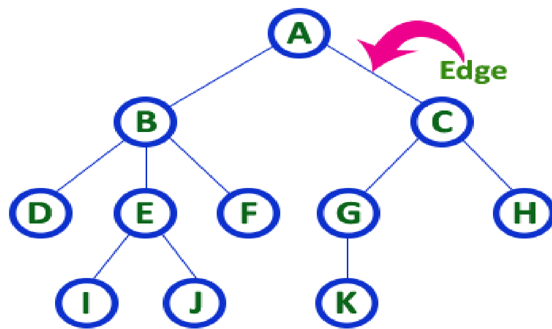
1.Root Node: In a **Tree** data structure, the first node is called as **Root Node**. Every tree must have a root node. We can say that the root node is the origin of the tree data structure. In any tree, there must be only one root node. We never have multiple root nodes in a tree.



Here 'A' is the 'root' node

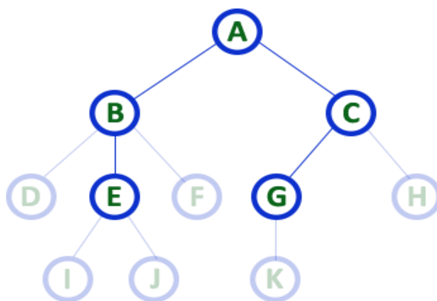
- In any tree the first node is called as ROOT node

2. Edge: In a **Tree**, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



- In any tree, 'Edge' is a connecting link between two nodes.

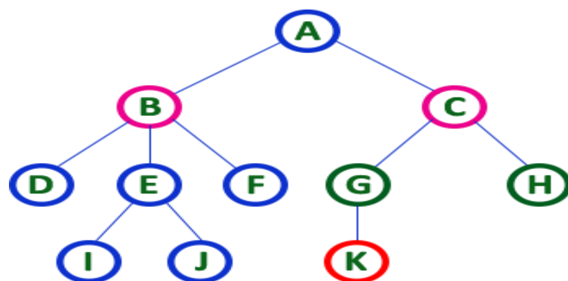
3. Parent Node: In a **Tree**, the node which is a predecessor of any node is called as **PARENT NODE**. In simple words, the node which has a branch from it to any other node is called a parent node. Parent node can also be defined as "**The node which has child / children**". Here, A is parent of B&C. B is the parent of D,E&F and so on…



Here A, B, C, E & G are Parent nodes

- In any tree the node which has child / children is called 'Parent'
- A node which is predecessor of any other node is called 'Parent'

4. Child Node: In a **Tree** data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.



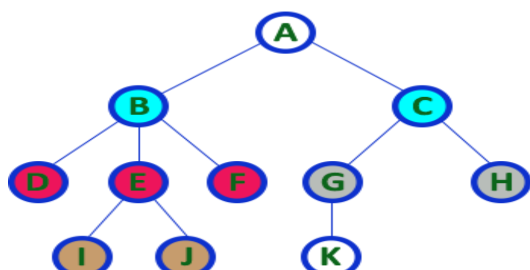
Here B & C are Children of A

Here G & H are Children of C

Here K is Child of G

- descendant of any node is called as CHILD Node

5. Siblings: In a **Tree** data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with the same parent are called Sibling nodes.



Here B & C are Siblings

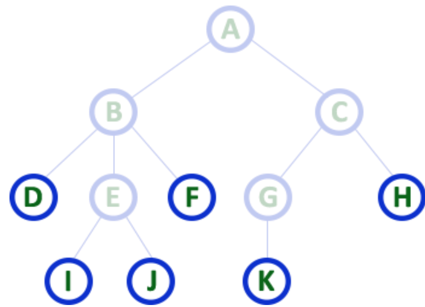
Here D E & F are Siblings

Here G & H are Siblings

Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

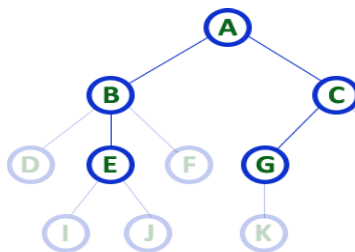
6. Leaf Node: In a **Tree** data structure, the node which does not have a child is called as LEAF Node. In simple words, a leaf is a node with no child. In a tree data structure, the leaf nodes are also called as External Nodes. External node is also a node with no child. In a tree, leaf node is also called as 'Terminal' node.



Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

7. Internal Nodes: In a **Tree** data structure, the node which has atleast one child is called as INTERNAL Node. In simple words, an internal node is a node with atleast one child. In a **Tree** data structure, nodes other than leaf nodes are called as Internal Nodes. The root node is also said to be Internal Node if the tree has more than one node. Internal nodes are also called as 'Non-Terminal' nodes.

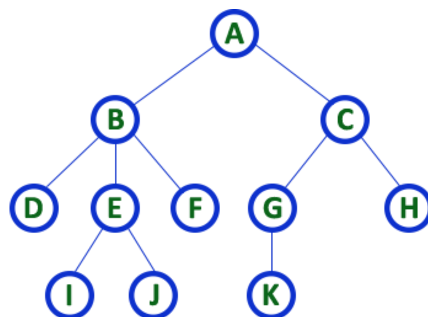


Here A, B, C, E & G are Internal nodes

- In any tree the node which has atleast one child is called 'Internal' node
- Every non-leaf node is called as 'Internal' node

8. Degree: In a **Tree** data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'

Degree of Tree is: 3



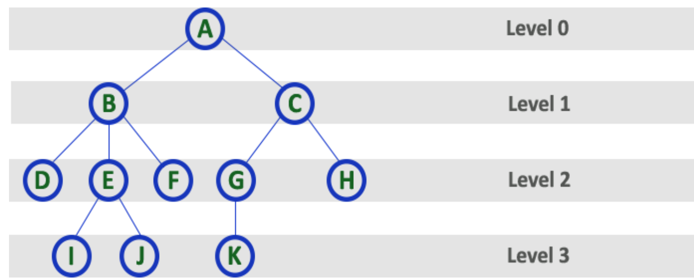
Here Degree of B is 3

Here Degree of A is 2

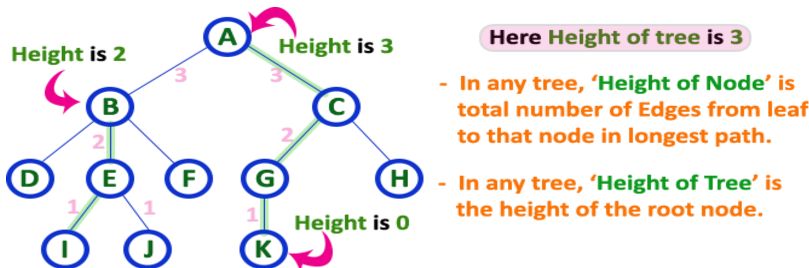
Here Degree of F is 0

- In any tree, 'Degree' of a node is total number of children it has.

