CST35 – DATA STRUCTURES

UNIT – 1

INTRODUCTION TO DATA STRUCTURES

DATA:

- It is a collection of facts or some known information, from which a conclusion can be drawn.

DATA STRUCTURES:

- Data Structure is a method of organizing large amount of data more efficiently, such that any operation on that data becomes easy

(or)

- Data structure is the structural representation of logical relationships between elements of data.
- Data structures are the building blocks of a program
- Algorithm is a step by step procedure to solve a problem.
- Before developing a program for an algorithm, first we should select data structures.

ALGORITHM:

- Algorithm is a sequence of unambiguous instructions to solve a problem.
- Algorithm is called as a solution for a given problem.
- A problem may have many solutions/algorithm. We need to select the best algorithm which is suitable for a given problem.
- After the algorithm for the problem is chosen, it is converted to program.

PROGRAM:

- Algorithm + Data Structure = Program
- The program may take zero or more input, process the instructions and produce a single output.

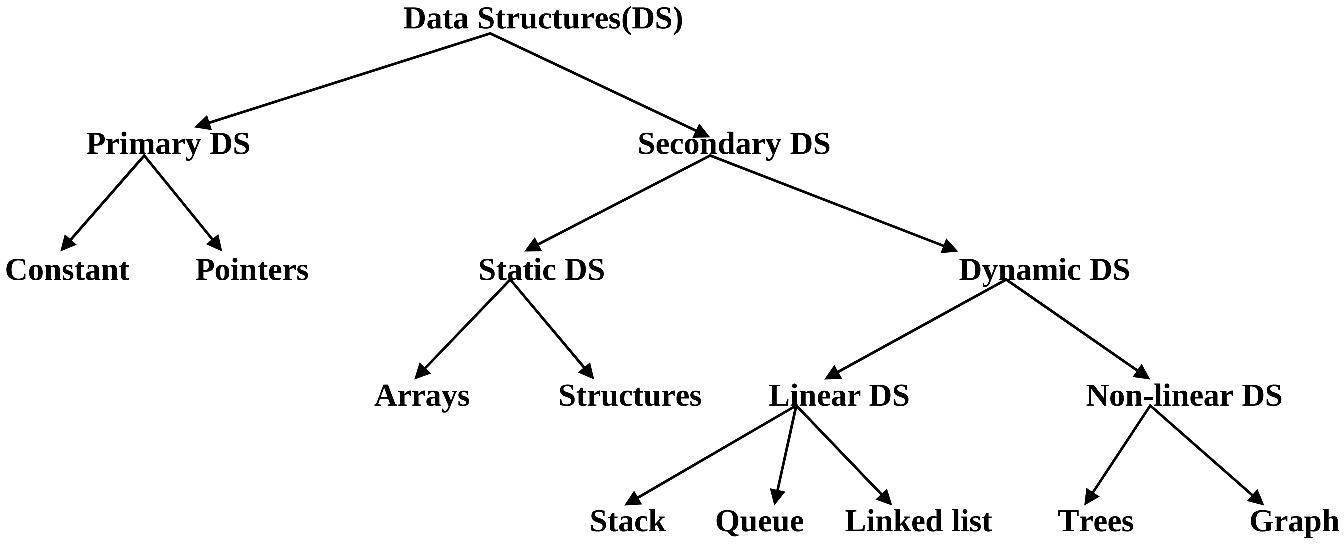
Characteristic of an Algorithm:

1. **Input:** Algorithm may take zero or more input
2. **Output:** Algorithm always produces a single output
3. **Definiteness:** Every instruction in an algorithm must be clear and unambiguous
4. **Finiteness :** For all possible inputs, algorithm must produce result in finite steps
5. **Effectiveness:** The algorithm must be feasible

Parameters considered to chose a best algorithm:

- Execution speed => Time taken to execute the algorithm for a specific input
- Memory => The memory required to store program and data.
- Easy to understand
- Easy to implement
- Correctness

Types of Data Structures:



Primary Data structure:

- These are data structures which operates on machine instructions

Types:

Constants: The value of the constant doesn’t change

Pointers: It stores the address of other variables which helps in direct access of that variable

Secondary Data structures:

i) Static Data structures:

- It is also called as fixed size data structure where the memory allocation of this type of data structure is fixed
- Types: Arrays, Structures
- **Arrays:** It is a collection of elements of same data type which is stored in contiguous memory location
- **Structures:** It is a collection of elements of different data type

ii) Dynamic Data structures:

- It is also called as dynamic size data structure where the memory allocation of this type of data structure varies at run time
- Types: Linear data structures, Non-linear data structures
- **Linear data structures:** They have a linear relation with their adjacent elements
  - Stack => Elements are inserted and deleted based on Last In First Out(**LIFO**) method
  - Queue => Elements are inserted and deleted based on Fast In First Out(**FIFO**) method
  - Linked list => One node is linked with another node by storing the address of the next node.
- **Non-Linear data structures:** They have a non-linear relation with other elements
  - Trees
  - Graph

ALGORITHMIC NOTATIONS:

Name of the algorithm : Each algorithm is given by its name

Data required for the algorithm: Data required for executing an algorithm is given as arguments to the algorithm

Comments : It describes the data passed to the algorithm or the steps of an algorithm

Steps of the algorithm: The sequence of steps which is performed by the algorithm

***Read* =>** Used to read the values

***Write* =>** Used to display the values

Example 1: Algorithm to add two numbers

```
Add(a,b)
c = a + b
write  c
```

In this example, add is a algorithm name, a and b are the data required to perform addition operation and the c= a+b and write c are the sequence of steps to perform addition operation.

Example 2: Algorithm to find greatest of two numbers

```
Great(a,b)
if(a > b)
then
write(“ a is greatest”)
else
write(“b is greatest”)
end if
```

Example 3: Algorithm to display n elements

```
Display(n)
for  i = 1 to n
        write  i
end for
```

Example 4: Algorithm to perform sum of n elements

```
sum( n )
sum =0
for  i = 1 to n
        sum = sum + i
end for
write  sum
```

Example 5: Algorithm to find maximum element in a given array

maxarray(a, n)

```
// a is an array of n elements
max = a[1]
for  i = 2 to n
    if ( a[ i ] > max )
          max = a[ i ]
    end if
end for
write  max
```

- In the above algorithm, At the initial step, a[1] is assigned as max. In for loop, Each of the next element starting from second element is compared with max. If the element is greater than max, value of max changes. After comparing all the elements in array, value in max is displayed.

CREATING PROGRAMS

We consider five phases: requirements, design, analysis, coding, and verification.

(i) ***Requirements*****:**

- Define and ***understand the objective/scope*** of the problem given.
- Based on objective, identify all the requirements to solve the problem.
- (i.e) What are the possible **inputs** for a given problem. What will be the **output** for a various inputs.

(ii) ***Design*** **:**

- Perform the design based on the requirements.
- The design which is developed should be simple and should specify entire aspects of the given problem.

(iii) ***Analysis*****:**

- Analysis is performed to check whether the design which is developed is based on the specifications of the problem.
- If there is any deviations in the design, it is corrected and design is modified based on the problem specification.

(iv) Refinement and coding:

- After the design is made, the appropriate best algorithm is chosen for the problem.
- Select the programming language which is best suited for solving the problem. After selecting a programming language, coding is done for the chosen algorithm.

(v) Verification.

- Verification consists of three distinct aspects: Program proving, Testing and Debugging.
- ***Program proving:*** Before executing your program you should attempt to prove it is correct.
- ***Testing:*** The program is tested for all possible inputs, to find whether any faults/bug is available in a program or the program is giving wrong output. If the program fails to respond correctly then debugging is needed to determine what went wrong and how to correct it.
- ***Debugging:*** Correction of bug in a program, if found. After the bug is identified, steps need to be performed to correct the bug in order to produce the required output.

vi) Documentation:

- After the program is successfully developed, the entire description about the working of program is given in documentation for easy understanding.

ANALYZING PROGRAMS:

Several criteria to analyze the programs are

i) Does it do what we want it to do ? Results obtained according to the specification of the program

ii) Whether Documentation is present ? To describe how it works or how to use it

iii) Modularity exists ? Does the larger problem is divided into logically related sub-modules which will improve efficiency

iv) Is the program readable? Readability

There are other criteria for analyzing the programs which will have direct relationship with performance(i.e) based on the computation time and storage requirements of the algorithm.

Time complexity:

It is the amount of time required to execute the algorithm for the specific input.

Space complexity:

It is the amount of memory taken by the algorithm for execution.

Performance analysis (or) Asymptotic analysis:

The following information must be known to determine, how much time it takes to execute any command.

i) The machine executing on

ii) The machine language instruction set

ii) Time required by each machine instruction

Consider three examples

……………..
 ……………..
 x = x + 1
 ……………..
 ……………...
	for i = 1 to n
	 x = x +1
	 end for
		for i = 1 to n
		 for j = 1 to n
		 x = x +1
		 end for
		 end for

In this example, the statement x=x+1 is not inside any loop. Hence, Number of time the statement executed is 1 **Frequency count = 1**	In this example, the statement x=x+1 is inside for loop. Hence, Number of time the statement executed is n **Frequency count = n**	In nested for loop, Number of times statement executed is equal to the number of times the innermost loop is executed. In this example, the number of times the statement x=x+1 executed is n^2 **Frequency count = n^2**

Asymptotic notations:

- It allows us to analyze an algorithm’s running time by identifying its behavior, as the input size for the algorithm increases. This is also known as an algorithm’s growth rate.

i) Big Oh(O):

Big Oh is defined as

The function f(n) = O(g(n)) if and only if there exists constants c and n_0, such that $f(n) \leq c * g(n)$ for all $n \geq n_0$. Here f(n) and g(n) are computation time of the algorithm

Example:

$f(n) = 3n + 2 \le 4*n$ for all $n \ge 2$, where $f(n) = 3n+2$, $c = 4$, $g(n) = n$ and $n_0 = 2$
$f(n) = O(1)$ represents computing time is constant
$f(n) = O(n)$ represents computing time is linear
$f(n) = O(n^2)$ represents computing time is quadratic
$f(n) = O(n^3)$ represents computing time is cubic
$f(n) = O(2^n)$ represents computing time is exponential
$f(n) = O(\log n)$ represents computing time is logarithmic

Note:

$O(1)<O(\log n)<O(n)<O(n^2)$
Here, Time taken to execute the algorithm having constant time O(1) is least. Time taken to execute the algorithm having quadratic time $O(n^2)$is high.

ii) Omega(Ω):

Omega is defined as

The function $f(n) = \Omega(g(n))$ if and only if there exists constants c and n_0, such that
$f(n) \ge c * g(n)$ for all $n \ge n_0$. Here f(n) and g(n) are computation time of the algorithm

Example:

$f(n) = 3n + 2 \ge 3*n$ for all $n \ge 1$, where $f(n) = 3n+2$, $c = 3$, $g(n) = n$ and $n_0 = 1$

iii) Theta(Θ)

Theta is defined as

The function $f(n) = \Theta(g(n))$ if and only if there exists constants c1, c2 and n_0, such that
$c1 * g(n) \le f(n) \le c2 * g(n)$ for all $n \ge n_0$. Here f(n) and g(n) are computation time of the algorithm

Example:

$3*n \le 3n + 2 \le 4*n$ for all $n \ge 2$, where c1=3, c2=4, $g(n) = n$, $f(n) = 3n+2$, $n_0 = 2$

ARRAY:

- Array is a collection of elements of same data type that are stored under a common name.

Characteristics of array:

- The elements of array are stored in contiguous memory location.
- Individual element of array can be accessed by using array name followed by a integer enclosed with asquare bracket which is called as subscript or index. Example: Second element in array is accessed by a[1].

Operations performed in an array:

i) Traversing => Used to visit all the elements in an array

ii) Sorting => Used to sort the elements of an array in ascending or descending order

iii) Searching => Used to search a given element from an array

iv) Insertion => Used to insert the element in the specified location in an array provided that array is not full

v) Deletion => Used to delete the particular element from array

vi) Merging => Used to merge the elements of two arrays into a single array

Algorithms for operations of array:

i) Traversing:

```
for  i = 1 to n
visit( a[ i ] ) (or) display( a[ i ] )
end for
```

ii) Sorting:

Write any sorting algorithm

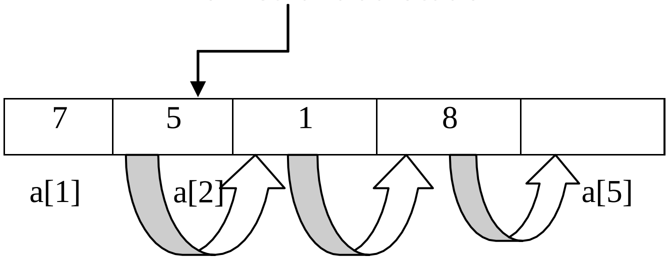
iii) Searching:

Write any searching algorithm

iv) Insertion:

```
if  a[high] ! = NULL
     Write "Array is full, cant insert element"
else
i=high
    while  i>Location
          a[ i ] = a[ i – 1 ]
          i - - ;
    end while
a[Location] = key
end  if
```

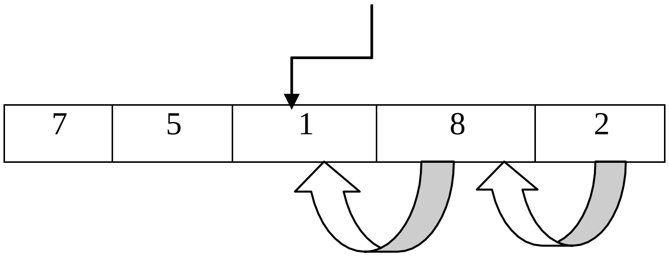
To Insert 10 at location 2



vi) Deletion:

```
Get the key element which is to be deleted
Initialize i=0
i = searcharray(a,key) // i gives the position of the key element
if i = = 0
     write " key not found"
else
      while  i < high
           a[ i ] = a [ i+1 ]
           i + +
      end while
end  if
a[high] = NULL
high = high -1
```

To delete the element 1



vi) Merging

write the merge algorithm in merge sort

Types of array:

Depending on the number of subscripts used, arrays are classified into

- One dimensional array
- Multidimensional array

1. One dimensional array:

- An array with only one subscript is called one dimensional array.

Declaration of an one dimensional array :

- Array must be declared before used in program. When the array is declared, the array is allocated with multiple blocks of empty memory of given size.

Syntax for declaration of an one dimensional array :

data_type arrayname[size] ; (or) *data_type arrayname[subscript] ;* where data type specifies the type of data which is to be stored in array, size specifies the maximum number of elements that can be stored in array.

Example:

1) int rollno[5]; where rollno is a integer array which can store roll number of 5 students.

rollno[0]	rollno[1]	rollno[2]	rollno[3]	rollno[4]

Initialization of one dimensional array:

- After declaration, array must be initialized, otherwise it hold garbage values.

Compile time initialization:

- Fixed values are assigned to the array.

Example:

1) int rollno[5]={1,2,3,4,5}; where rollno is a integer array which can store roll number of 5 students.

1	2	3	4	5
rollno[0]	rollno[1]	rollno[2]	rollno[3]	rollno[4]

Run time initialization:

- Values are assigned to the array at run time.(i.e,)values are assigned during the execution of program

Example1:

```
int  rollno[5];
for( i=0 ; i<5; i++)
{
scanf( “%d”,&rollno[i]);
}
```

i) Algorithm to display the array

Displayarray(a, n)

```
//a is an array of n elements
for  i = 1 to n
    write  a[i]
end for
```

- In the above algorithm, the entire elements of the array a is displayed

ii) Algorithm to find sum of the elements in array

Sumarray(a, n)

```
//a is an array of n elements
sum=0
for  i = 1 to n
     sum = sum + a[ i ]
end for
write  sum
```

- In the above algorithm, each element in the array is added and the result is stored in sum

iii) Algorithm to find maximum element in a given array

maxarray(a, n)

```
// a is an array of n elements
max = a[1]
for  i = 2 to n
    if ( a[ i ] > max )
          max = a[ i ]
    end if
end for
write max
```

- In the above algorithm, At the initial step, a[1] is assigned as max. In for loop, Each of the next element starting from second element is compared with max. If the element is greater than max, value of max changes. After comparing all the elements in array, value in max is displayed.

<u>2. Multi dimensional array:</u>

- An array with more than one subscript is called multi dimensional array.
- Multidimensional arrays are slower in execution than one dimensional arrays.

<u>Syntax:</u>

datatype arrayname[size 1][size 2][size 3]……[size n];

<u>Example:</u>

- int a[3][3] ; where a is a two dimensional array of integer type and can hold 9 elements.
- float b[4][4][4] ; where b is a three dimensional array of float type and can hold 64 elements.

<u>Two dimensional array</u>

- An array with two subscript is called two dimensional array
- Two dimensional array is used to store the data in tabular form (i.e) rows and columns.