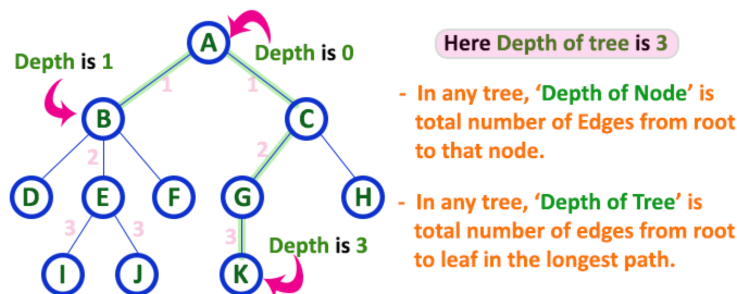
9. Level: In a **Tree** data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



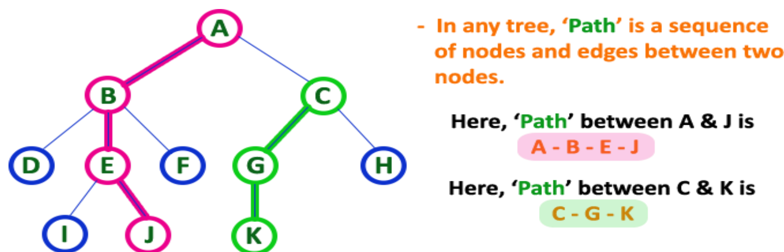
10. Height: In a **Tree** data structure, the total number of edges from leaf node to a particular node in the longest path is called as HEIGHT of that Node. In a tree, height of the root node is said to be height of the tree. In a tree, height of all leaf nodes is '0'.



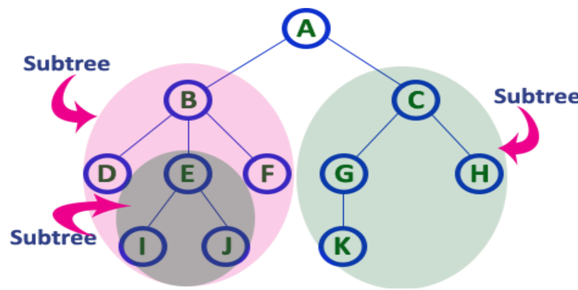
11. Depth: In a **Tree** data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**



12. Path: In a **Tree** data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**. 57



13. Sub Tree: In a **Tree** data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.



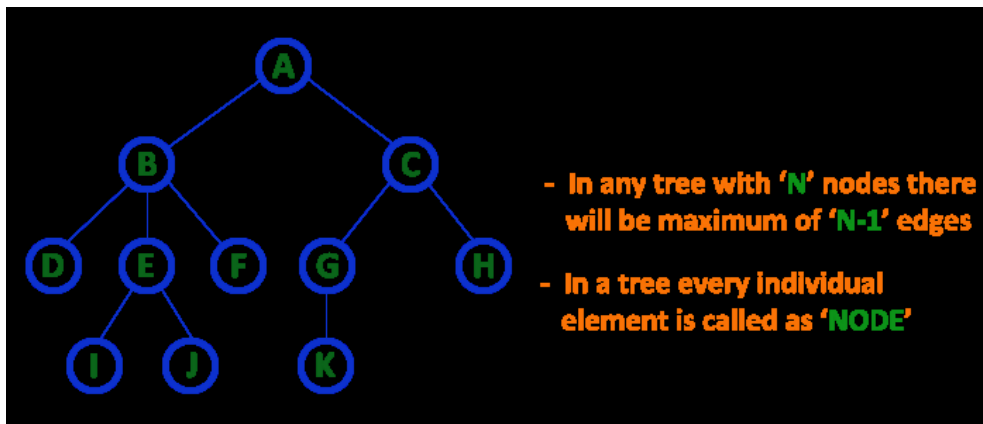
TREE REPRESENTATIONS:

A tree data structure can be represented in two methods. Those methods are as follows...

1. List Representation

2. Left Child - Right Sibling Representation

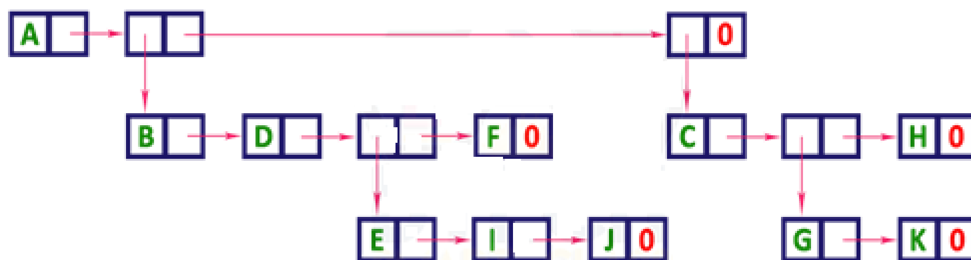
Consider the following tree...



1. List Representation

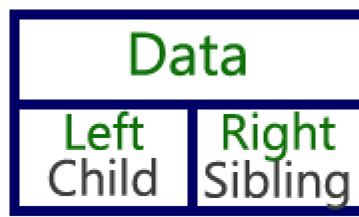
In this representation, we use two types of nodes one for representing the node with data called 'data node' and another for representing only references called 'reference node'. We start with a 'data node' from the root node in the tree. Then it is linked to an internal node through a 'reference node' which is further linked to any other node directly. This process repeats for all the nodes in the tree.

The above example tree can be represented using List representation as follows...



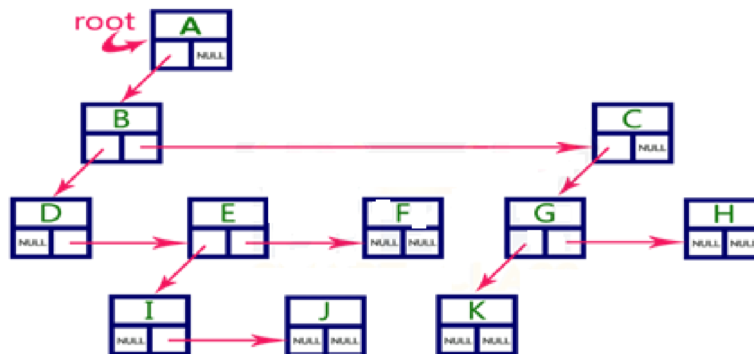
2. Left Child - Right Sibling Representation

In this representation, we use a list with one type of node which consists of three fields namely Data field, Left child reference field and Right sibling reference field. Data field stores the actual value of a node, left reference field stores the address of the left child and right reference field stores the address of the right sibling node. Graphical representation of that node is as follows...



In this representation, every node's data field stores the actual value of that node. If that node has left a child, then left reference field stores the address of that left child node otherwise stores NULL. If that node has the right sibling, then right reference field stores the address of right sibling node otherwise stores NULL.

The above example tree can be represented using Left Child - Right Sibling representation as follows...

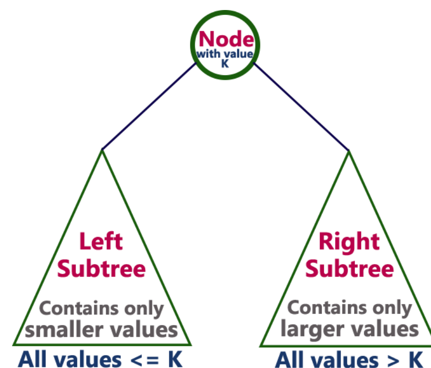


BINARY SEARCH TREE

To enhance the performance of binary tree, we use a special type of binary tree knownas **Binary Search Tree**. Binary search tree mainly focuses on the search operation in a binarytree. Binary search tree can be defined as follows...

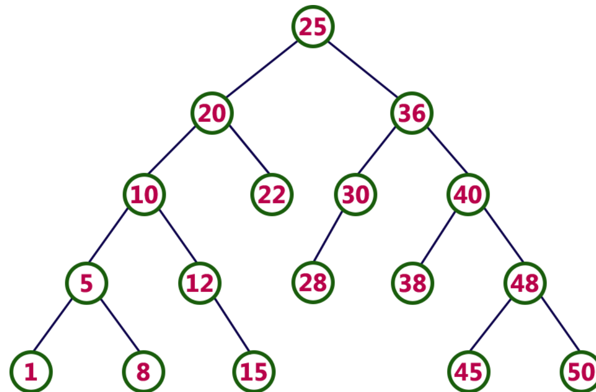
Binary Search Tree is a binary tree in which every node contains only smaller values inits left subtree and only larger values in its right subtree.

In a binary search tree, all the nodes in the left subtree of any node contains smaller values and all the nodes in the right subtree of any node contains larger values as shown in the following figure...

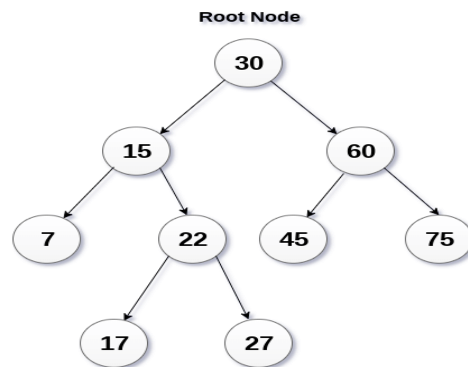


Example

The following tree is a Binary Search Tree. In this tree, left subtree of every node contains nodes with smaller values and right subtree of every node contains larger values.



Every binary search tree is a binary tree but every binary tree need not to be binarysearch tree.



Binary Search Tree

Advantages of using binary search tree

1. Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes o($\log_2 n$) time. In worst case, the time it takes to search an element is 0(n).
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

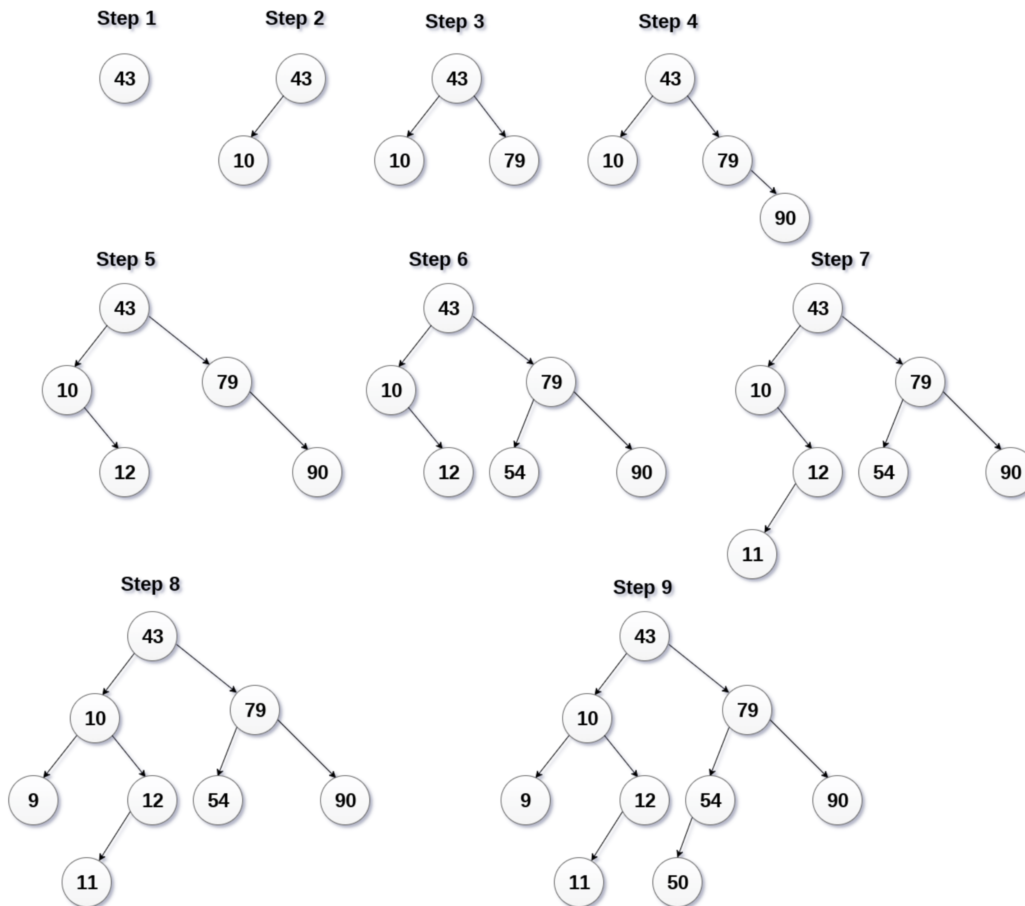
Example1:

Create the binary search tree using the following data elements.

43, 10, 79, 90, 12, 54, 11, 9, 50

1. Insert 43 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element, insert it as the root of the left sub-tree.
3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements, is shown in the image below.



Binary search Tree Creation

Example2

Construct a Binary Search Tree by inserting the following sequence of numbers...

10,12,5,4,20,8,7,15 and 13

OPERATIONS ON A BINARY SEARCH TREE

The following operations are performed on a binary search tree...