<u>Declaration of two dimensional array:</u>

data_type arrayname[row size] [column size] ;   where row size specifies the number of rows and column size specifies the number of columns.

<u>Example:</u>

int a[2][2] ; where a is the two dimensional array of integer type with 2 rows and 2 columns and have 2*2 elements.

int a[3][3] ; where a is the two dimensional array of integer type with 3 rows and 3 columns and have 3*3 elements.

Algorithm to read and display the two dimensional array:

```
for  i = 1 to 2
    for  j = 1 to 2
          read a[ i ] [ j ]
for  i = 1 to 2
    for  j = 1 to 2
          write  a[ i ] [ j ]
```

POINTERS:

- A pointer is a variable which stores the address of other variables.
- Since pointer knows the address, it can directly access those variables.
- Integer pointer can store only address of integer variable , floating pointer can store only address of float variables, character pointer can store only address of character variables.

Declaration of pointer:

Syntax

```
datatype  *pointervariable ;
```

Example:

int *p ; where p is a integer pointer variable which stores the address of integer variable.

float *p ; where p is a float pointer variable which stores the address of float variable.

char *p ; where p is a char pointer variable which stores the address of character variable.

Initialization of pointer variable:

Syntax

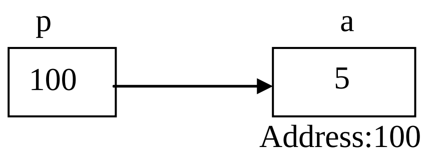
```
pointervariable  = & variable ;
```

Here address of the normal variable is stored in a pointer variable.

Example:

```
int *p ;
int  a=5 ;
```

```
p = &a ;
```



- Here the address of the variable a is stored in the pointer variable p. Since pointer p knows the address of a, p can directly access the variable a.
- *p represents the value pointed by the pointer variable. (or) value at the pointer p

Example program:

```
#include<stdio.h>
int main( )
{
int *p ;
int a=5 ;
p=&a ;
printf(“%d”, p);       // Here p  represents the address stored in the pointer. Hence 100 is printed
printf(“%d”, *p);         // Here *p represents the value at pointer p. Hence value 5 is printed
return 0;
}
```

Output:

```
100     5
```

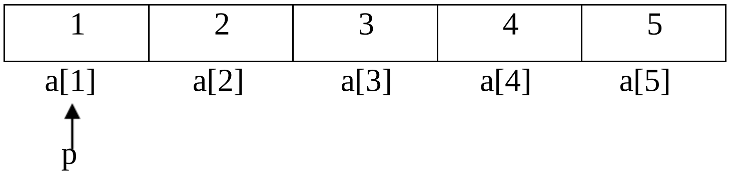
POINTERS AND ARRAYS:

- An array is a collection of elements of same data type.
- A pointer can be made to point to any of the element in an array.

Consider an integer pointer and an integer array.

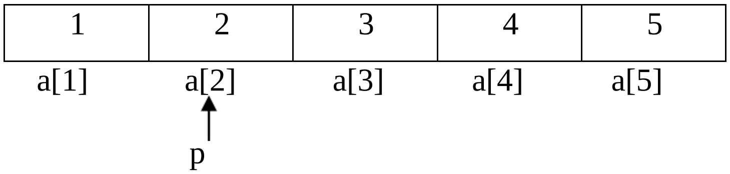
int *p ;

int a[5]={1,2,3,4,5};



1	2	3	4	5
a[1]	a[2]	a[3]	a[4]	a[5]

- ***p = &a[1];***
- The integer pointer p points to 1st element in the array. (i.e) p points to a[1].
- If we increment a pointer, p points to the second element in the array(i.e) p points to a[2].



1	2	3	4	5
a[1]	a[2]	a[3]	a[4]	a[5]

Program:

```
#include<stdio.h>
int main( )
{
int *p ;
int a[5] = {1,2,3,4,5} , i ;
p = &a[1];
for(i=1 ; i<=5 ; i++)
{
printf(“%d”, *p);
p++;
}
return 0;
}
```

Output:

```
1    2    3  4  5
```

POINTER ARRAY (or) ARRAY OF POINTERS

- Pointer array is an array containing pointers as its elements.
- Each pointer in the array points to any variable/array by storing the address of variable/array in pointer.
- The Advantage of a pointer array is that, ***the pointers can be reordered in any manner without moving the data items.***

Example1:

```
#include<stdio.h>
int main( )
{
int  *p[3] ;
int a[3] = {1,2,3} , i ;
for(i=1 ; i<=3 ; i++)
{
p[ i ] = &a[ i ];
}
for(i=1 ; i<=3 ; i++)
{
printf("Values in array is %d", *p[ i ] );
}
return 0;
}
```

Output:

```
1    2    3
```

Example 2:

```
#include<stdio.h>
int main( )
{
int  *p[3] ;
int  a=1, b=2, c=3 ;
p[1]=&a;
p[2]=&b;
p[3]=&c;
for(i=1 ; i<=3 ; i++)
{
printf("Value of variables a b and c are %d", *p[ i ] );
}
return 0;
}
```

Output:

```
1    2    3
```

SEARCHING:

- Searching is a processing of finding a given element from a set of elements.
- Searching is successful if the element is found or else it is unsuccessful

i) LINEAR SEARCH:

- Linear search is performed either in sorted or unsorted list.
- It is a process of searching a given element from the array linearly one by one until the nth element in array.
- If the element is found, the position of that element is returned.
- If the element is not found in array, searching becomes unsuccessful.

ALGORITHM:

Linearsearch(A,n,x)

```
//A is an array which contains n elements
// x is a search element
flag=0
for  i=1 to n
        if  x= = a[ i ]
                write the position of x
                flag = 1
                break
        end if
end for
if  flag = = 0
        Write element is not found
```

Example:

4	7	10	8	3
a[1]	a[2]	a[3]	a[4]	a[5]

Here n = 5, x = 10(search element)

Comparison 1:

i = 1 , 1 < = 5 => True

x = = a[i] => 10 = = a[1]

10 = = 4 => False

Comparison 2:

i = 2 , 2 < = 5 => True

x = = a[i] => 10 = = a[2]

10 = = 7 => False

Comparison 1:

i = 3 , 3 < = 5 => True

x = = a[i] => 10 = = a[3]

10 = = 10 => True

Position of the search element 10 is displayed. (i.e) The position 3 is displayed

Complexity analysis for linear search:

Best case:

Search element matches with the first element

Number of comparisons: 1

Asymptotic complexity : T(n) = O(1)

Worst case:

Search element doesn't exist

Number of comparisons: n

Asymptotic complexity : T(n) = O(n)

Average case:

Search element may be present at any position in array

Expected number of comparisons T(n) = $\sum_{i=1}^{n} Pi\ .\ i$

$P_i = \frac{1}{n}$

$T(n) = \frac{1}{n} * \frac{n(n+1)}{2}$

$T(n) = \frac{n+1}{2}$

T(n) = O(n)

ii) BINARY SEARCH:

- Binary search algorithm is the ***fast searching algorithm***. It works on the principle of divide and conquer.
- To perform binary search, the ***input elements should be in sorted order***.

Binary Search algorithm:

```
BinarySearch(A,x,low,high)
// A is an array of n elements
// low =1 and high= n
while(low < = high)
        mid = (low + high) / 2
    if(x = = a[mid])
        return mid
   else if(x > a[mid])
        low = mid +1
   else
        high = mid -1
end while
```

- **Step1:** In binary search, Initially our search range is the entire array. Hence, low points to 1st position and high points to nth position.
- **Step2:** While the low < = high, find the position of the middle element. Compare of search element with the middle element. If it matches, its position is returned.
- **Step3:** If the search element is greater than the middle element, Search range will become right side of the middle element. Hence low is changed to mid+1. ***Goto step2***
- **Step4:** If the search element is less than the middle element, Search range will become left side of the middle element. Hence high is changed to mid-1. ***Goto step2***

Example:

2	4	6	8	10
a[1]	a[2]	a[3]	a[4]	a[5]

Here n = 5, x = 8(search element)

low = 1, high =5

while (1<=5)

mid = (1+5)/2 => ***mid =3***

2	4	6	8	10
a[1]	a[2]	a[3]	a[4]	a[5]
↑		↑		↑
low		mid		high

if x = = a[mid] => if 8 = = a[3] => (i.e) 8 = = 6 =>False

else if x > a[mid] => 8 > a[3] => (i.e) 8 > 6 => True

Since search element 8 > middle element 6, search range will be on right side of middle element.

Hence, low = mid +1

2	4	6	8	10
a[1]	a[2]	a[3]	a[4]	a[5]
		↑	↑	↑
		mid	low	high

Now *low = 4 and high =5*

While 4 < =5

mid = (4+5) / 2 => **mid = 4**

2	4	6	8	10
a[1]	a[2]	a[3]	a[4]	a[5]
			↑↑	↑
			mid low	high

if x = = a[mid] => 8 = = a[4] => 8 = = 8 => True

Since the element is found, the position of the search element is returned and displayed.

Complexity analysis for binary search:

Best case:

If the search element is present in the middle position, Number of comparison = 1

Asymptotic complexity : T(n) = O(1)

Worst case:

Number of comparisons = $\log_2 n$

Asymptotic complexity : T(n) = O(logn)

Average case:

Number of comparisons = $\log_2 n$

Asymptotic complexity : T(n) = O(logn)

iii) FIBONACCI SEARCH

- The Fibonacci search technique is a method of searching a sorted array using divide and conquer algorithm that narrows down possible locations with the aid of Fibonacci numbers .
- Fibonacci search divides the array into two unequal parts that have sizes that are consecutive Fibonacci numbers

Algorithm:

Let a[1..n] be the input array and element to be searched be x.

1. Find the smallest Fibonacci Number greater than or equal n. Let this number be Fn. Let the two Fibonacci numbers preceding it be Fn-1 and Fn-2 respectively.
2. While the array has elements to be inspected:
   a. The position of the element which is to be compared is found by **i = min(offset +Fn-2 , n).** Initially offset value is set to zero. In subsequent executions, offset is the difference between values of old Fn and new Fn.
   b. Compare x with the element which is at position i. **If** x = = element, return that position.
   c. Else if **x > element**, the value of Fn will be the previous Fibonacci value from the current value. The corresponding Fn-1 and Fn-2 values are changed accordingly and value of i is found. The value in that position is compared. If it matches return that position.
   d. Else if **x < element**, the value of Fn will be the 2nd previous Fibonacci value from the current value. The corresponding Fn-1 and Fn-2 values are changed accordingly and value of i is found. The value in that position is compared. If it matches return that position.
   e. When there is a single element remaining for comparison, (i.e.,) if Fn-1 = 1, compare x with that remaining element. If match, return that position.
   f. Goto step c

Example:

2	4	6	8	10	12
a[1]	a[2]	a[3]	a[4]	a[5]	a[6]

Fibonacci series is 0, 1, 1, 2, 3,5,8,13,21,34……

Fibonacci number **Fn = Fn-1 + Fn-2**

Here n = 6, x = 8(search element)

The smallest Fibonacci number which is greater than n=6 is 8. **Fn =8**

Fn-1 = 5

Fn-2 = 3

Fn-2	Fn-1	Fn	Offset	i=min(offset+Fn-2, n)	a[i]	Comparison
3	5	**8**	0	i = min(0+3, 6) i=3	a[3] =6	8> 6 => Fn changed to prev Fibonacci value in next step
2	3	**5**	3	i = min(3+2, 6) i = 5	a[5]=10	8<10 => Fn changed to 2nd prev Fibonacci value in next step
1	1	**2**	3	i = min(3+1, 6) i = 4	a[4] =8	8 = = 8 => The position of search element 8 is returned.

SORTING:

- Sorting is a process of arranging the set of elements either in ascending or descending order.

i) BUBBLE SORT:

- In bubble sort, ***for n elements, there will be a maximum of n-1 passes*** to make a sorted list of elements. In each pass, each element is compared with its adjacent element in the array. If the kth element is greater than k+1 th element, Swapping is performed on those two elements. By this way all the elements are compared.
- ***k is the number of comparison in each pass. In pass 1, Number of comparison =n-1, In pass 2, number of comparison = n-2*** and so on. *After the end of each pass,* the largest element will be settled at the actual position in array. Hence, *number of comparisons is reduced by 1 after each pass.*

Algorithm:

```
Bubblesort(a, n)
// a is an array of n elements
   for  j = 1 to n-1
         for k= 1 to n – j
              if (a[k] > a[k+1] )
                     swap (a[k] , a[k+1] )
              end if
         end for
   end for
   write the array a
```

Example:

Pass 1: j=1

a[1]	**a[2]**	**a[3]**	**a[4]**	**a[5]**
6	4	8	5	2
a[1] > a[2] => 6 > 4 => True => Swap done				
4	6	8	5	2
a[2] > a[3] => 6>8 => False				
a[3] > a[4] => 8>5 => True => Swap done				
4	6	5	8	2
a[4] > a[5] => 8>2=> True => Swap done				
4	6	5	2	8

Pass 2: j=2

a[1]	**a[2]**	**a[3]**	**a[4]**	**a[5]**
4	6	5	2	8
a[1] > a[2] => 4>6 => False				
4	6	5	2	8
a[2] > a[3] => 6>5 = True => Swap done				
4	5	6	2	8
a[3] > a[4] => 6>2 => True => Swap done				
4	5	2	6	8

Pass 3: j=3

a[1]	a[2]	a[3]	a[4]	a[5]
4	5	2	6	8
a[1] > a[2] => 4>5 => False				
4	5	2	6	8
a[2] > a[3] => 5>2 = True => Swap done				
4	2	5	6	8

Pass 4: j=4

a[1]	a[2]	a[3]	a[4]	a[5]
4	2	5	6	8
a[1] > a[2] => 4>2 => True=> Swap done				
2	4	5	6	8

Elements in sorted order:

a[1]	a[2]	a[3]	a[4]	a[5]
2	4	5	6	8

ii) Selection sort:

- Selection sort algorithm selects the smallest element in array and exchange it with the 1st element. From the remaining set of elements, selects the smallest element and exchange it with 2nd element and so on.
- Initially k is the position of the minimum element. The kth element is compared with the remaining elements in array and the position of actual minimum element is found. If that position is not equal to k, then swapping of those two elements is done. This process is continued from k = 1 to n-1. By this way, entire array is sorted.

Algorithm:

SELECTIONSORT(A,N)

```
// a is an array of n elements
for  k = 1 to n -  1
        min = k
        for j = k +1 to n
            if( a[ j ] < a[ min ] )
                min =j
            end if
        end for
        if( min ! = k)
                Swap ( a[k] , a[min])
        end if
end for
```

Example:

k=1, min = 1

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]
↑ min				

j=2 , a[2] < a[1] => 4 < 6 => True

min = 2

j=3, a[3] < a[2] => 8<4 => False

j=4, a[4] < a[2] => 5<4 => False

j=5 , a[5] < a[2] => 2<4 => True

min =5

since 1 ! = 5

swap a[1] and a[5]

2	4	8	5	6
a[1]	a[2]	a[3]	a[4]	a[5]

k=2, min =2

2	4	8	5	6
a[1]	a[2]	a[3]	a[4]	a[5]
	↑ min			

j=3, a[3] < a[2] => 8 < 4 => False

j=4, a[4] < a[2] => 5 < 4 => False

j=5, a[5] < a[2] => 6 < 4 => False

Since 2 = = 2 , Swap not performed

2	4	8	5	6
a[1]	a[2]	a[3]	a[4]	a[5]

k=3, min =3

2	4	8	5	6
a[1]	a[2]	a[3]	a[4]	a[5]
		↑ min		

j=4, a[4] < a[3] => 5< 8 => True

min = 4

j=5, a[5[< a[4] => 6 < 5 => False

since 4 ! = 3

swap a[3] and a[4]

2	4	5	8	6
a[1]	a[2]	a[3]	a[4]	a[5]

k=4, min =4

2	4	5	8	6
a[1]	a[2]	a[3]	a[4]	a[5]
			↑ min	

j=5, a[5] < a[4] => 6< 8 => True

min = 5

since 5 ! = 4

swap a[4] and a[5]

2	4	5	6	8
a[1]	a[2]	a[3]	a[4]	a[5]

iii) INSERTION SORT:

- In insertion sort, the key element is chosen and that key element will be inserted at the appropriate position by comparing all the elements to left of a key element.
- The first key element chosen will be 2nd element, next is 3rd element and it goes on upto n.(i.e) chosing a key element ranges from 2 to n.

Algorithm:

```
Insertionsort(a,n)
// A is an array of n elements
for  j= 2 to n     // j represents the position of key element
         key = a[j]
         i = j – 1
        while ( i> 0 and key < a[ i ] )
              a[ i + 1 ] = a[ i ]
              i = i – 1
        end while
        a[ i + 1] = key
end for
write  a
```

Example:

j=2, key = 4

i = 1

6	4	8	5
a[1]	a[2]	a[3]	a[4]
↑ i			

while 1>0 and 4<6 => True

a[2]=a[1] => a[2] = 6

i=0

	6	6	8	5
0	a[1]	a[2]	a[3]	a[4]
↑ i				

while 0 > 0 => False

a[0+1] = 4.