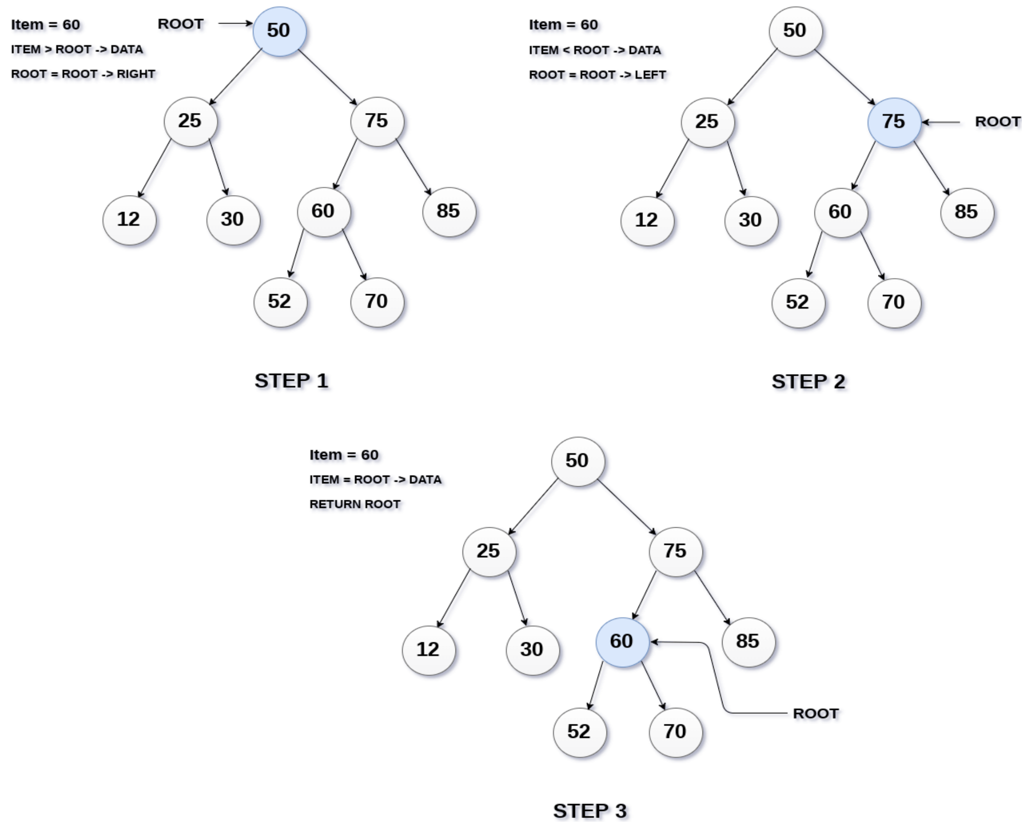
1. **Search**

2. **Insertion**
3. **Deletion**

1. Search Operation in BST

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the factthat, element in BST are stored in a particular order.

1. Compare the element with the root of the tree.
2. If the item is matched then return the location of the node.
3. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
4. If not, then move to the right sub-tree.
5. Repeat this procedure recursively until match found.
6. If element is not found then return NULL.



Algorithm:

Search (ROOT, ITEM)

```
Step 1: IF ROOT -> DATA = ITEM OR ROOT = NULL
                Return ROOT
        ELSE
                IF ROOT < ROOT -> DATA
                        Return search(ROOT -> LEFT, ITEM)
                ELSE
                        Return search(ROOT -> RIGHT,ITEM)
                [END OF IF]
        [END OF IF]

Step 2: END
```

2. Insert Operation in BST

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must node violate the property of binary search tree at each value.

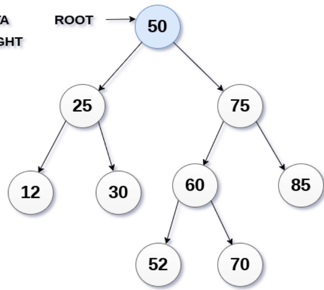
1. Allocate the memory for tree.
2. Set the data part to the value and set the left and right pointer of tree, point to NULL.
3. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
4. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
5. If this is false, then perform this operation recursively with the right sub-tree of the root.

Insert (TREE, ITEM)

```
o Step 1: IF TREE = NULL
                Allocate memory for TREE
                SET TREE -> DATA = ITEM
                SET TREE -> LEFT = TREE -> RIGHT = NULL
          ELSE
                IF ITEM < TREE -> DATA Insert(TREE
                        -> LEFT, ITEM)
                ELSE
                        Insert(TREE -> RIGHT, ITEM)
                [END OF IF]
          [END OF IF]
o Step 2: END
```

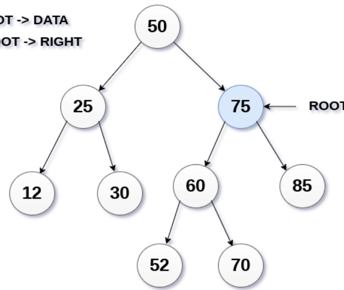
Item = 95

ITEM > ROOT -> DATA
ROOT = ROOT -> RIGHT



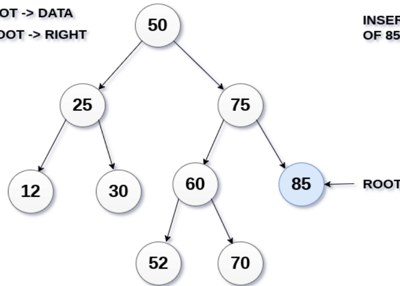
STEP 1

ITEM > ROOT -> DATA
ROOT = ROOT -> RIGHT



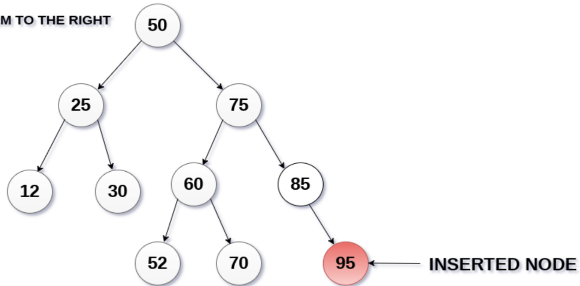
STEP 2

ITEM > ROOT -> DATA
ROOT = ROOT -> RIGHT



STEP 3

INSERT ITEM TO THE RIGHT OF 85



STEP 4

3. Delete Operation in BST

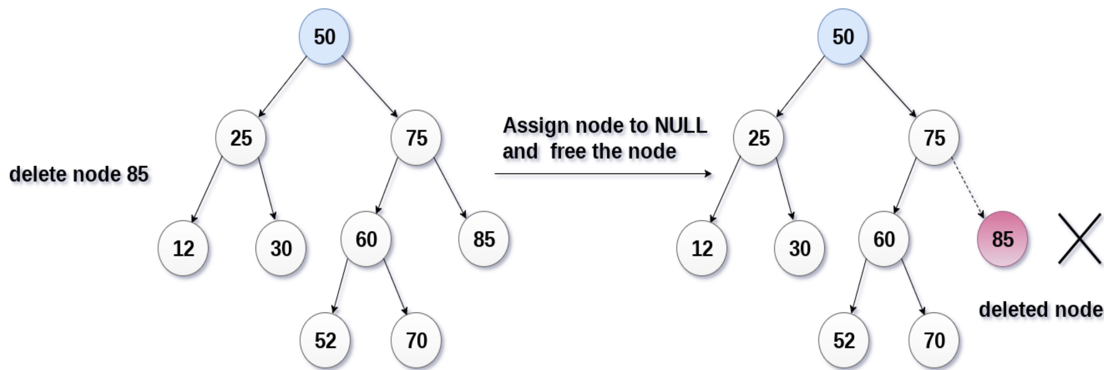
Delete function is used to delete the specified node from a binary search tree. However, we must delete a node from a binary search tree in such a way, that the property of binary searchtree doesn't violate.

There are **three situations of deleting a node** from binary search tree.

a) The node to be deleted is a leaf node

It is the simplest case, in this case, replace the leaf node with the NULL and simple free the allocated space.

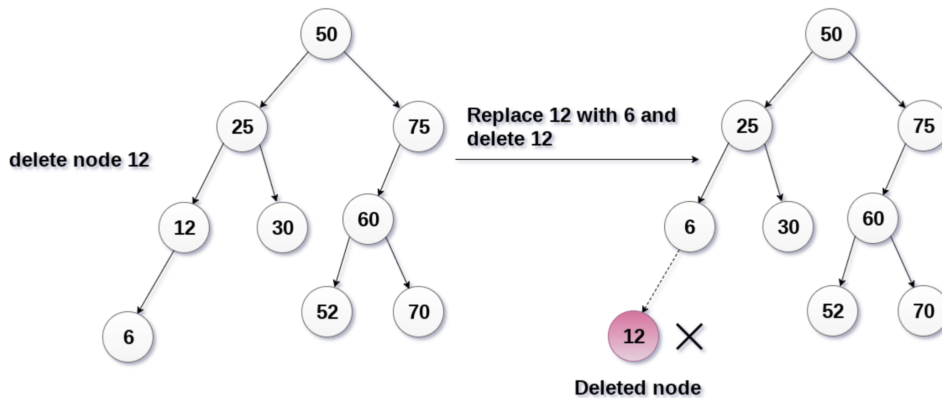
In the following image, we are deleting the node 85, since the node is a leaf node, therefore the node will be replaced with NULL and allocated space will be freed.



b) The node to be deleted has only one child.

In this case, replace the node with its child and delete the child node, which now contains the value which is to be deleted. Simply replace it with the NULL and free the allocated space.

In the following image, the node 12 is to be deleted. It has only one child. The node will be replaced with its child node and the replaced node 12 (which is now leaf node) will simply be deleted.



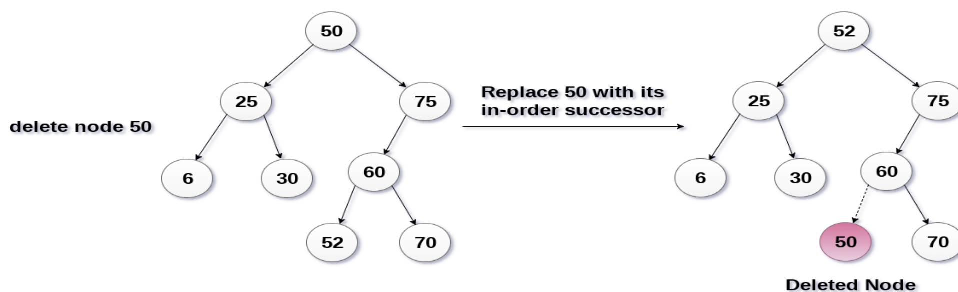
c) The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Algorithm Delete (TREE, ITEM)

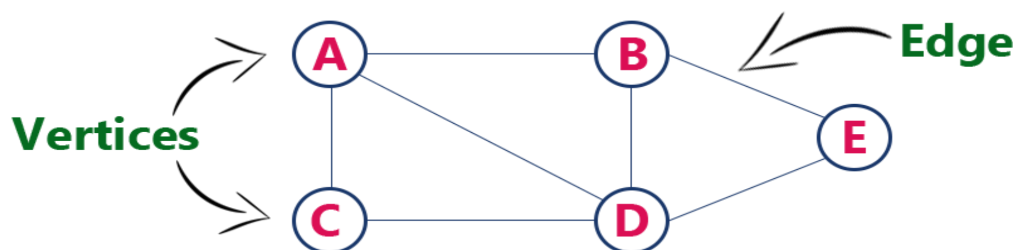
```
Step1: IF TREE=NULL

            Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
            Delete(TREE->LEFT,ITEM)
       ELSE   IF   ITEM>TREE->DATA
            Delete(TREE->RIGHT,ITEM)
       ELSE IF TREE->LEFT AND TREE->RIGHT
            SET TEMP = findLargestNode(TREE -> LEFT)
            SET TREE -> DATA = TEMP -> DATA
            Delete(TREE -> LEFT, TEMP -> DATA)
       ELSE
            SET TEMP = TREE
       IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
            SET TREE = NULL
       ELSE IF TREE -> LEFT != NULL
            SET TREE = TREE -> LEFT
       ELSE
            SET TREE = TREE -> RIGHT
       [END OF IF]
       FREE TEMP
       [END OF IF]
Step 2: END
```

GRAPH TERMINOLOGY

Graph :- Graphs are non-linear data structures comprising a finite set of nodes and edges. The nodes are the elements and edges are ordered pairs of connections between the nodes. Generally, a graph is represented as a pair of sets (V, E). V is the set of vertices or nodes. E is the set of Edges. Simple Definition of Graph:- Graph G can be defined as G = (V , E)

Where V = {A,B,C,D,E} and E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}.



Graph Terminology:-

1) Vertex :Individual data element of a graph is called as Vertex. **Vertex** is also known as **node**. In above example graph, A, B, C, D & E are known as vertices.

2) Edge:An edge is a connecting link between two vertices.

Edges are three types.

1. Undirected Edge **-** An undirected egde is a bidirectional edge. If there is undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).
2. Directed Edge **-** A directed egde is a unidirectional edge. If there is directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).
3. Weighted Edge **-** A weighted egde is a edge with value (cost) on it.

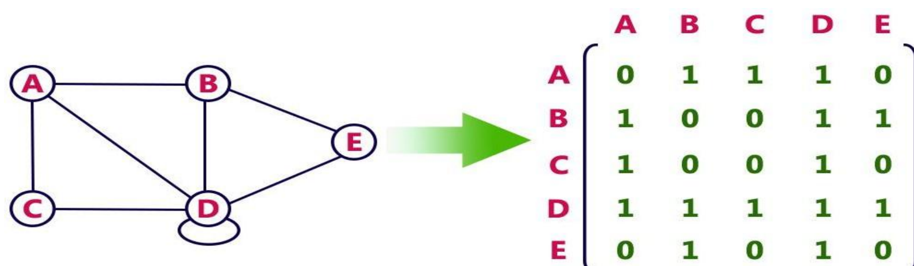
3) Undirected Graph : A graph with only undirected edges is said to be undirected graph.
4) Directed Graph :A graph with only directed edges is said to be directed graph.
5) Mixed Graph :A graph with both undirected and directed edges is said to be mixed graph.
6) End vertices or Endpoints : The two vertices joined by edge are called end vertices (or endpoints) of that edge.
7) Origin : If a edge is directed, its first endpoint is said to be the origin of it.
8) Destination : If a edge is directed, its first endpoint is said to be the origin of it and the other endpoint is said to be the destination of that edge.
9) Adjacent : If there is an edge between vertices A and B then both A and B are said to be adjacent. In other words, vertices A and B are said to be adjacent if there is an edge between them.
10) Incident: Edge is said to be incident on a vertex if the vertex is one of the endpoints of that edge.
11) Outgoing Edge : A directed edge is said to be outgoing edge on its origin vertex.
12) Incoming Edge : A directed edge is said to be incoming edge on its destination vertex.
13) Degree :Total number of edges connected to a vertex is said to be degree of that vertex.
14) Indegree : Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
15) Outdegree : Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
16) Parallel edges or Multiple edges : If there are two undirected edges with same end vertices and two directed edges with same origin and destination, such edges are called parallel edges or multiple edges.
17) Self-loop : Edge (undirected or directed) is a self-loop if its two endpoints coincide with each other.
18) Simple Graph : A graph is said to be simple if there are no parallel and self-loop edges.
19) Path : A path is a sequence of alternate vertices and edges that starts at a vertex and ends at other vertex such that each edge is incident to its predecessor and successor vertex.