4	6	8	5
a[1]	a[2]	a[3]	a[4]

j=3, key =8

i=2

while 2>0 and 8<6 => False

a[i+1] = key => a[3] = 8 => No changes here

j=4, key =5

i=3

4	6	8	5
a[1]	a[2]	a[3]	a[4]
		↑ i	

while 3>0 and key < a[3] => 5 < 8 => True

a[3+1]=a[3] => a[4]=8

4	6	8	8
a[1]	a[2]	a[3]	a[4]

i=2,

4	6	8	8
a[1]	a[2]	a[3]	a[4]
	↑ i		

while 2> 0 and 5 < a[2] => 5 < 6 => True

a[2+1] = a[2] => a[3]=6

4	6	6	8
a[1]	a[2]	a[3]	a[4]

i=1,

4	6	6	8
a[1]	a[2]	a[3]	a[4]
↑ i			

while 1>0 and 5< a[1] => 5< 4 => False

a[2] = key => a[2] = 5

4	5	6	8
a[1]	a[2]	a[3]	a[4]

iv) SHELL SORT:

Step1: The current element element is compared with the another element which is at certain distance and swapping is performed, if necessary.

Step 2: If swapping is not performed with a current distance, distance is reduced by half and step 1 continues until distance becomes zero.

Algorithm:

Shellsort(a , n)

```
// a is an array of n elements
        dist=n/2
        repeat
                repeat
                        swap=0
                        for i=1 to n-dist
                                if(a[i]>a[i+dist])
                                   swap( a[i] , a[i+dist])
                                   swap=1
                                end if
                        end for
                until swap=1
        dist = dist/2
        until dist!=0
write a
```

Example:

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]

dist = 5/2 **=> dist = 2**

swap=0

i =1

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]

a[1] > a[3] => False

i =2

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]

a[2] > a[4] => False

i=3

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]

a[3] > a[5] => True => Swap a[3] and a[5] , ***swap =1***

6	4	2	5	8
a[1]	a[2]	a[3]	a[4]	a[5]

while(swap) => while(1)
Set swap =0
i =1

6	4	2	5	8
a[1]	a[2]	a[3]	a[4]	a[5]

a[1] > a[3] => True => Swap a[1] and a[3], ***swap=1***

2	4	6	5	8
a[1]	a[2]	a[3]	a[4]	a[5]

i=2 and i=3

2	4	6	5	8
a[1]	a[2]	a[3]	a[4]	a[5]

a[2] > a[4] => False   a[3]> a[5] => False

While(swap) => while(1)

Set swap=0

i =1, i=2 and i=3

2	4	6	5	8
a[1]	a[2]	a[3]	a[4]	a[5]

a[1]> a[3]=>F

a[2] > a[4] => F   a[3]> a[5] => F

while(swap) => while(0)
dist= 2/2 => **dist = 1**
while(distance) => while(1)
swap=0
i=1,i=2, i=3

2	4	6	5	8
a[1]	a[2]	a[3]	a[4]	a[5]

a[1]>a[2] =>F a[2]>a[3] =>F   a[3]> a[4] => True => *Swap a[3] and a[4]* , ***swap =1***

2	4	5	6	8
a[1]	a[2]	a[3]	a[4]	a[5]

while(swap) => while(1)
swap =0
i=1, i=2, i=3, i=4

2	4	5	6	8
a[1]	a[2]	a[3]	a[4]	a[5]

a[1]>a[2]=>F   a[2]>a[3]=>F a[3]>a[4]=>F a[4] > a[5] => F

while(swap) => while(0) => False
dist = 1/2 => **dist = 0**
while(dist) => while(0) => False

Sorted elements:

2	4	5	6	8
a[1]	a[2]	a[3]	a[4]	a[5]

v) QUICK SORT:

- Quick sort is otherwise called as partition exchange sorting
- It uses divide and conquer method.

Principle:

1. Choose any number in the array and name it as partition element or pivot element (P). For simplicity we take the first number as the ***pivot element P***.
2. With respect to P, divide the array into two partitions i.e., left, right partition. The numbers which are less than P are placed in the left side of P. And the numbers which are greater than P are placed in the right side of P, for eg., *If the position of P is J,* then it satisfies the following conditions.

   i) Each number in the position from 1 to J-1 are less than or equal to P.

   ii) Each number in the position from J+1 to n are greater than or equal to P.

   iii) The Pivot element is placed in its proper position.
3. We can repeat the steps 1 and 2 for the left and right partition.

Algorithm:

qsort(left,right)

```
// left points to 1st element in array and right points to nth element in array
    if(left<right)
            j=partition(a,left,right);
            qsort(left,j-1);
            qsort(j+1,right);
    end if
```

partition(a, left, right)

```
if(left<right)
        pivot=a[left];
        i=left;
        j=right+1;
        repeat
                repeat
                        i++
                until  a[i]<pivot and i<=right
                repeat
                        j--
                until  a[j]>pivot and j>=left
                if(i<j)
                        interchange(a,i,j)
                end if
        until i<j
        interchange(a,left,j);
        return j;
```

Example:

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]

left =1 and right =5
qsort(1,5)
1<5

j = partition(a,1,5)

partition(1,5)
1<5
Pivot = a[1] => **PIVOT = 6**

6	4	8	5	2	
a[1]	a[2]	a[3]	a[4]	a[5]	6
↑ i					↑ j

Incrementing i
i=2, check a[2] < pivot => 4 < 6 => True => increment i
i=3, check a[3] < pivot => 8 < 6 => False => Stop incrementing i
Decrement j
j=5, check a[5]>pivot => 2>6 => False => stop decrementing j

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]
		↑ i		↑ j

if 3 < 5 => True
 interchange a[3] and a[5]

6	4	2	5	8
a[1]	a[2]	a[3]	a[4]	a[5]
		↑ i		↑ j

Incrementing i, we get
i=4, check a[4] < pivot => 5<6 => True => increment i
i=5, check a[5] < pivot => 8 < 6 => False => Stop incrementing i
Decrementing j, we get
j=4, check a[4] > pivot => 5>6 => False => Stop decrementing j

6	4	2	5	8
a[1]	a[2]	a[3]	a[4]	a[5]
			↑ j	↑ i

if 5 < 4 => False
Interchange a[left] and a[j] and then return the position of j. Here left =1, so a[1] and a[4] is interchanged and this function **partition(1,5) returns 4.**

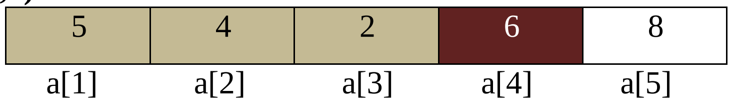
5	4	2	6	8
a[1]	a[2]	a[3]	a[4]	a[5]
			↑ j	↑ i

j = 4

qsort(1, 3) , qsort(5,5)

We take,

qsort(1,3)

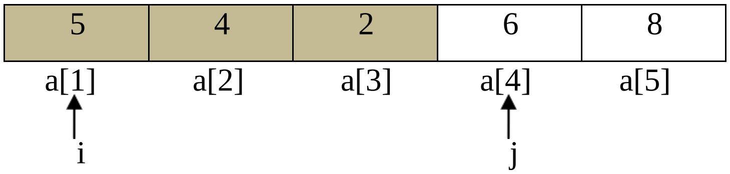


1<3

j= partition(a,1,3)

partition(a,1,3) // left =1, right =3

pivot = a[1] => **PIVOT = 5**



Incrementing i, we get

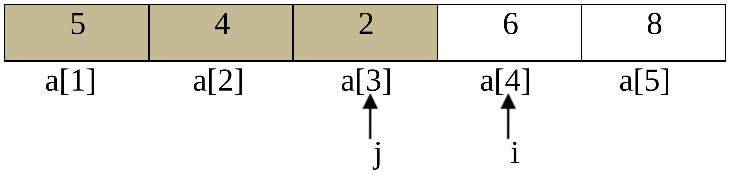
i=2, check 4 < 5 => True => increment i

i=3, check 2 < 5 => True => increment i

i=4, check 6 < 5 => False => Stop incrementing i

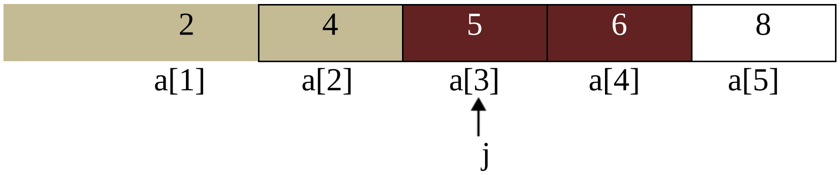
Decrement j

j=3, check 2>5 => False => stop decrementing j



if 4< 3 => False

interchange a[left] and a[j]. (i.e) interchange a[1] and a[3] and the position of j is returned.



j = 3

qsort(1, 2) and qsort(4,3)

we take

qsort(1,2)

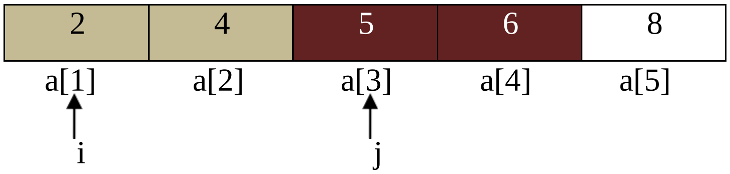
1<2

j= partition(a,1,2)

partition(a,1,2) // Here **left=1 and right =2**

1<2

Pivot =a[1] => **PIVOT = 2**



Incrementing i, we get

i=2, check 4 < 2 => False => Stop incrementing i

Decrementing j, we get,
j=2, check 4>2 => True => Decrement j
j=1, check 2>2 => False => Stop decrementing j

2	4	5	6	8
a[1]	a[2]	a[3]	a[4]	a[5]
↑ j	↑ i			

if 2<1 => False
interchange a[first] and a[j]. Here both are a[1]. Hence no change occurs.
Position of j is returned which is 1.

j=1

qsort(1,0) => 1<0 => False
qsort(2,2) => 2<2 => False
Taking qsort(4,3) and qsort(5,5),
qsort(4,3) => 4<3 => False
qsort(5,5) => 5<5 => False

Elements in sorted order

2	4	5	6	8
a[1]	a[2]	a[3]	a[4]	a[5]

vi) MERGE SORT

- Merge sort follows divide and conquer method.
- It involves partitioning a set of elements based on middle element recursively until we get a single element.
- After that, the partitions are merged by sorting the elements in each partition in recursive manner thereby getting a sorted list of elements.

Algorithm:

```
// partition performed upto 1 or 2 elements
partition(low, high)
// low points to 1st element in array and high points to nth element in array
        if(low<high)
                mid=(low+high)/2;
                partition(low,mid);
                partition(mid+1,high);
                msort(low,mid,high);
        end if
// Performs sorting and merging
msort(low, mid, high)
        m=low;
        i=low;
        j=mid+1;
        // compare two partition
        while(i<=mid && j<=high)
                if(a[i]<a[j])
                        b[m]=a[i];
                        i=i+1
                else
                        b[m]=a[j];
                        j=j+1;
                m=m+1;
        end while
        //append remaining element
        if(i<=mid)
                for(k=i;k<=mid;k++)
                        b[m]=a[k];
                        m=m+1;
                end for
        else if(j<=high)
                for(k=j;k<=high;k++)
                        b[m]=a[k];
                        m=m+1;
        end if
        // copy to a[ ]
        for(k=low;k<=high;k++)
                a[k]=b[k];
        end for
```

Example:

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]

- Partition is performed until we get the single element.

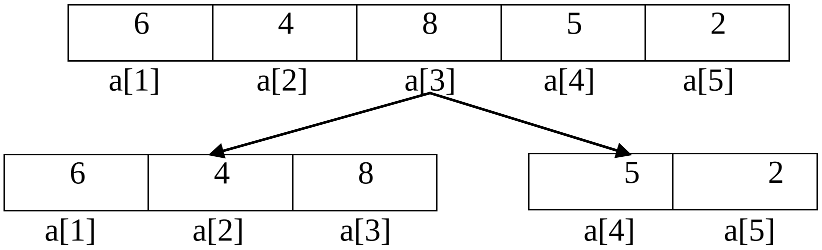
partition(1,5)

low=1, high=5

1<5

mid = (1+5)/2 =>**mid = 3**

partition(1, 3) , partion(4, 5)



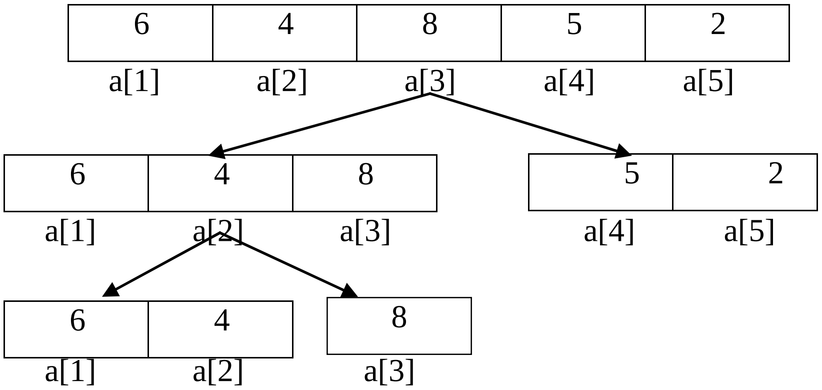
partition(1,3)

low=1, high=3

1<3

mid = (1+3)/2 => **mid =2**

partition(1,2) , partition(3,3)

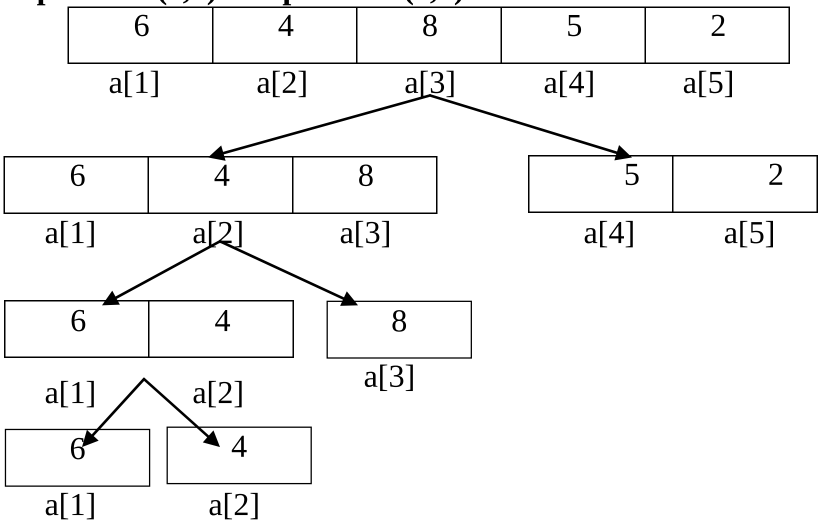


partition(1,2)

low = 1 , high =2

mid =(1+2)/2 => **mid = 1**

partition(1,1) and partition(2,2)



partition(1,1) => 1< 1 => **False, since there is only one element**
partition(2,2) => 2< 2 => **False**
partition(3,3) => 3<3 => **False**

- **Merging of partitions:**

Taking **partition(1,2)**
In this low =1 , mid=1 and high =2

6	4
a[1]	a[2]
↑	↑
i	j

msort(1,1,2)
When comparing the two elements, 4<6, Hence, 4 is stored in b array and j moves to next position 3.

4	
b[1]	b[2]
	↑
	m

Since there is no element at position 3, the remaining element is stored in b array.

4	6
b[1]	b[2]

Copy the elements of b array to a array

4	6
a[1]	a[2]

Taking, partition(1,3)
In this, low =1, mid=2 and high =3

4	6	8
a[1]	a[2]	a[3]
↑		↑
i		j

4 < 8 => True=> 4 is stored in 1st position of b array. i incremented to 2.
6<8 => True => 6 is stored in 2nd position of b array. i incremented to 3.
Since there is no element to compare further, the remaining element 8 is stored in 3rd position in b array.

4	6	8
b[1]	b[2]	b[3]

Copy the elements of b array to a array

4	6	8
a[1]	a[2]	a[3]

Taking **partition(4,5)**
msort(4,4,5)

5	2
a[4]	a[5]
↑	↑
i	j

When comparing the two elements, 2<5, Hence, 2 is stored in b array and j moves to next position 6.

2	
b[4]	b[5]
	↑ m

Since there is no element at position 3, the remaining element is stored in b array.