GRAPH REPRESENTATION
Graph data structure is represented using following representations...
1. Adjacency Matrix

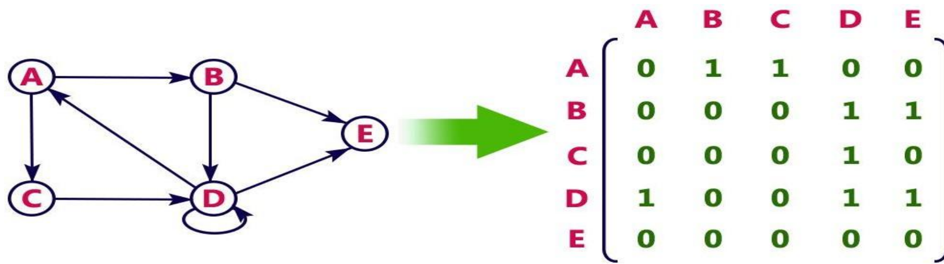
2. Incidence Matrix

3. Adjacency List

Adjacency Matrix : In this representation, the graph is represented using a matrix of size total number of vertices by a total number of vertices. That means a graph with 4 vertices is represented using a matrix of size 4X4. In this matrix, both rows and columns represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents that there is an edge from row vertex to column vertex and 0 represents that there is no edge from row vertex to column vertex.
For example, consider the following
undirected graph representation...



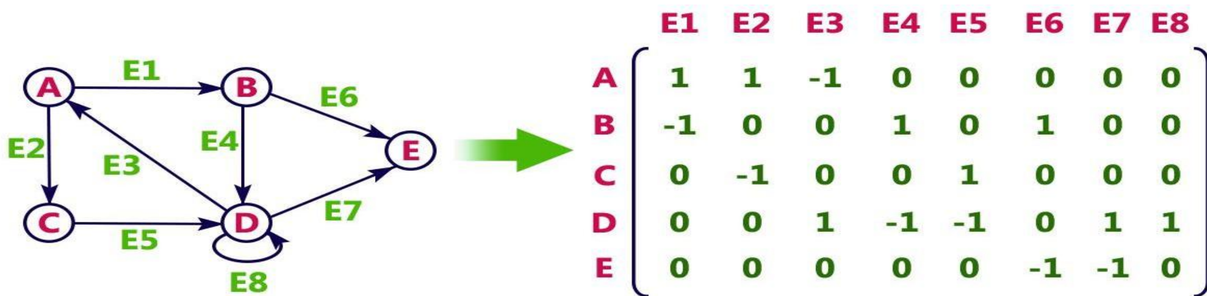
Directed graph representation...



Incidence Matrix :

In this representation, the graph is represented using a matrix of size total number of vertices by a total number of edges. That means graph with 4 vertices and 6 edges is represented using a matrix of size 4X6. In this matrix, rows represent vertices and columns represents edges. This matrix is filled with 0 or 1 or -1. Here, 0 represents that the row edge is not connected to column vertex, 1 represents that the row edge is connected as the outgoing edge to column vertex and -1 represents that the row edge is connected as the incoming edge to column vertex.

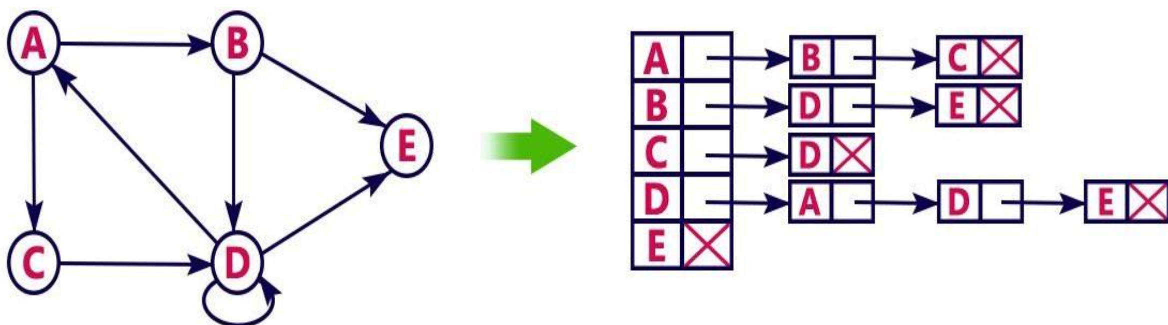
For example, consider the following directed graph representation...



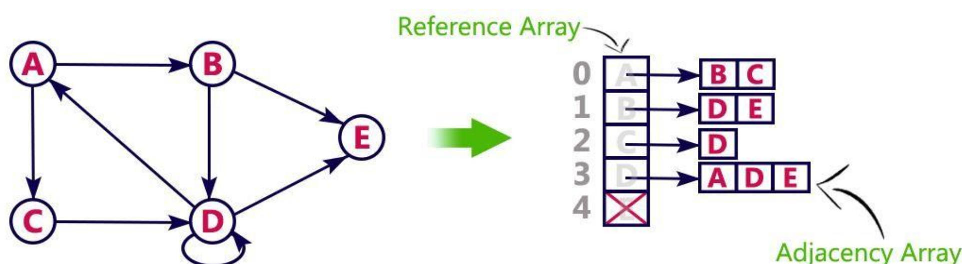
Adjacency List:

In this representation, every vertex of a graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...



This representation can also be implemented using an array as follows..



ELEMENTARY GRAPH OPERATIONS

Given a graph G = (V E) and a vertex v in V(G)

Various graph operations are:-

1) Traversal - visiting all vertices in G exactly once.There are two graph traversal techniques.

Depth First Search (DFS)

Breadth First Search (BFS)

2) Connected components

3) Spanning tree

BREADTH FIRST SEARCH(BFS):

BFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

We use the following steps to implement BFS traversal...

 Step 1 - Define a Queue of size total number of vertices in the graph.

 Step 2 - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.

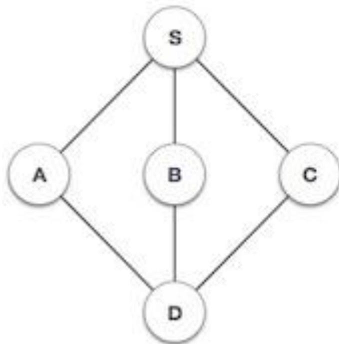
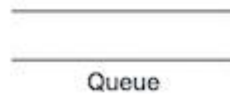
 Step 3 - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.