2	5
b[4]	b[5]

Copy the elements of b array to a array

2	5
a[4]	a[5]

Taking **partition(1,5)**
msort(1,3,5)

4	6	8	2	5
a[1]	a[2]	a[3]	a[4]	a[5]
↑ i			↑ j	

When comparing the two elements a[1] and a[4], a[4]< a[1] => 2<4 and hence **2 is stored in 1st position of b array** and j incremented to next position 5

4	6	8	2	5
a[1]	a[2]	a[3]	a[4]	a[5]
↑ i				↑ j

When comparing the two elements a[1] and a[5], a[1]< a[5] => 4<5 and hence **4 is stored in 2nd position of b array** and i incremented to next position 2

4	6	8	2	5
a[1]	a[2]	a[3]	a[4]	a[5]
	↑ i			↑ j

When comparing the two elements a[2] and a[5], a[5]< a[2] => 5<6 and hence **5 is stored in 3rd position of b array** and j incremented to next position 6

4	6	8	2	5	
a[1]	a[2]	a[3]	a[4]	a[5]	6
	↑ i				↑ j

Since there is no elements at position 6, the remaining elements in 1st partition is stored in 4th and 5th position of b array respectively.

Finally the elements of b array is copied to a array

2	4	5	6	8
a[1]	a[2]	a[3]	a[4]	a[5]

vii) RADIX SORT

- Radix sort is an non-comparative sorting algorithm
- Radix sort is also called as bucket sort

Algorithm:

```
Find the maximum element in the array
Find the number of digits in that element, say w
for  k = 1 to w  // k represents the number of digits in that number
    for  i = 1 to n
            L = (a[ i ] / pow(10, k-1) ) % 10
            Enqueue( Q[L] , a[ i ] )
     end for
     set j =1
     for  i = 0 to 9
         while( Q[ i ] !=empty)
             a[ j++ ] = dequeue( Q[ i ] )
          end while
     end for
end for
```

Example:

Consider the array

150	213	002	041
a[1]	a[2]	a[3]	a[4]

The maximum element in the array is 213.

Number of digits in this number is 3. (i.e) w=3

Q[0]	150			
Q[1]	041			
Q[2]	002			
Q[3]	213			
Q[4]				

.
.
Q[9]

Traceout:

i) Storing the elements in array to the bucket based on the unit digit.

k=1, 1<=3

i =1
L = (a[1] / pow(10,0)) % 10
L = 150 % 10 => **L =0**
Q[0]=150
i =2
L = 213 % 10 => **L =3**
Q[3]=213
i =3
L = 002 % 10 => **L =2**
Q[2]=002
i =4
L = 041 % 10 => **L =1**
Q[1]=041

Removing elements from bucket and store in array

150	041	002	213
a[1]	a[2]	a[3]	a[4]

ii) Storing the elements in array to the bucket based on the tenth digit.

Q[0]	002			
Q[1]	213			
Q[2]				
Q[3]				
Q[4]	041			
Q[5]	150			

.
.
Q[9]

k=2, 2<=3

i =1, a[1]=1**5**0
L = (a[1] / pow(10,1)) % 10
L = (150/10) % 10 => L = 15 % 10 => **L=5**
Q[5]=150
i =2, a[2]=0**4**1
L =4
Q[4]=041
i =3, a[3]=0**0**2
L = 0
Q[0]=002
i =4, a[4]=2**1**3
L =1
Q[1]=213

Removing elements from bucket and store in array

002	213	041	150
a[1]	a[2]	a[3]	a[4]

iii) Storing the elements in array to the bucket based on the hundredth digit.

Q[0]	002	041		
Q[1]	150			
Q[2]	213			
Q[3]				
Q[4]				

.
.
Q[9]

k=3, 3<=3

i =1, a[1]=**0**02

L = (a[1] / pow(10,1)) % 10

L = (002/100) % 10 => L = 0 % 10 => **L=0**

Q[0]= 002

i =2, a[2]=**2**13

L =2

Q[2]=213

i =3, a[3]=**0**41

L = 0

Q[0]=041

i =4, a[4]=**1**50

L =1

Q[1]=150

Removing elements from bucket and store in array

002	041	150	213
a[1]	a[2]	a[3]	a[4]

viii) HEAP SORT:

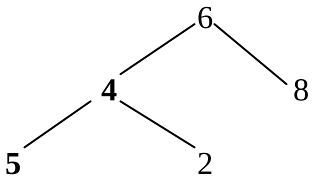
```
// CREATES HEAP
heapify(a,n)
        for  i=n/2 to 1
                adjust(a,i,n);
        end for
// ADJUST THE HEAP TO SATISFY HEAP PROPERTY
adjust(a,i,n)
        j=2*i;
        key=a[i];
        while(j<=n)
                if((j<n)&&(a[j]<a[j+1]))
                        j=j+1
                end if
                if(key>a[j])
                        break
                end if
                a[j/2]=a[j]
                j=2*j
        end while
        a[j/2]=key

// PERFORMS SORTING OPERATION
hsort(a, n)
        heapify(a,n);
        for i=n to 2
                swap(a[i],a[1])
                adjust(a,1,i-1)
        end for
```

Example:

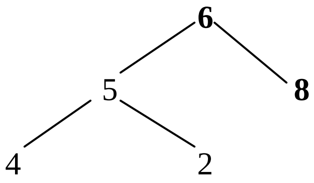
In max heap, the parent node must be larger than its child nodes.

6	4	8	5	2
a[1]	a[2]	a[3]	a[4]	a[5]



```
heapify(a,5)
i=2, 2>=1
   adjust(a,2,5)
adjust(a,2,5)
j=2*2 => j=4
key = a[2] => key = 4
while(4<=5)
     if  4<5 and a[4] < a[5]  => False
     if  4> 5 => False
     a[2] = a[4] => a[2] = 5
     j=2*j => j=8
while(8<=5) => False
a[4] =4
```

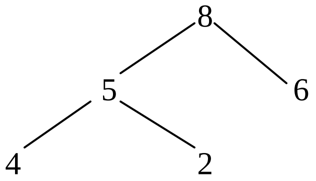
6	5	8	4	2
a[1]	a[2]	a[3]	a[4]	a[5]



```
i=1, 1>=1
    adjust(a,1,5)
adjust(a,1,5)
j=2*1 => j=2
key = a[1] => key =6
while(2<=5)
    if(2<5 and a[2]<a[3]) => True
        j=3
    if  6> a[3] => 6>8 => False
    a[1]=a[3] => a[1]=8
    j=2*3=> j=6,
while(6<=5) => False
a[3]=6
```

8	5	6	4	2
a[1]	a[2]	a[3]	a[4]	a[5]

Hence we created the heap such that, the parent node is higher than its child nodes.



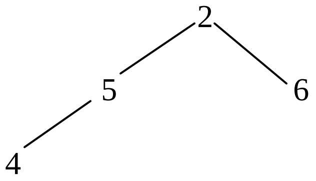
Heap sort:

i=n => i = 5 , 5>=2
Swap(a[5], a[1])

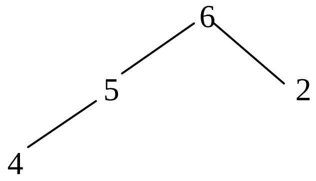
Swap the last element in heap with the root element 8. Then, delete the last element in heap which is 8 and then place it in array.

2	5	6	4	**8**
a[1]	a[2]	a[3]	a[4]	a[5]

Adjust remaining elements a[1] to a[4]



=>

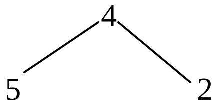


6	5	2	4	**8**
a[1]	a[2]	a[3]	a[4]	a[5]

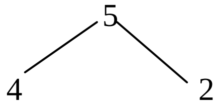
i=4, 4>=2
Swap(a[4] , a[1])
Swap the last element in heap with the root element 6. Then, delete the last element in heap which is 6 and then place it in array.

4	5	2	**6**	**8**
a[1]	a[2]	a[3]	a[4]	a[5]

Adjust remaining elements a[1] to a[3]



=>

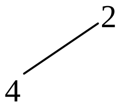


5	4	2	6	**8**
a[1]	a[2]	a[3]	a[4]	a[5]

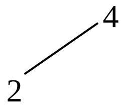
i=3, 3>=2
Swap(a[3] , a[1])
Swap the last element in heap with the root element 5. Then, delete the last element in heap which is 5 and then place it in array.

2	4	**5**	**6**	**8**
a[1]	a[2]	a[3]	a[4]	a[5]

Adjust remaining elements a[1] to a[2]



=>



4	2	**5**	**6**	**8**
a[1]	a[2]	a[3]	a[4]	a[5]

i=2, 2>=2

Swap(a[2] , a[1])

Swap the last element in heap with the root element 4. Then, delete the last element in heap which is 4 and then place it in array.

2	**4**	**5**	**6**	**8**
a[1]	a[2]	a[3]	a[4]	a[5]

i=1, 1>=2 => **False**

Sorted list of elements:

2	**4**	**5**	**6**	**8**
a[1]	a[2]	a[3]	a[4]	a[5]

TIME COMPLEXITY OF THE SEARCHING AND SORTING ALGORITHMS

Technique	Best case	Average case	Worst case
Searching			
1. Linear search	Ω (1)	θ (n)	O(n)
2. Binary search	Ω (1)	θ (logn)	O (logn)
Sorting			
1. Merge sort	Ω (nlogn)	θ (nlogn)	O(nlogn)
2. Quick sort	Ω (nlogn)	θ (nlogn)	O(n^2)
3. Heap sort	Ω (nlogn)	θ (nlogn)	O(nlogn)
4. Bubble sort	Ω (n)	θ (n^2)	O(n^2)
5. Selection sort	Ω (n^2)	θ (n^2)	O(n^2)
6. Insertion sort	Ω (n)	θ (n^2)	O(n^2)
7. Shell sort	Ω (nlogn)	θ (n^2)	O(n^2)
8. Radix sort	Ω (n)	θ (n)	O(n)

CST35 - DATA STRUCTURES
UNIT – 1 TWO MARKS

1. **Define Data structures. (or) What is the need of data structure?**
   - Data Structure is a method of organizing large amount of data more efficiently, such that any operation on that data becomes easy. It also represents the logical relationship between the elements.
2. **What are the types of data structure?**

   i) Primary Data Structures
   - Constants
   - Pointers

   ii) Secondary Data Structures
   - Static data structures
     - Array
     - Structures
   - Dynamic data structures
     - Linear data structures
       - Stack, Queue, Linked list
     - Non-linear data structures
       - Trees, Graph
3. **What is a primary data structure?**
   - It is a basic data structures that directly operate upon the machine instructions.
   - All the basic constants (i.e integers, floating point numbers character constants, string constants) and pointers are considered as primary data structure.
4. **Define static data structures**
   - A data structures formed when the number of data items is known in advance is referred as static data structure or fixed size data structure.
   - Eg. Arrays ,structures.
5. **Define dynamic data structures**
   - A data structure formed when the number of data items are not known in advance is known as dynamic data structure or variable size data structure
6. **Define linear data structure**
   - Linear data structure is data structure having a linear relationship between its adjacent elements. Stack, Queue, Linked lists are examples of linear data structure.
7. **Define non-linear data structure.**
   - Non-linear data structures are data structures that don’t have a linear relationship between its adjacent elements but have a hierarchical relationship between the elements. Trees and graphs are examples of nonlinear data structures.
8. **Define algorithm**
   - Algorithm is a sequence of unambiguous instructions to solve a problem.
   - A problem may have many solutions/algorithm. We need to select the best algorithm which is suitable for a given problem.
   - After the algorithm for the problem is chosen, it is converted to program.

9. What are the properties of algorithm ?

- **Input:** Algorithm may take zero or more input
- **Output:** Algorithm always produces a single output
- **Definiteness:** Every instruction in an algorithm must be clear and unambiguous
- **Finiteness :** For all possible inputs, algorithm must produce result in finite steps
- **Effectiveness:** The algorithm must be feasible

10. List out the phases of program creation?

- Requirements
- Design
- Analysis
- Refinement and Coding
- Verification

11. What are the basic operations of data structures?

- Traversing
- Searching
- Sorting
- Insertion
- Deletion
- Merging

12. State the need for algorithm analysis

- Since a problem may have many algorithms, Comparing one algorithm with another is performed based on the *amount of computing resources used by algorithm*. By analyzing the algorithm, best algorithm for the problem can be chosen based on *time complexity and the space complexity.*

13. What is an array?

- Array is a collection of elements of same data type that are stored under a common name.
- The elements of array are stored in contiguous memory location.

14. Why do we use a Multidimensional array?

- A multidimensional array can be useful to organize subgroups of data within an array. In addition to organizing data stored in elements of an array, a multidimensional array can store memory addresses of data in a pointer array and an array of pointers.
- Syntax:
  - datatype arrayname[size 1][size 2][size 3]……[size n];
- Example:
  - int a[4][4][4] ; where a is a three dimensional array of int type and can hold 64 elements.

15. What are the various algorithmic notations?

Name of the algorithm : Each algorithm is given by its name

Data required for the algorithm: Data required for executing an algorithm is given as arguments to the algorithm

Comments : It describes the data passed to the algorithm or the steps of an algorithm

Steps of the algorithm: The sequence of steps which is performed by the algorithm

***Read* =>** Used to read the values

***Write* =>** Used to display the values

16. Define time complexity with a suitable example.

- It is the total amount of time required by an algorithm to complete its execution. It is also called as running time or execution time of an algoritm
- Time complexity depends on input data, read and write speed of machine etc
- **Example:** for i=1 to n

```
for  i=1 to n
        x=x+1
end  for
```

Number of times loop executed = ***n*** => **T(n) = O(n)**

17. What is the use of space complexity?

- It is the amount of memory resources utilized to execute an algorithm
- The memory is required to store program instructions, store variables, constants etc

18. Mention the use of pointer array.

- The Advantage of a pointer array is that, ***the pointers can be reordered in any manner without moving the data items.***
- By storing the address of the array element in a pointer, we can directly access that element in the array

19. What is an multidimensional array?

- An array with more than one subscript is called multi dimensional array.
- Multidimensional arrays are slower in execution than one dimensional arrays.

Example:

- int a[4][4][4] ; where a is a three dimensional array of int type and can hold 64 elements.

20. What do you meant by sorting?

- Sorting a sequence of elements involves ***rearranging them in either ascending or descending order*** depending upon the relationship among the elements.

21. What is the necessity for sorting technique?

- All data processing requires accessing records efficiently and quickly.
- Search techniques are most efficient only when the data items are sort according to some specified keys.
- If the list/file is not sorted, searching a record takes more time when the list/file is large.

22. What are the factors to be considered while choosing a sorting technique?

- Running time of the sorting technique
- Memory space needed for the sorting technique
- Number of comparisons required for sorting the list
- Programming time

23. What is meant by internal sorting?

- An internal sort is a sorting process that takes place entirely within the main memory of a computer.
- This is possible whenever the data to be sorted is small.
- For sorting larger data, it may be necessary to hold only small amount of data in memory at a time
- So, The rest of the data is normally held on some larger memory like a hard-disk.

24. When sorting method is said to be stable? Give example for stable and unstable sorting.

- A sorting method is said to be stable when it has minimum number of swaps
- **Stable sorting techniques:** Bubble sort, Insertion sort, Selection sort, Merge sort
- **Unstable sorting techniques:** Quick sort, Heap sort, Shell sort, Radix sort

25. What do you meant by searching?

- Searching is an operation to find the location of the given data in the array .
- Searching is said to be successful if the data is present otherwise said to be unsuccessful.
- There are three types of searching:
    - Linear search
    - Binary search
    - Fibonacci search

26. What is meant by linear search?

- Linear search is otherwise called as sequential search .
- Let a be the array of N numbers and X be the element to be searched .
- Our aim is to find whether X is present or not by comparing all the elements in array from 1 to N.

27. What is meant by binary search?

- Binary search is implemented using sorted array. It splits the array into two sub arrays based on middle element recursively.
- If the search element is a middle element, its position is returned or if it is less than the middle element, searching is performed on the left side of middle element, else, searching is performed on the right side of middle element.

28. What is meant by Fibonacci search ?

- The Fibonacci search technique is a method of searching a sorted array using divide and conquer algorithm that narrows down possible locations with the aid of Fibonacci numbers .
- Fibonacci search divides the array into two parts that have sizes that are consecutive Fibonacci numbers

29. What do you mean by performance analysis of an algorithm?

- It is a process of making evaluative judgement about the algorithms
- Performance analysis consider mainly execution speed, memory consumption of that algorithm
- Since there are multiple alternative algorithms will be available to solve a problem, we analyse and select a best suitable algorithm based on our requirements.