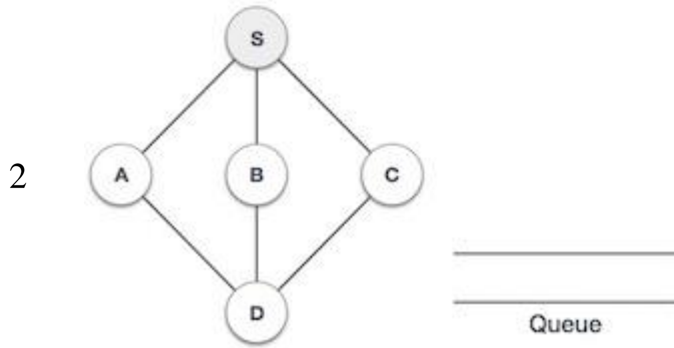
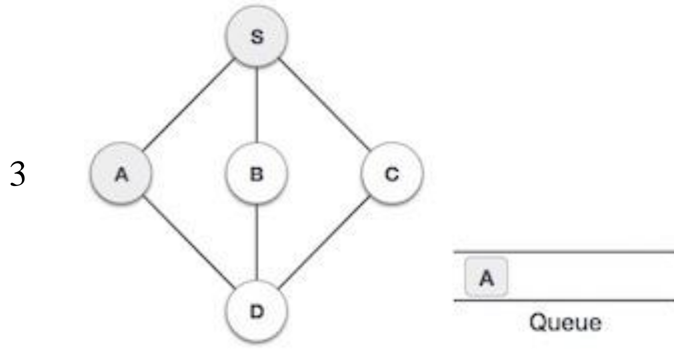
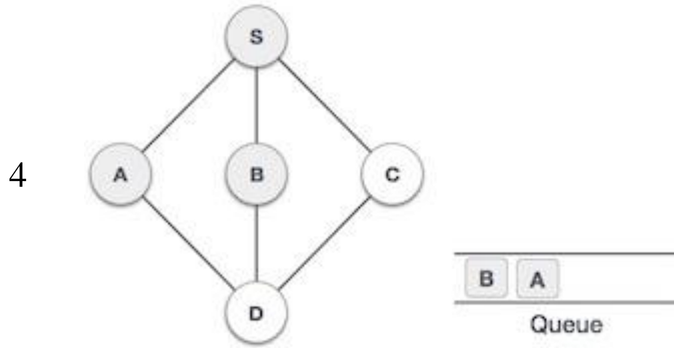
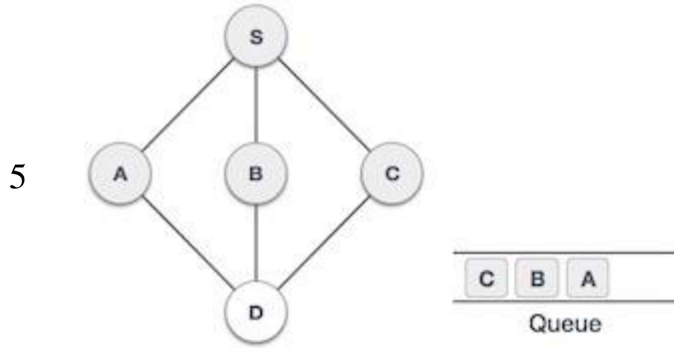
 Step 4 - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.

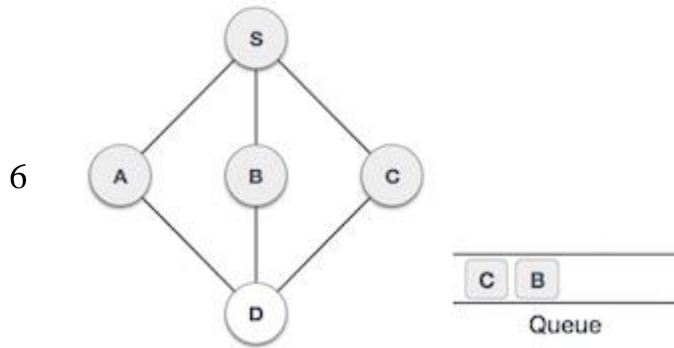
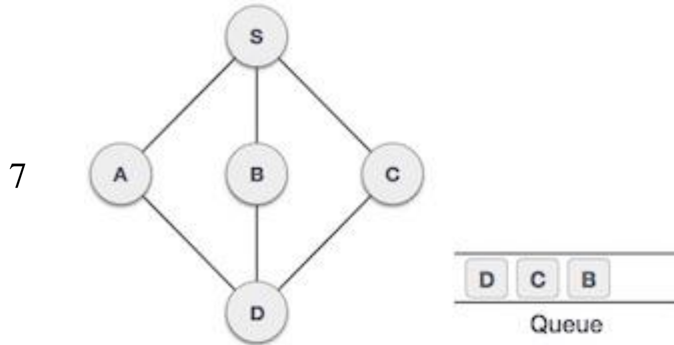
 Step 5 - Repeat steps 3 and 4 until queue becomes empty.

 Step 6 - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

Example:

Step	Traversal	Description
1	 	Initialize the queue.

2		We start from visiting **S** (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from **S**. In this example, we have three nodes but alphabetically we choose **A**, mark it as visited and enqueue it.
4		Next, the unvisited adjacent node from **S** is **B**. We mark it as visited and enqueue it.
5		Next, the unvisited adjacent node from **S** is **C**. We mark it as visited and enqueue it.

6		Now, **S** is left with no unvisited adjacent nodes. So, we dequeue and find **A**.
7		From **A** we have **D** as unvisited adjacent node. We mark it as visited and enqueue it.

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

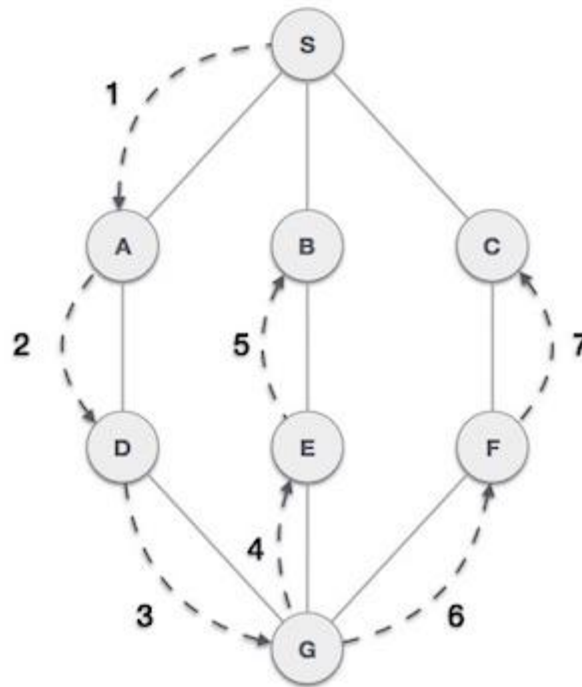
DEPTH FIRST SEARCH

DFS traversal of a graph produces a spanning tree as final result. Spanning Tree is a graph without loops. We use Stack data structure with maximum size of total number of vertices in the graph to implement DFS traversal.

We use the following steps to implement DFS traversal...