30. Compare linear search and binary search.

Linear Search	Binary Search
It is not as fast as binary search.	It is faster than linear search.
The elements need not be inputed in sorted order.	The elements must be inputed in sorted order.
Searching is performed in a sequential manner	Searching is performed based on the middle element in the array, recursively.
Time complexity is more	Time complexity is less

31. Define Big oh(O) notation

Big Oh is defined as

The function f(n) = O(g(n)) if and only if there exists constants c and n_0, such that

$f(n) \leq c * g(n)$ for all $n \geq n_{0.}$ Here f(n) and g(n) are computation time of the algorithm

Example:

$f(n) = 3n + 2 \leq 4*n$ for all $n \geq 2$, where $f(n) = 3n+2$, $c = 4$, $g(n) = n$ and $n_0 = 2$

32. Define Omega(Ω) notation

Omega is defined as

The function f(n) = Ω(g(n)) if and only if there exists constants c and n_0, such that $f(n) \geq c * g(n)$ for all $n \geq n_0$. Here f(n) and g(n) are computation time of the algorithm

Example:

$f(n) = 3n + 2 \geq 3*n$ for all $n \geq 1$, where f(n) = 3n+2, c = 3 , g(n) =n and $n_0 = 1$

33. Define Theta(Θ) notation.

Theta is defined as

The function f(n) = Θ(g(n)) if and only if there exists constants c1, c2 and n_0, such that $c1 * g(n) \leq f(n) \leq c2 * g(n)$ for all $n \geq n_0$. Here f(n) and g(n) are computation time of the algorithm

Example:

$3*n \leq 3n + 2 \leq 4*n$ for all $n \geq 2$, where c1=3, c2=4, g(n) =n, f(n) = 3n+2, $n_0 = 2$

34. Define heap.

- A heap is defined to be a complete binary tree with the property such that each parent node is larger than its child nodes.
- The root node in heap is the largest element in the tree.

35. Define max heap.

- A max heap referred as descending heap of size n is defined as a complete binary tree of n nodes such that the data in the each parent node is greater than its child nodes.
- The root node in heap is the largest element in the tree.

36. Define min heap.

- A min heap referred as ascending heap of size n is defined as a complete binary tree of n nodes such that the data in the each parent node is lesser than its child nodes.
- The root node in heap is the smallest element in the tree.

37. How sorting is performed in heap sort?

Step1: Set of elements in array is adjusted according to the max heap property.

Step2: Then the root element is swapped with the last element. Then the last element is deleted from heap and stored in array.

Step3: Then, Repeat the **step 1** until heap has no element

38. Which is the fastest sorting technique & why?

- Quick sort technique is the fastest sorting technique.
- The advantage is that, The data can be moved to great distance in one move to place it in its exact position as in final list which reduces unnecessary swaps.

39. Which is the fastest sorting technique & why?

- It follows divide and conquer technique.
- It divides the array into sub-array based on mid position recursively, until the we get a single element.
- It, then conquers each sub-array by merging each sub-array recursively thereby forming the sorted array

UNIT – 2

LINEAR DATA STRUCTURES:

- In linear data structure, elements will have a linear relationship with the adjacent elements.
- Example: Stack, Queue, Linked list

STACK:

- Stack is an ordered collection of homogeneous elements, where insertion and deletion are performed only at one end(top of the stack).
- Insertion of elements in stack is called PUSH
- Deletion of elements in stack is called POP
- Stack follows LIFO(Last In First Out) strategy. (i.e) The element which is pushed last will be popped out first
- Stack act as a Static Data structure if it is implemented using arrays
- Stack act as a Dynamic Data structure if it is implemented using Linked list
- Initially top = -1, which implies that stack is empty

Stack[4]	
Stack[3]	
Stack[2]	
Stack[1]	
Stack[0]	

Representation of stack of size 5 as an array

OPERATIONS OF STACK

i) Push operation:

- Push operation is used to insert the element at top of the stack.
- Before performing push operation, it is necessary to check the stackfull condition, because if a stack is full, push operation cannot be performed.
- If Stack is not full, (i.e) there is some space available in stack, then, top pointer is incremented by one and then the element will be pushed at the top of the stack.

Algorithm:

```
Push(element)
if (top = = size – 1)
    Write “Stack is full. Can‟t perform push”
else
    top = top + 1
    stack [ top ] = element
end if
```

- Consider the size of stack is 3. Initially **top = -1**.
- Consider, We need to push the element 10 into the stack
  - When we check the stack, we find that stack is not full. So we increment top by 1. Hence **top = 0**. We store the element 10 in stack[0]

Stack[2]	
Stack[1]	
top→ Stack[0]	10

- We need to push the element 15 into the stack
  - When we check the stack, we find that stack is not full. So we increment top by 1. Hence **top = 1**. We store the element 15 in stack[1]

Stack[2]	
top→ Stack[1]	15
Stack[0]	10

- We need to push the element 20 into the stack
  - When we check the stack, we find that stack is not full. So we increment top by 1. Hence **top = 2**. We store the element 20 in stack[2]

top→ Stack[2]	20
Stack[1]	15
Stack[0]	10

- We need to push the element 25 into the stack. When we check the stack, we find that stack is full.(i.e) top = = size – 1 becomes true. So we can‟t push the element 25 unless some of the elements from the stack is popped.

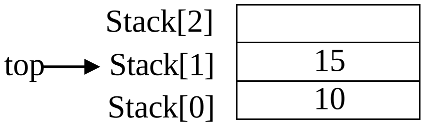
POP operation:

- Pop operation deletes the element at top of the stack.
- Before performing pop operation, it is necessary to check the stack empty condition, because if a stack is empty, pop operation cannot be performed.
- If Stack is not empty, (i.e) there is some values are available in stack, then, element at top of stack is popped out and top pointer is decremented by 1.

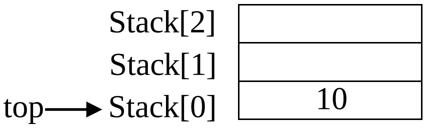
Algorithm:

```
Pop( )
if (top = = – 1)
    Write "Stack is empty. Can‟t perform pop"
else
    element = stack[top]
    top = top -1
    return(element)
 end if
```

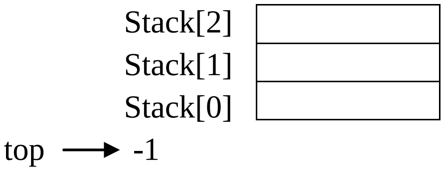
- If pop operation is to be done, element at top of stack 20 is popped and top is decremented by 1



- If pop operation is to be done, element at top of stack 15 is popped and top is decremented by 1



- If pop operation is to be done, element at top of stack 10 is popped and top is decremented by 1



- Since top is -1, stack is empty. Hence further pop operation cannot be done.

Display:

- Display operation displays the element in the stack
- If there is no elements in stack, element can‟t be displayed
- If there are elements in stacks, it displays the elements from the top position of the stack till the zeroth position

```
display( )
if top = = -1
   Write " stack is empty. No elements to display"
else
    for i = top to 0
         Write stack [ i ]
    end for
end if
```

<u>APPLICATIONS OF STACK:</u>

- Book in a table
- Shipment in cargo
- Plate on a tray
- Towers of hanoi
- Conversion of infix expression to postfix expression
- Evaluation of arithmetic expression
    - An arithmetic expression consists of operands and operators
    - **Notations:** Infix => operand *operator* operand
    - prefix => *operator* operand operand
    - Postfix => operand operand *operator*

CONVERSION OF INFIX EXPRESSION TO POSTFIX EXPRESSION:

Algorithm for converting infix to postfix:

```
While(tokens are available)   // token may be a alphabet or number or operator or „(„ or „)"
        ch = read(token)
        if(ch = = „ ( „ )
                 push(ch)
                 break
        end if
      if(ch = = „ ) „ )
                 pop all operators until „(„ is encountered
                 break
     end if
     if ( ch = = operand)
            output(ch) (or) Write the operand in postfix expression
     else
       if ( priority(ch) <= priority( stack[top])
             element = pop( )
             output(element)
             push(ch)
       else
             push(ch)
end while
write postfix expression
```

<u>Example 1</u>

Convert the infix expression (a * b) to postfix

Operator stack	**Input token**	**Postfix expression**
(	**(**	**Empty**
(	**a**	**a**
* (	*****	**a**
* (	**b**	**ab**
* (	**)** Here close bracket encountered, so all symbols upto open bracket is popped and operators alone are stored in postfix expression	**ab***

Example 2:

((a + b) ^ c – (d*e) / f)

Operator stack	Input token	Postfix expression
		Empty
(	**(**	**Empty**
((	**(**	**Empty**
((	**a**	**a**
+ ((	+	**a**
+ ((	**b**	**Ab**
+ **(** (	**)**	**ab+**
^ (	^	**ab+**
^ (	**c**	**ab+c**
^ (	**-** //Here priority of – is less than ^ , so ^ is popped from stack and stored in postfix expression and – is pushed to stack in next step	**ab+c**

- (		**ab+c^**
(- (	**(**	**ab+c^**
(- (	**d**	**ab+c^d**
* (- (	*****	**ab+c^d**
* (- (	**e**	**ab+c^de**
* (- (	**)** // Here close bracket encountered, so all symbols upto open bracket is popped and operators alone are stored in postfix expression	**ab+c^de**
- (		**ab+c^de***
/ - (	**/** // Here priority of operator / is greater than - ,so / is pushed to a stack	**ab+c^de***
/ - (	**f**	**ab+c^de*f**

/ - (	**)** // Here close bracket encountered, so all symbols until open bracket are popped out and operators are stored in postfix expression	**ab+c^de*f/-**
Stack is empty	**Tokens are not available**	**ab+c^de*f/-**

EVALUATION OF POSTFIX EXPRESSION USING STACK:

Algorithm for evaluating postfix expression:

```
While(tokens are available)
        ch = read(token)
        if(ch = = operand)
        read  ch
        push(ch)
        else if (ch = = operator)
         y = pop( )
         x = pop( )
         result = x operator  y
         push(result)
       else
         write "invalid postfix expression"
      end if
end while
value = pop()
write  value
```

Example 1:

Postfix expression is ab*

Stack	Input token
5	**a** // a is a operand. So, read the value for a. For (eg) a = 5. Then push value of a to stack
7 5	**b** // b is a operand. So, read the value for b. For (eg) b = 7. Then push value of b to stack

7 → y =7 5 → x = 5	***** // ***** is a operator. So, pop two values from stack. (i.e) pop 1st value and store it in y and pop 2nd value and store it in x.
35	Find the result = x * y => result = 35 Push the result to the stack

<u>Example 2:</u>

Postfix expression is ab+c^de*f/-

Stack	Input token
5	**a** // a is a operand. So, read the value for a. For (eg) a = 5. Then push value of a to stack
7 5	**b** // b is a operand. So, read the value for b. For (eg) b = 7. Then push value of b to stack
7 → y =7 5 → x = 5	**+** // ***** is a operator. So, pop two values from stack. (i.e) pop 1st value and store it in y and pop 2nd value and store it in x.
12	Find the result = x + y => result = 12 Push the result to the stack
2 12	**c** // c is a operand. So, read the value for a. For (eg) c = 2. Then push value of a to stack
2 → y =2 12 → x = 12	^ // ^ is a operator. So, pop two values from stack. (i.e) pop 1st value and store it in y and pop 2nd value and store it in x.

14	Find the result = x ^ y => 12 ^ 2 => 1100 ^ 0010 => result = 1110 result = 14. Push the result to the stack
3 14	**d** // d is a operand. So, read the value for d. For (eg) d = 3. Then push value of a to stack
2 3 14	**e** // e is a operand. So, read the value for e. For (eg) e = 2. Then push value of a to stack
2 → y =2 3 → x = 3 14	* // * is a operator. So, pop two values from stack. (i.e) pop 1st value and store it in y and pop 2nd value and store it in x.
6 14	Find the result = x * y => result = 3 * 2 => result = 6 Push the result to the stack
3 6 14	**f** // f is a operand. So, read the value for f. For (eg) f = 3. Then push value of f to stack
3 → y =3 6 → x = 6 14	**/** // **/** is a operator. So, pop two values from stack. (i.e) pop 1st value and store it in y and pop 2nd value and store it in x.
2 14	Find the result = x / y => result = 6/3 => result = 2 Push the result to the stack
2 → y =2 14 → x = 14	**-** // **-** is a operator. So, pop two values from stack. (i.e) pop 1st value and store it in y and pop 2nd value and store it in x.
12	Find the result = x - y => result = 14 – 2 => result= 12 Push the result to the stack
Pop the value from stack and display it. Hence 12 is popped and displayed. Value of the **ab+c^de*f/-** is **12**	**No token present**

QUEUES:

- Queue is an ordered collection of homogeneous elements where insertion(enqueue) and deletion(dequeue) takes place at two ends(front and rear).
- Enqueue operation is performed at rear
- Dequeue operation is performed at front
- Queue follows a strategy FIFO(First In First Out). (i.e) The element which is enqueued first is dequeued from the queue first.

OPERATIONS OF QUEUE:

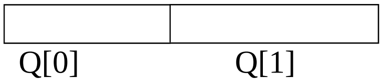
i) ENQUEUE:

- Enqueue of an element can be performed in the queue only if the space is available in queue.
- Hence, before inserting an element, it is necessary to check the status of queue.
- If space is available in queue, the element will be inserted at the rear position in queue

Algorithm:

```
enqueue(int element)
if (front = = -1)
      front = 0
end if
if (rear = = SIZE – 1)
     write " Queue is full. Can‟t perform enqueue"
else
     rear = rear +1
     Q[ rear ] = element
end if
```

Consider the **SIZE of queue is 2.** Initially, **front = -1, rear = -1**



enqueue(10)

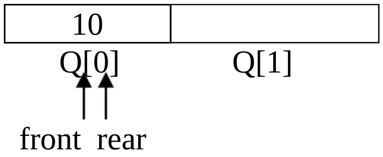
front = = -1 => True

    front = 0

rear = = 1 => -1 = = 1 => false

rear = 0

Q[0] = 10



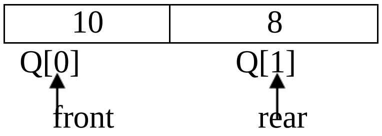
enqueue(8)

front = = -1 => False

rear = = 1 => 0 = = 1 => false

rear = 1

Q[1] = 8



enqueue(15)

front = = -1 => False

rear = = 1 => 1 = = 1 => True => **Queue is full**

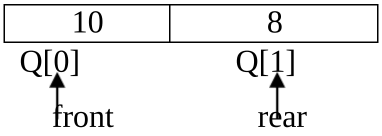
ii) DEQUEUE:

- Dequeue of an element can be performed in the queue only if element is present.
- Hence, before performing dequeue, it is necessary to check the status of queue.
- If element is present in queue, the element is dequeued from the front.

Algorithm:

```
dequeue( )
if (front = = -1)
      Write " Queue is empty. Can"t dequeue elements"
else
     Write " The element dequeued is Q [ front ]
      if ( front = = rear )
             front = rear = -1
     else
             front = front + 1
     end if
end if
```

At present, Queue has elements

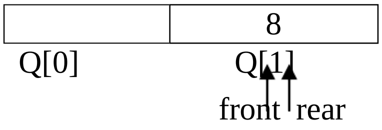


dequeue( )

front = = -1 => false

The element dequeued is 10

front = 1



dequeue( )

Front = = -1 => false

The element dequeued is 8

if 1 = = 1 => true

  front = rear = -1

Q[0]	Q[1]

Dequeue( )

front = = -1 => true => Queue is empty. Can‟t dequeue elements

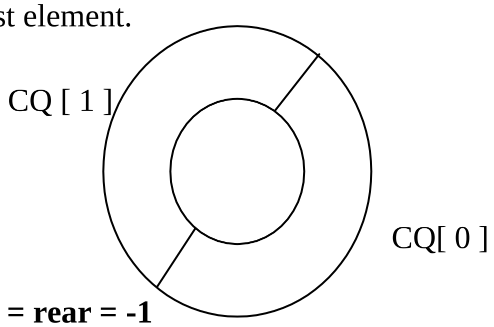
<u>Display elements in a queue:</u>

- Display operation displays the elements from the front of queue to the rear.

```
display( )
if( front = = -1)
  write " Queue is empty"
else
   for i = front to rear
       write Q[ i ]
   end for
end  if
```

<u>CIRCULAR QUEUE:</u>

- Circular queue is similar to queue, except that the elements are arranged in circular manner.
- Since the elements are in circular manner, the next element of the last position in queue is the first element.