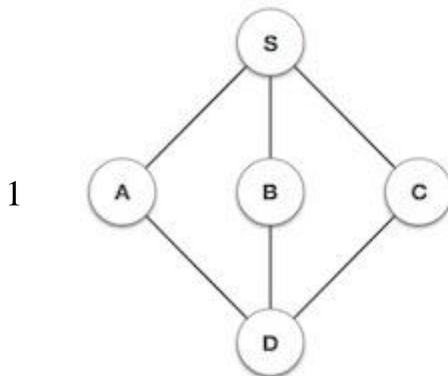
- Step 1 - Define a Stack of size total number of vertices in the graph.
- Step 2 - Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
- Step 3 - Visit any one of the non-visited adjacent vertices of a vertex which is at the top of stack and push it on to the stack.
- Step 4 - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- Step 5 - When there is no new vertex to visit then use back tracking and pop one vertex from the stack.
- Step 6 - Repeat steps 3, 4 and 5 until stack becomes Empty.
- Step 7 - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph,Back tracking is coming back to the vertex from which we reached the current vertex.

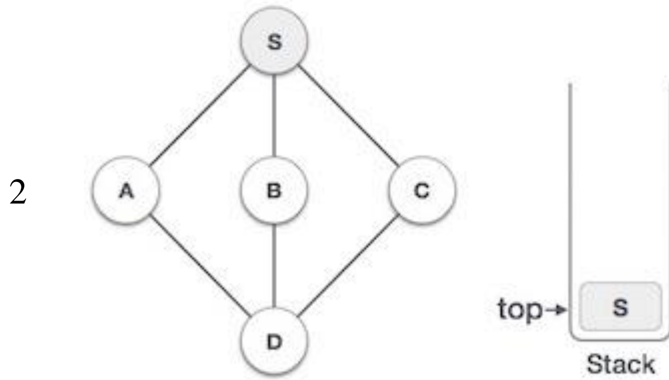
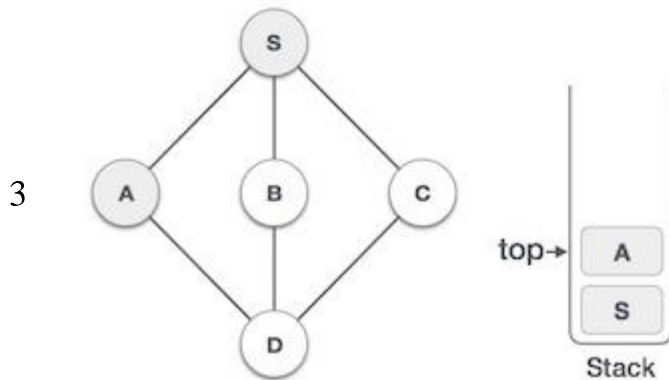
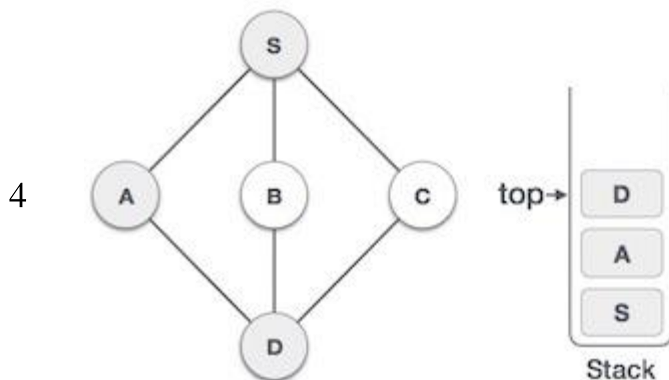
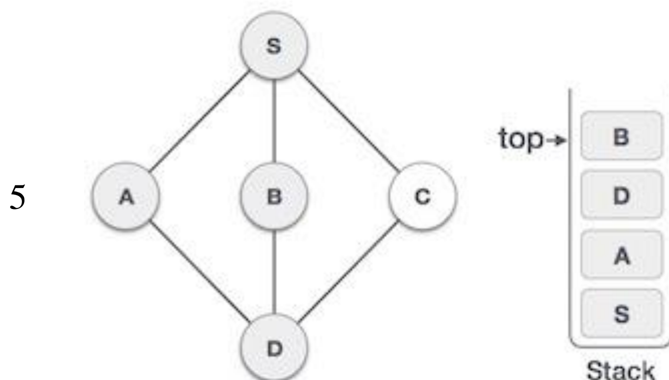
Example:



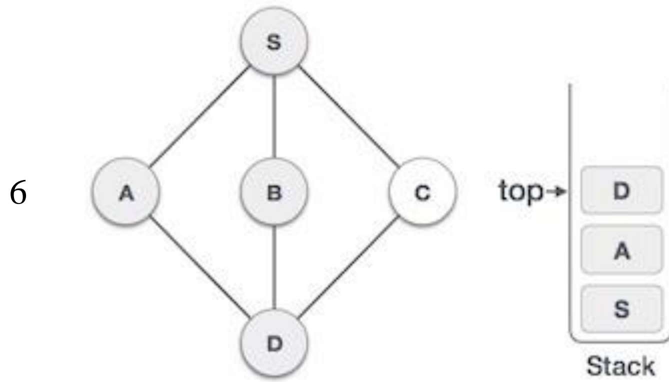
As in the example given above, DFS algorithm traverses from S to A to D to G to E to B first, then to F and lastly to C. It employs the following rules.

- **Rule 1** − Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- **Rule 2** − If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- **Rule 3** − Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

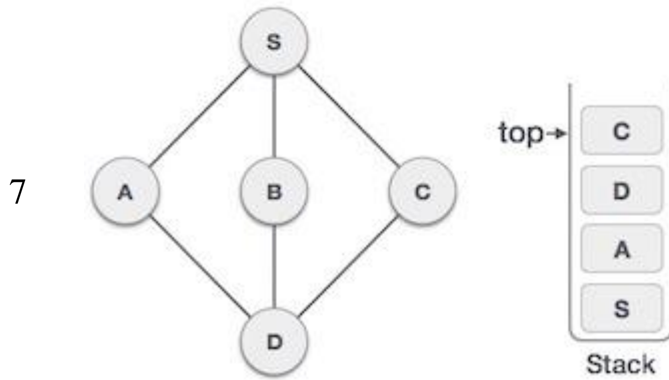
Step	Traversal	Description
2		Mark **S** as visited and put it onto the stack. Explore any unvisited adjacent node from **S**. We have three nodes and we can pick any of them. For this example, we shall take the node in an alphabetical order.
3		Mark **A** as visited and put it onto the stack. Explore any unvisited adjacent node from A. Both **S** and **D** are adjacent to **A** but we are concerned for unvisited nodes only.
4		Visit **D** and mark it as visited and put onto the stack. Here, we have **B** and **C** nodes, which are adjacent to **D** and both are unvisited. However, we shall again choose in an alphabetical order.
5		We choose **B**, mark it as visited and put onto the stack. Here **B** does not have any unvisited adjacent node. So, we pop **B** from the stack.

6



We check the stack top for return to the previous node and check if it has any unvisited nodes. Here, we find **D** to be on the top of the stack.

7



Only unvisited adjacent node is from **D** is **C** now. So we visit **C**, mark it as visited and put it onto the stack.

As **C** does not have any unvisited adjacent node so we keep popping the stack until

we find a node that has an unvisited adjacent node. In this case, there's none and we keep popping until the stack is empty.