Initially **front = rear = -1**

i) ENQUEUE:

- Enqueue of an element can be performed in the queue only if the space is available in queue.
- Hence, before inserting an element, it is necessary to check the status of queue.
- If space is available in queue, the element will be inserted at the rear position in queue

Algorithm:

```
enqueue(int element)
if ( front = = (rear + 1) % MAX )
    write “ Queue is full. Can‟t enqueue” // (i.e) front and rear are equal
    break
end if
if (front = = -1)
     front = rear = 0
else
    rear = ( rear + 1 ) % MAX
end if
CQ[ rear ] = element
```

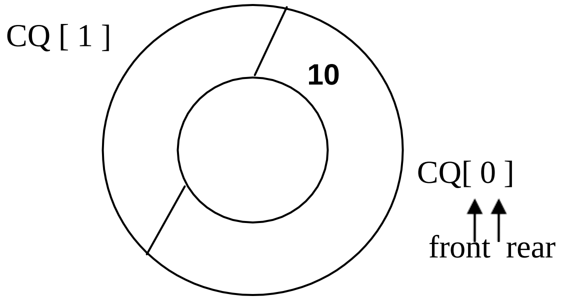
Consider the **SIZE of circular queue is 2.** Initially, **front = -1, rear = -1**

enqueue(10)

if -1 = = 0 => false

if front = = -1 => true => **front = rear = 0**

CQ[0] = 10

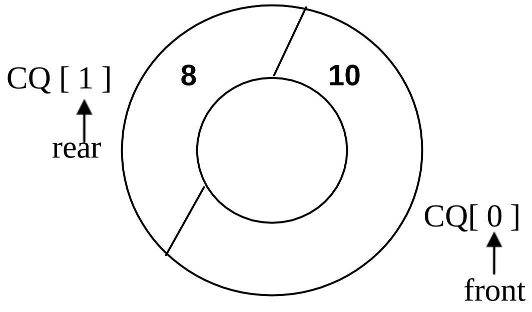


enqueue(8)

if front = = -1 => false

rear = (rear+1) % 2 => rear = 1 %2 => **rear = 1**

CQ[1] = 8



enqueue(15)

if front = (1 + 1) % 2 => 0 = = 0 => Circular queue is full

ii) DEQUEUE:

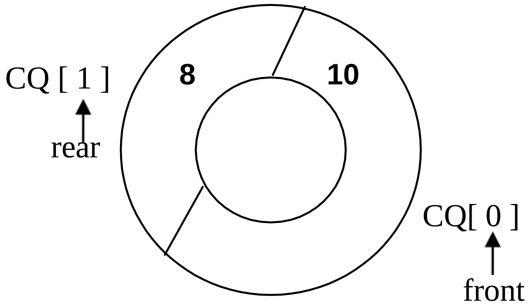
- Dequeue of an element can be performed in the circular queue only if element is present.
- Hence, before performing dequeue, it is necessary to check the status of circular queue.
- If element is present in circular queue, the element is dequeued from the front.

Algorithm:

```
dequeue( )
if (front = = -1)
      Write “ Queue is empty. Can‟t dequeue elements”
else
     Write “ The element dequeued is CQ [ front ]
      if ( front = = rear )
             front = rear = -1
     else
             front = (front + 1) % MAX
     end if
end if
```

Trace out:

At present, circular queue has following elements



dequeue( )

The element dequeued is 10

front = 1

dequeue( )

The element dequeued is 8

front = rear = -1 => Circular queue is empty

Display:

- Display the elements in circular queue

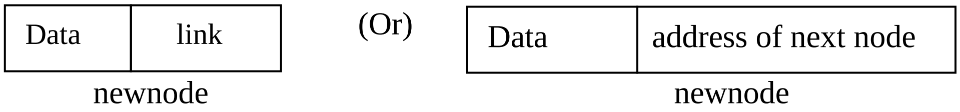
```
display( )
if ( front = = -1)
       write “ Queue is empty
while ( front ! = rear)
       write CQ[ front ]
     front = (front + 1 ) % MAX
end while
write CQ [ rear ]
```

LINKED LIST:

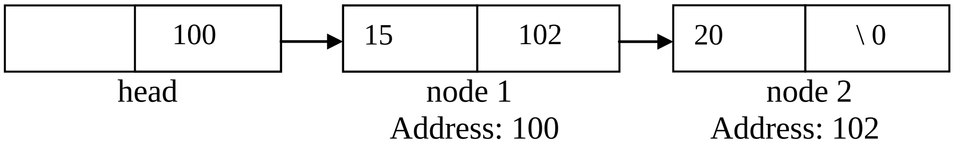
- Linked list is a dynamic data structure where its size can be varied during its use
- Adjacency between the elements of linked list are maintained by using links/pointers. Link/ Pointer stores the address of next node
- Each element in a linked list is called as node

SINGLE LINKED LIST

Representation of node in a Single linked list



- Each node has two parts. Data part where the actual data is stored and the link part where the address of next node is stored.
- (i.e) Each node contains pointer to the next node in list
- Each node has a name and a specific address, using that the node can be accessed
- Head is the empty node which points to the first node in the list. To make the head to point to the first node in a list, address of the first node is stored in a head node.
- Head node doesn‟t store any data.



Append operation in linked list:

- Append is used to append the node at end of the linked list.
- When there is no node in a linked list, head points to NULL
- When the first node is created, head pointer is made to point to the first node by storing the address of that node in head.
- head is a pointer which should always points to 1st node in the list.
- tail always points to last node. When there is only one node in list, both head and tail will be same

Append( )

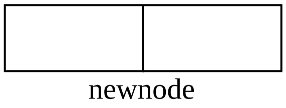
```
Create a node named as "newnode"
newnode → data = value
newnode → next = NULL
if head = = NULL
    head = newnode
else
    tail → next = newnode
end if
tail = newnode
```

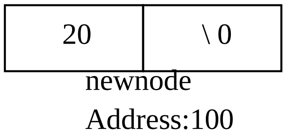
EXAMPLE:

Since there is no node initially *head=NULL

Append 20

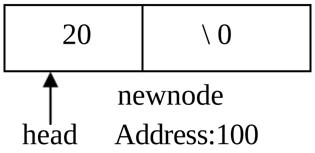
Initially we need to create a empty node named as newnode

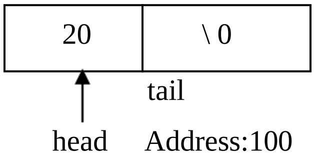




If head = = NULL => True. Hence Address of newnode is stored in head

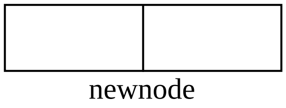
head = 100

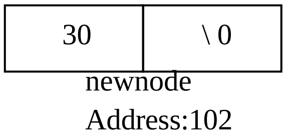




Append 30

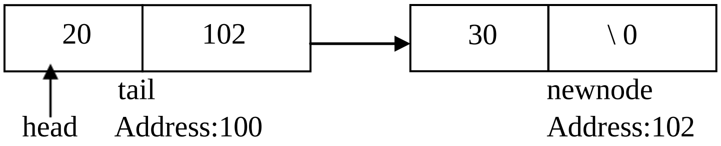
Initially we need to create a empty node named as newnode

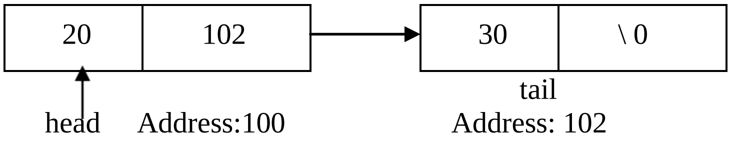




If head = = NULL => False.

tail → next = 102





INSERTION IN A SINGLE LINKED LIST:

- A node can be inserted in a single linked list at any position
- If the newnode is to be inserted at 1st position in the linked list, then the address of the present head should be stored in next part of newnode. Then head is made to point to the newnode by storing its address in head.
- P is a pointer which is used to point to the position where insertion need to be done.

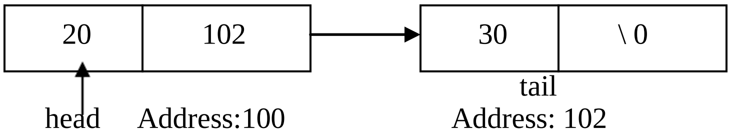
Algorithm:

insert( )

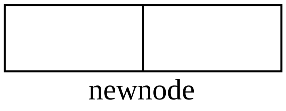
```
Create a node named as "newnode"
newnode → data = value
newnode → next = NULL
p= head
if pos = = 1
    newnode → next = head
    head = newnode
else
    for i = 1 to pos
        prevnode = p
        p = p → next
    end for
prevnode → next = newnode
newnode → next = p
end if
```

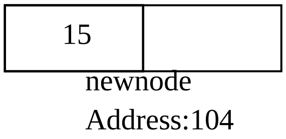
Example:

Insertion of element 15 at position 1



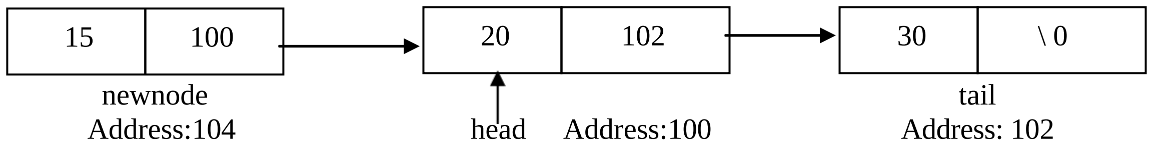
we need to create a empty node named as newnode and store value in it



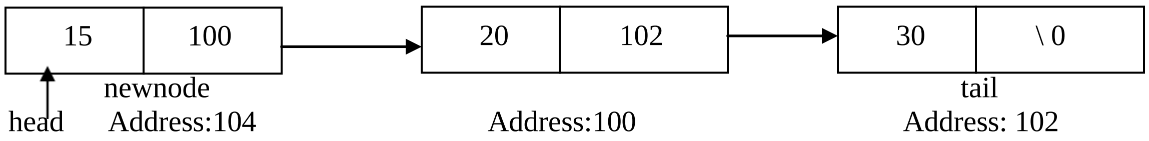


if pos = = 1 => True

newnode →next = 100

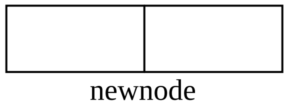


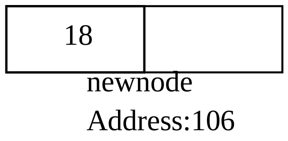
head = 104



Insertion of element 18 at position 2:

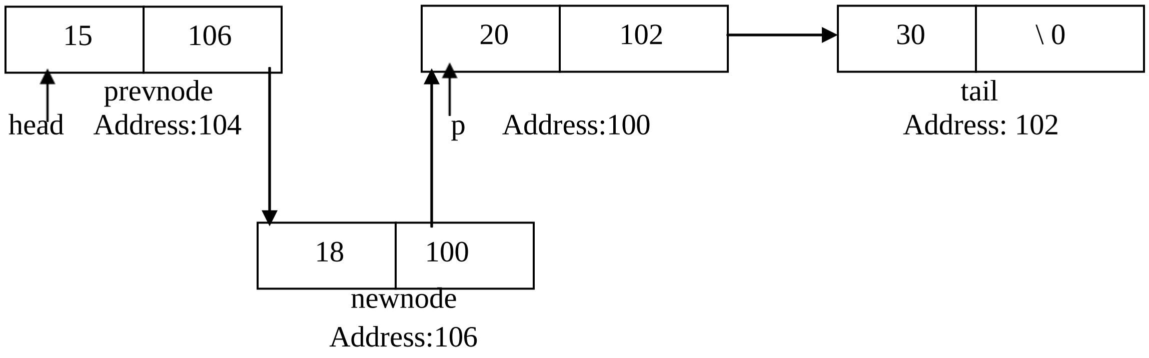
we need to create a empty node named as newnode and store value in it





If pos = = 1 => False

P points to head node. Make the p as prevnode and Move the pointer p to position 2



DELETION OF ELEMENT IN A LINKED LIST:

- A node can be deleted in a single linked list at any position
- If the node is to be deleted at 1st position in the linked list, then the head is made to point to second node and 1st node should be deleted
- P is a pointer which is used to point to the position where deletion need to be done.

Algorithm:

delete( )

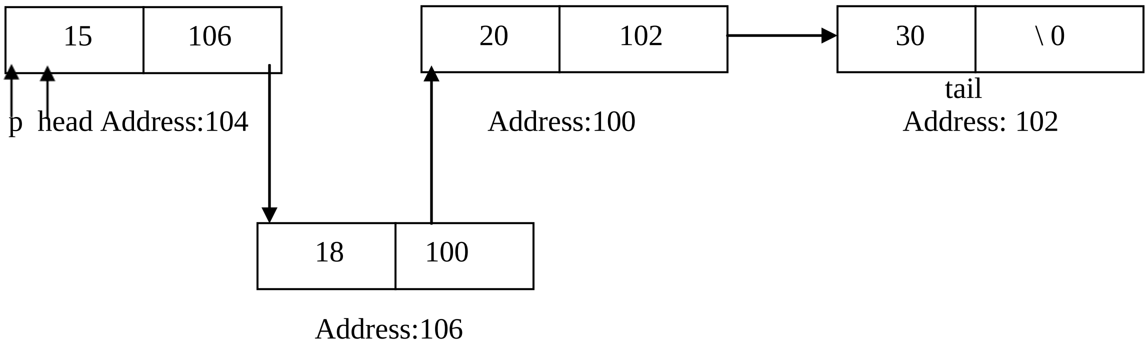
```
p= head
if pos = = 1
      head = p →next
      free( p )
else
      for i = 1 to pos
                prevnode = p
                p = p → next
     end for
prevnode → next = p → next
if( p →next = = NULL)
       tail = prevnode
end if
free(p)
end if
```

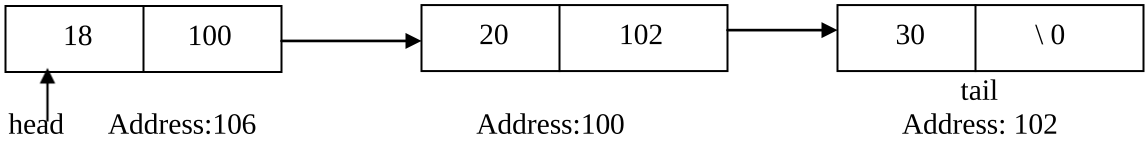
Example:

Deletion of element at position 1

If pos = = 1 => True

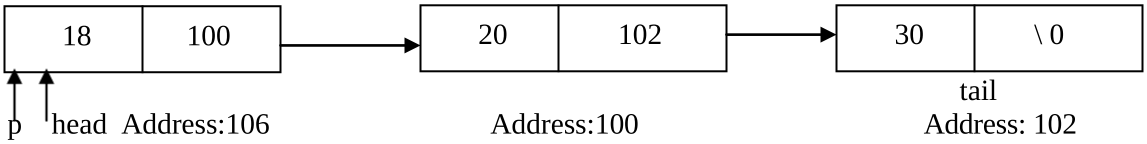
head = 106





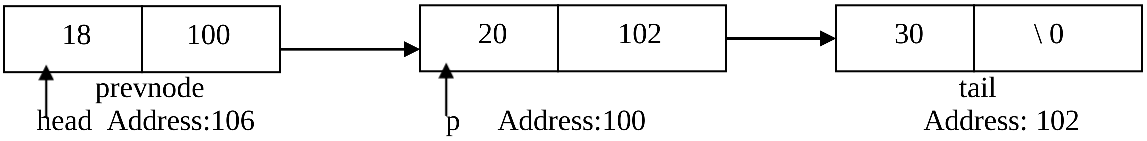
Deletion of element at position 2

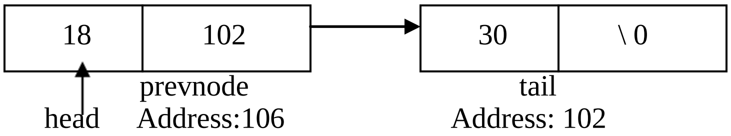
p = head



if pos = = 1 => False

Make p as prevnode and Move the pointer p to position 2

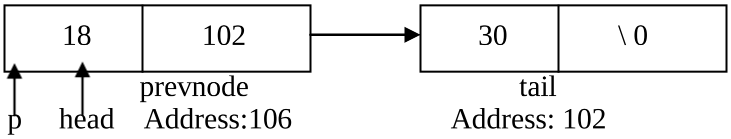




<u>Displaying elements in a linked list:</u>

Display( )

```
p = head
while( p ! = NULL)
    write  p  → data
    p = p→next
end while
```



while p ! = NULL => True

18 is displayed

p points to next node 30

while p ! = NULL => True

30 is displayed

p points to next location. (i.e) There is no node . Hence p points to NULL

while p! = NULL => False

CIRCULAR LINKED LIST:

- Circular linked list is similar to the single linked list. In circular linked list, the last node in the list is connected to the first node by storing the address of the first node in the last node.

APPEND OPERATION IN CIRCULAR LINKED LIST:

- Append is used to append the node at end of the linked list.
- *When there is no node in a linked list, head points to NULL*
- When the first node is created, head pointer is made to point to the first node. The first node is made to point to itself forming a circular list. Whenever a node is appended, the last node is made to point to first node.
- head is a pointer which should always points to 1st node in the list.
- tail always points to last node. When there is only one node in list, both head and tail will be same

Append( )

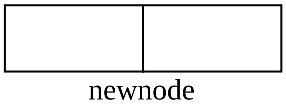
```
Create a node named as "newnode"
newnode → data = value
if head = = NULL
    head = newnode
else
    tail → next = newnode
end if
newnode →next =head
tail = newnode
```

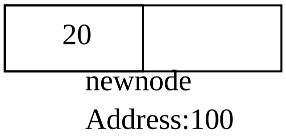
EXAMPLE:

Since there is no node initially *head=NULL

Append 20

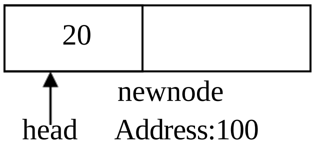
Initially we need to create a empty node named as newnode

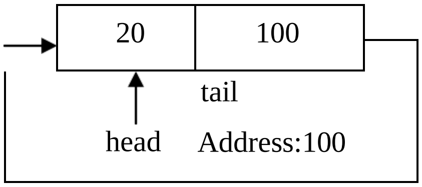




If head = = NULL => True. Hence Address of newnode is stored in head

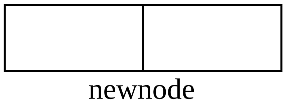
head = 100

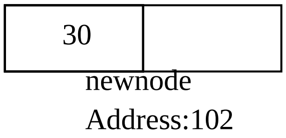




Append 30

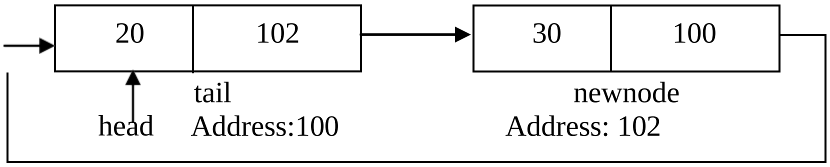
Initially we need to create a empty node named as newnode

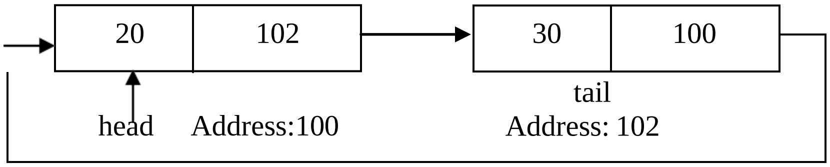




If head = = NULL => False.

tail → next = 102 , newnode →next =100





INSERTION IN A CIRCULAR LINKED LIST:

- A node can be inserted in a circular linked list at any position
- If the newnode is to be inserted at 1st position in the linked list,then head is made to point to the newnode by storing its address in head. Then the last node is made to point to the first node.
- p is a pointer which is used to point to the position where insertion need to be done.

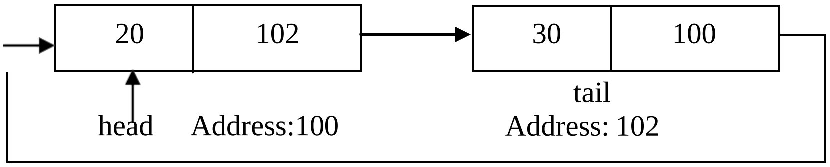
ALGORITHM:

insert( )

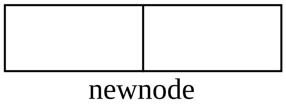
```
Create a node named as "newnode"
newnode → data = value
p= head
if pos = = 1
    newnode → next = head
    head = newnode
    tail → next = head
else
    for i = 1 to pos
        prevnode = p
        p = p → next
    end for
prevnode → next = newnode
newnode → next = p
end if
```

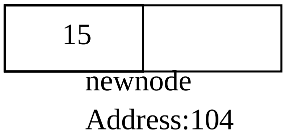
EXAMPLE:

INSERTION OF ELEMENT 15 AT POSITION 1



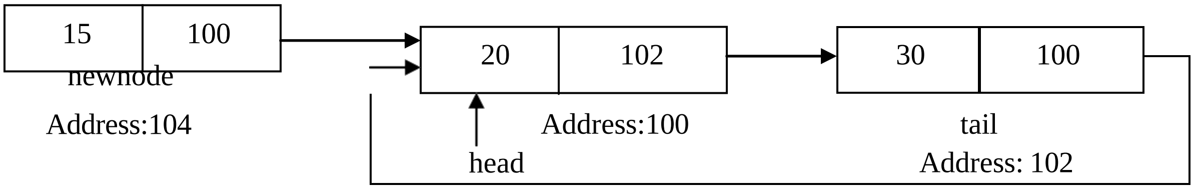
we need to create a empty node named as newnode and store value in it



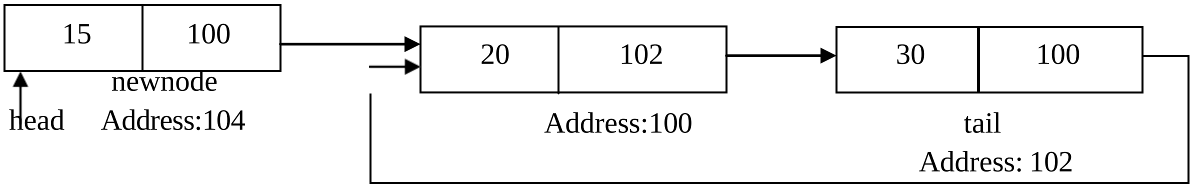


if pos = = 1 => True

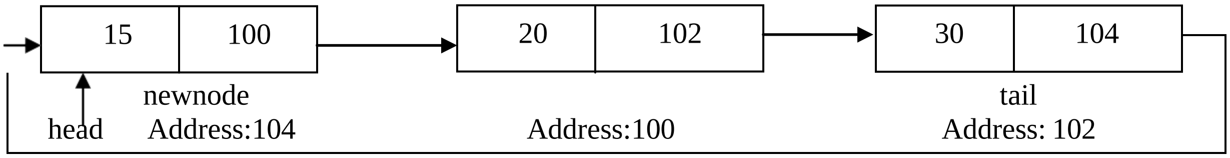
newnode →next = 100



head = 104

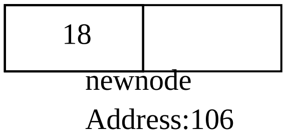


tail → next = 104



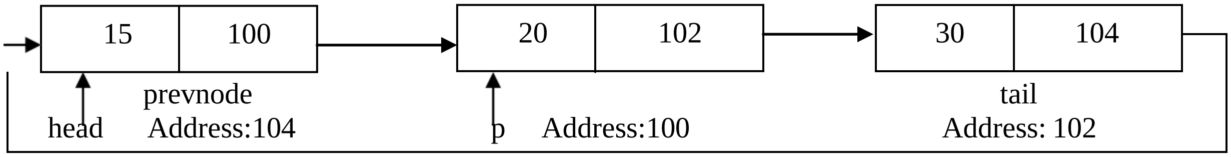
INSERTION OF ELEMENT 18 AT POSITION 2:

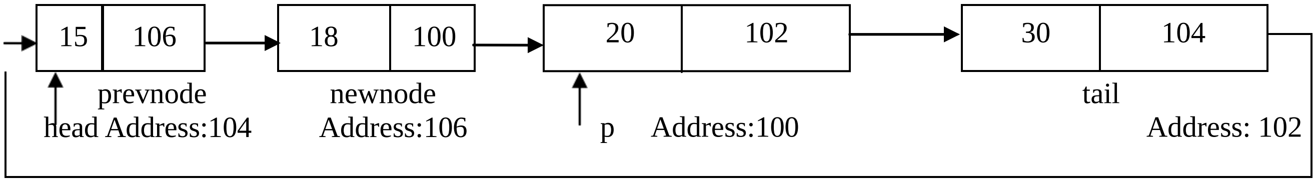
we need to create a empty node named as newnode and store value in it



If pos = = 1 => False

p points to head node. Make the p as prevnode and Move the pointer p to position 2





DELETION OF ELEMENT IN A CIRCULAR LINKED LIST:

- A node can be deleted in a circular linked list at any position
- If the node is to be deleted at 1st position in the linked list, then the head is made to point to second node and last node is linked to the head node. Then the 1st node will be deleted.
- p is a pointer which is used to point to the position where deletion need to be done.

ALGORITHM:

delete( )

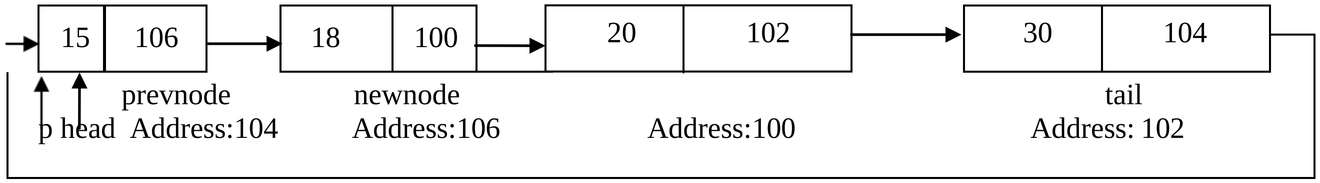
```
p= head
if head = = tail
      free(p)
      head = tail = NULL
else if pos = = 1
      head = p →next
      tail →next = head
      free( p )
else
      for i = 1 to pos
                prevnode = p
                p = p → next
     end for
 prevnode → next = p → next
 if( p →next = = NULL)
              tail = prevnode
end if
free(p)
end if
```

Example:

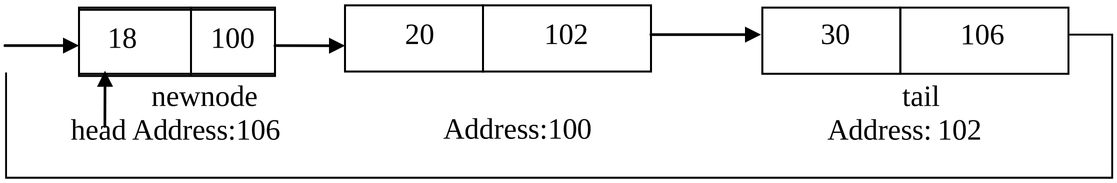
DELETION OF ELEMENT AT POSITION 1

If pos = = 1 => True

head = 106



tail → next = 106 , free(p)

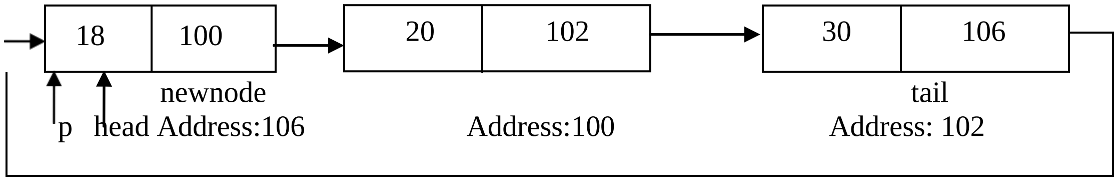


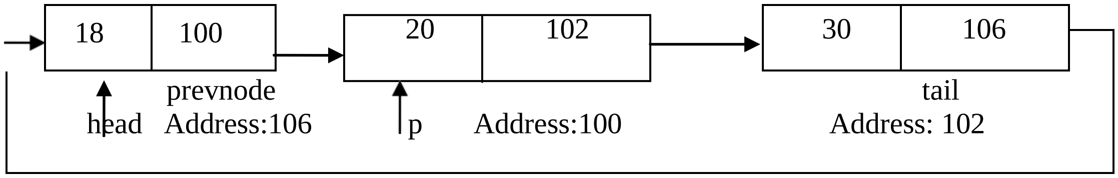
DELETION OF ELEMENT AT POSITION 2

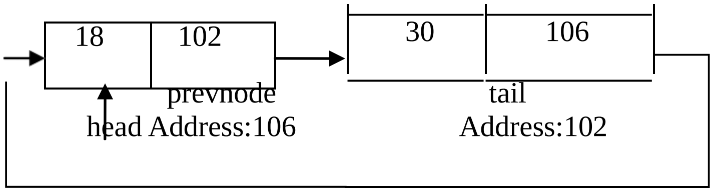
p = head

If pos = = 1 => False

p points to head node. Then, Make the p as prevnode and Move the pointer p to position 2



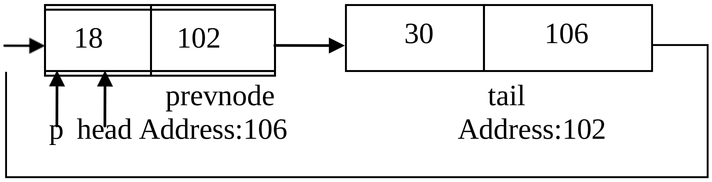




DISPLAYING ELEMENTS IN A LINKED LIST:

Display( )

```
p = head
while( p ! = NULL)
    write  p  → data
    p = p→next
    if p = = head
        break
    end  if
end while
```



while p ! = NULL => True

18 is displayed

p points to next node 30

while p ! = NULL => True

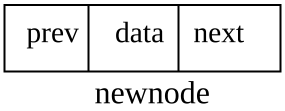
30 is displayed

p points to next location. (i.e) head node . since p = = head, break executed.

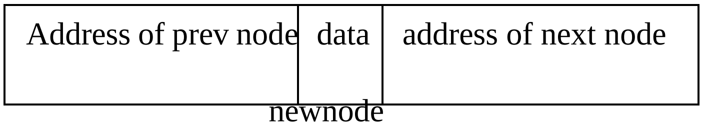
DOUBLE LINKED LIST:

- Double linked list is a linked list where the elements can be traversed both in forward direction and also in backward direction

Representation of node in a Double linked list



(Or)



- Each node has three parts. Address of the previous node, data and address of next node
- By using the address of previous node, the previous node in the list can be accessed and by using the address of next node, the next node in the list can be accessed
- (i.e) Each node contains pointer to the previous node and also next node in list
- Each node has a name and a specific address, using that the node can be accessed

Append operation in linked list:

- Append is used to append the node at end of the linked list.
- When there is no node in a linked list, head points to NULL
- When the first node is created, head pointer is made to point to the first node by storing the address of that node in head.
- Previous node address of the first node is always NULL
- tail always points to last node. When there is only one node in list, both head and tail will be same

Append( )

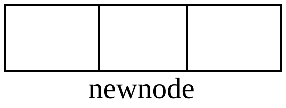
```
Create a node named as "newnode"
newnode → data = value
newnode → next = NULL
if head = = NULL
    newnode → prev =NULL
    head = newnode
else
    tail → next = newnode
    newnode → prev = tail
end if
tail = newnode
```

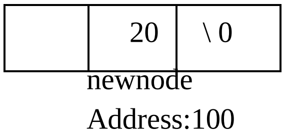
EXAMPLE:

Since there is no node initially *head=NULL

Append 20

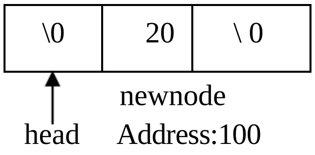
Initially we need to create a empty node named as newnode

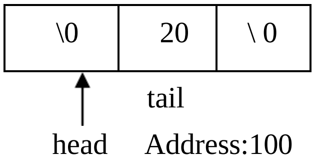




If head = = NULL => True. Hence Address of newnode is stored in head

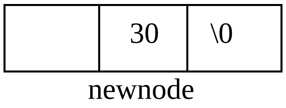
head = 100





Append 30

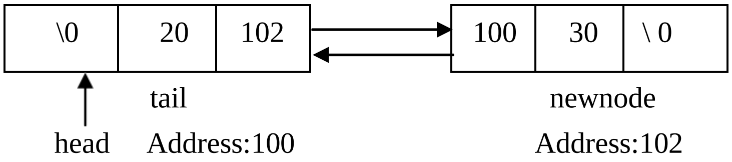
Initially we need to create a empty node named as newnode and store the value



If head = = NULL => False.

tail → next = 102

newnode → prev = 100



INSERTION IN A DOUBLE LINKED LIST:

- A node can be inserted in a double linked list at any position

Insertion at 1st position:

- If the newnode is to be inserted at 1st position in the linked list, head is made to point to the newnode.
- Previous node address of newnode is made as NULL.
- Address of the present head (i.e) p, should be stored in next part of newnode. Address of newnode is stored in Previous part of p.

Insertion at any position:

- P is a pointer which is used to point to the position where insertion need to be done.
- Both the forward and backward links are created to the new node which is inserted at certain position.

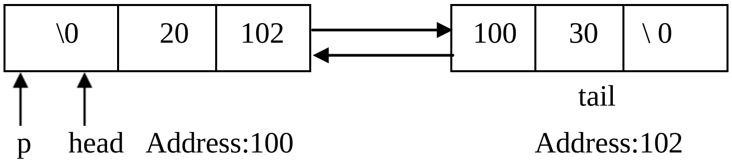
Algorithm:

insert( )

```
Create a node named as “newnode”
newnode → data = value
newnode → next = NULL
p= head
if pos = = 1
    head = newnode
    newnode → prev = NULL
    newnode → next = p
    p→ prev = newnode
else
    for i = 1 to pos
          prevnode = p
          p = p → next
    end for
newnode→ prev = p→ prev
newnode → next = p
p→prev→next = newnode
p→prev = newnode
end if
```

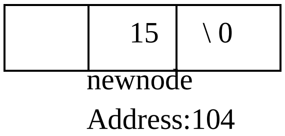
Example:

At present, link list has elements



Insertion of element 15 at position 1

we need to create a empty node named as newnode and store value in it



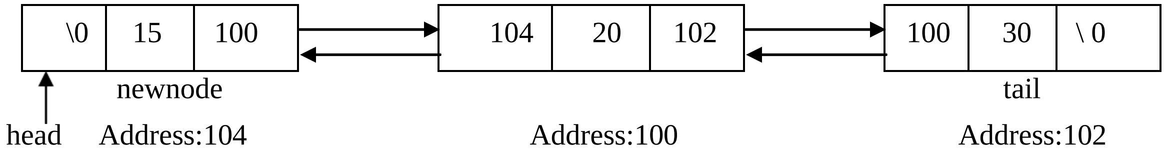
if pos = = 1 => True

head = 104

newnode →prev = NULL

newnode → next = 100

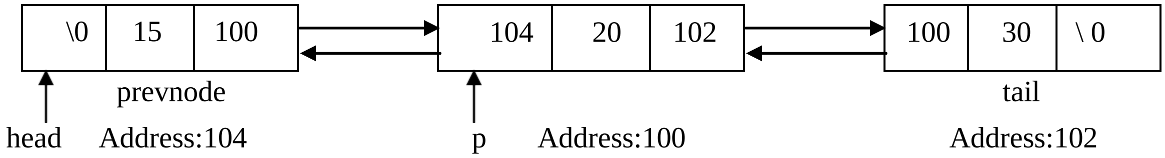
p → prev = 104



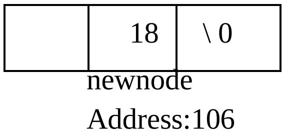
<u>Insertion of element 18 at position 2:</u>

At present, link list has elements

Initially pointer p points to head node. Make the p as prevnode whenever p moves to next position. Move p until the position where insertion is to be done.(i.e) p is moved to position 2.



- we need to create a empty node named as newnode and store value in it



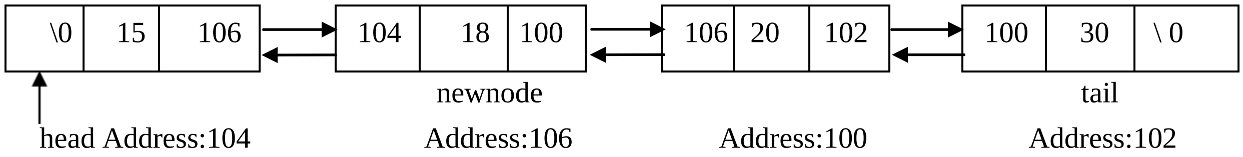
if pos = = 1 => False

newnode→ prev = 104

newnode → next = 100

p→prev→next = 106

p→prev = 106



DELETION OF ELEMENT IN A DOUBLE LINKED LIST:

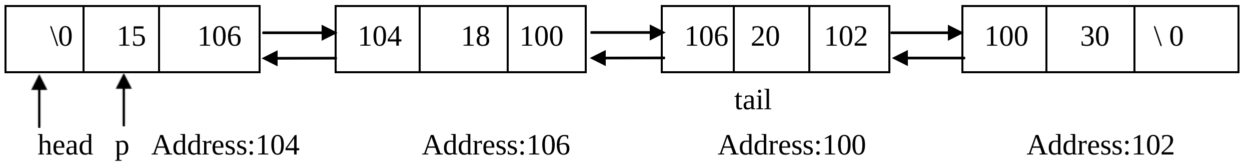
- A node can be deleted in a single linked list at any position
- If the node is to be deleted at 1st position in the linked list, then the head is made to point to second node and 1st node should be deleted
- P is a pointer which is used to point to the position where deletion need to be done.

Algorithm:

```
delete( )
p= head
if pos = = 1
      head = p →next
      head → prev =NULL
      free( p )
else
      for i = 1 to pos
                p = p → next
     end for
     if   p→ next = = NULL                    // Deletion of last node
              p → prev → next = NULL
              tail = p → prev
     else
              p → prev → next = p →next
              p → next → prev = p → prev
     end if
free(p)
end if
```

Example:

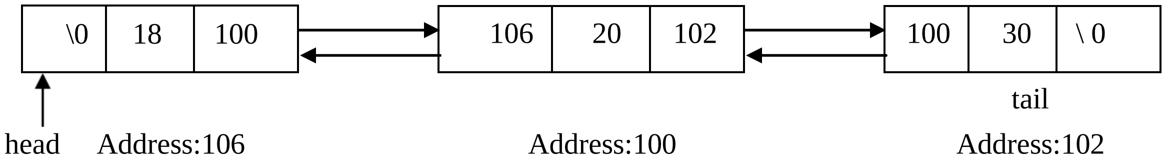
At present, the linked list has elements,



Deletion of element at position 1

if pos = = 1 => True

head = 106



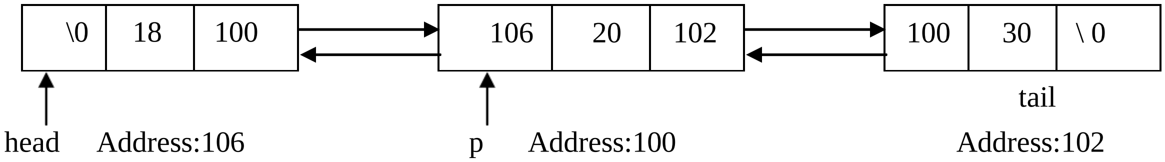
head → prev = NULL

Deletion of element at position 2

if pos = = 1 => False

P points to head node. Move the pointer p to position where the deletion to be done. (i.e) pos 2

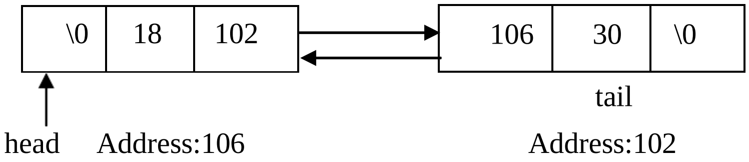
At present, elements in linked list are,



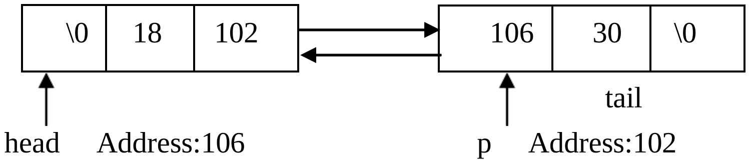
if p → next = =NULL => False

p → prev→ next = 102

p → next → prev = 106



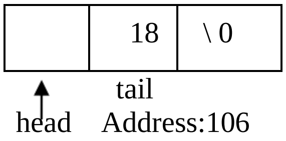
Deletion at position 2



if p → next = = NULL => True

p → prev → next = NULL

tail = 106



Displaying elements in a double linked list:

displayforward( )

```
p = head
while( p ! = NULL)
    write p  → data
    p = p→next
end while
```

displaybackward( )

```
p = tail
while( p ! = NULL)
    write p  → data
    p = p→prev
end while
```

LINKED STACK

- It is also called as implementation of stack using linked list.
- head is a pointer which initially points to NULL
- The head pointer specifies the top of the stack.
- Push operation is similar to insertion of element in linked list at position 1

Operations of stack:

i) PUSH OPERATION:

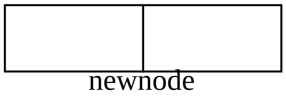
- push operation push the element to the top of the stack(head)
- Initially when there is no elements, head = NULL.
- If new element is to be pushed, new node is created and the element is stored.
- Address of the present head node is stored in next link of the newnode, thereby the new element is inserted at 1st position
- head is made to point to newnode.(i.e) 1st node, which is a top of stack

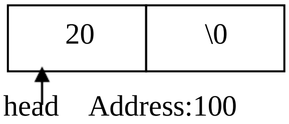
Algorithm:

```
push ( int element)
create a newnode
newnode→data = element
newnode → next = head
head = newnode
```

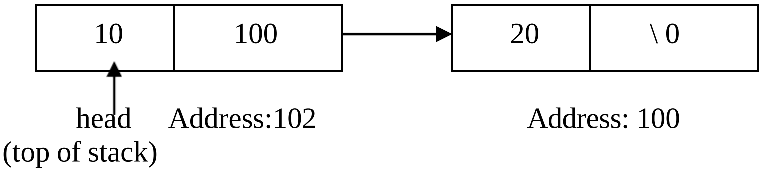
push(20)

Initially we need to create a empty node named as newnode





push (10)



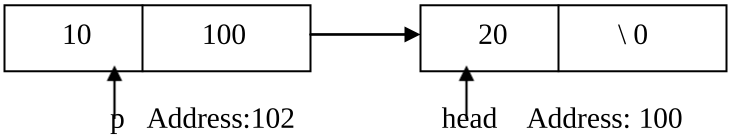
POP OPERATION:

- pop operation removes the element at top of the stack
- If stack is empty, pop operation cannot be performed. (i.e) when head = NULL, pop can"t be performed
- Make the pointer p to point to 1st node. Move the head to point to next location. Free the memory pointed by p

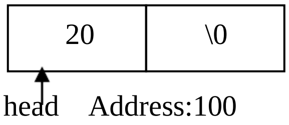
Algorithm:

pop( )

```
p = head
print " the element popped is head → data"
head = head → next
free ( p )
```



free(p)



Display:

Display operation displays the elements in stack from the top of stack. (i.e) head

```
display( )
p = head
while p ! = NULL
    print " p → data"
    p = p → next
end while
```

LINKED QUEUE (or) Implementation of Queue using linked list.

- Initially *front =* rear = NULL.
- Enqueue is similar to append operation in linked list.
- Dequeue is similar to deleting the 1st node in linked list.

i) ENQUEUE OPERATION:

- In enqueue operation, element is always inserted at the rear position.

Algorithm:

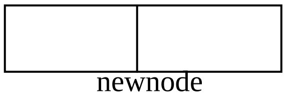
enqueue(int element)

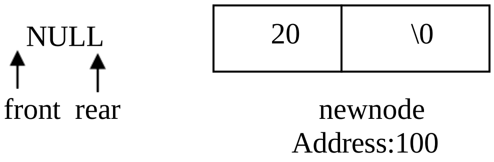
```
create newnode
newnode → data = element
newnode → next = NULL
if ( front = = NULL)
      front = rear = newnode
else
      rear → next = newnode
      rear = newnode
end  if
```

Trace out:

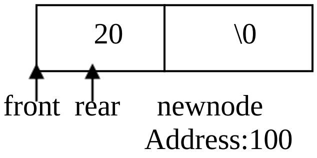
enqueue(25)

Initially we need to create a empty node named as newnode



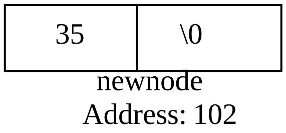


Front = = NULL => true

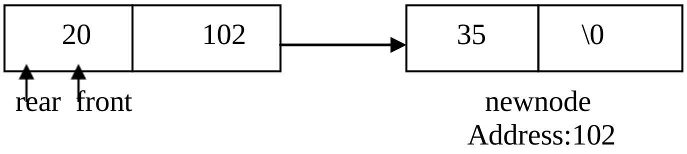


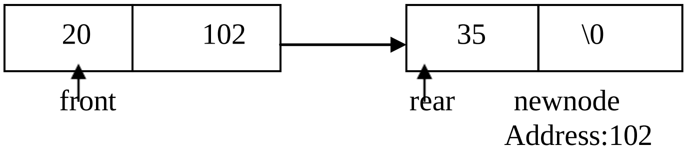
enqueue(35)

Initially we need to create a empty node named as newnode



front = = NULL => false





ii) DEQUEUE OPERATION:

- Dequeue of an element can be performed in the queue only if element is present.
- Hence, before performing dequeue, it is necessary to check the status of queue.
- If element is present in queue, the element is dequeued from the front.

Algorithm:

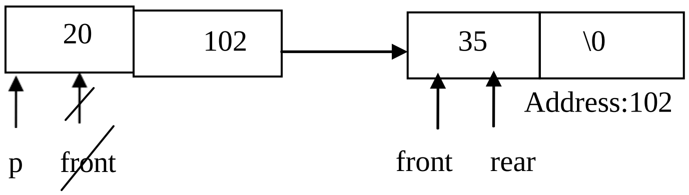
dequeue( )

```
if (front = = NULL)
    Write " Queue is empty. Can‟t dequeue elements"
else
    p = front
    front = front →next
    Write " deleted element is p → data"
    free( p )
end if
```

TRACE OUT:

dequeue( )

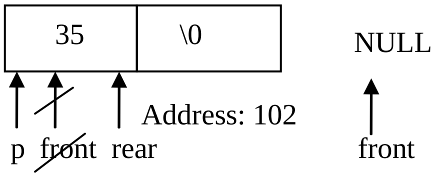
if front = = NULL => false



deleted element is 20

dequeue( )

if front = = NULL => false



deleted element is 35

dequeue( ) => front = = NULL => true => queue is empty

DISPLAY:

- Display is used to display the elements in queue

display( )

```
 if( front = = NULL)
    write " Queue is empty"
else
   p = front
   while ( p ! = NULL)
       Write p → data
       p = p → next
   end while
```

SINGLE LINKED LIST PROGRAMS:

1. Function to count the number of nodes in a linked list.

Procedure:

- head is a pointer to the first node. P is also a pointer to 1st node. Initialize count to zero.
- While p is not null, increment value of count and move the p to next position.
- When p points to null, return the value of count.

Algorithm:

```
int Count( )
p = head
while ( p ! = NULL)
    count ++;
    p = p → next
end  while
return  count
```

2. Function to Search an element from a linked list

Procedure:

- P is a pointer which points to 1st node in list.
- While p is not null, compare the value in p with x. If it matches, return 1. If it doesn"t matches move p to next position thereby compare all elements.
- If no match found with any of elements, return 0.

Algorithm:

```
int Search( node  * head, int x)    // head is a pointer to 1st node, x is the element to be searched
  p = head
  while ( p ! = NULL)
     if ( p → data = = x)
          return 1        // (i.e) element is present
     end if
     p = p→ next
  end while
return 0      //(i.e) element is not present
```

3. Function to Merge two linked list

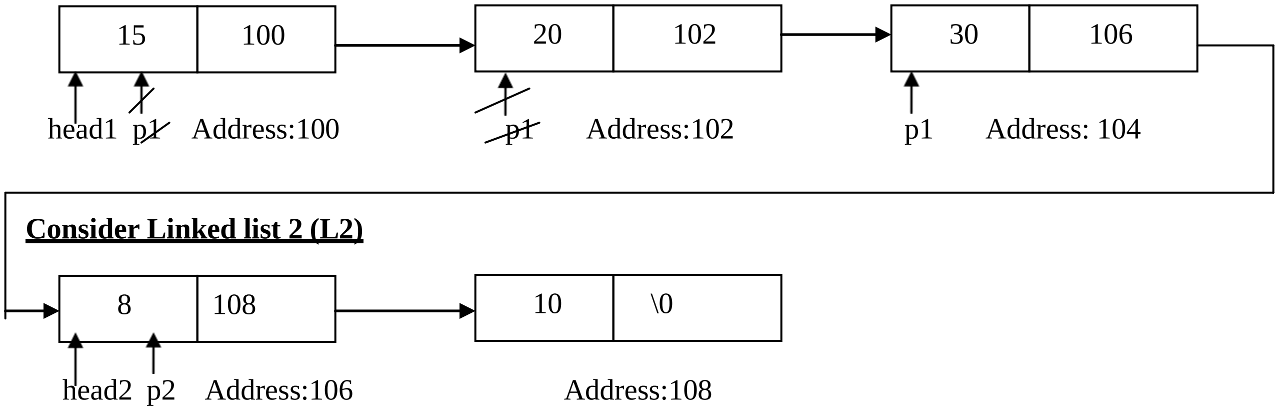
Procedure:

- p1 points to 1st node in L1 and p2 points to 1st node in L2.
- Move the pointer p1 to the last node in L1
- Link the last node of L1 to 1st node of L2.
- Return the merged list

Algorithm:

```
Merge( node * head1, node * head2)     // head1 and head2 are pointers to 1st node in L1 and L2
p1 = head1, p2=head2 // p1 points to 1st node in 1st linked list and p2 points to 1st node in 2nd linked list
while( p1 → next ! = NULL)
    p1 = p1 → next
end while
p1 → next = p2
return ( head1 )
```

Consider Linked list 1 (L1)



4. Function to find the intersection point of two linked list L1 and L2 (i.e) L1 ∩ L2

Procedure:

- Get the count of L1
- Get the count of L2
- Traverse the larger linked list until the distance, such that, number of nodes in L1 and L2 are same.
- Now traverse, both the linked list simultaneously to find the intersection node in L1 and L2

Algorithm:

```
Intersection( node * head1, node * head2)
c1 = count(L1)
c2 = count(L2)
if c1 > c2
    d = c1 – c2 // d is the distance need to be traversed in larger linked list
    get_intersectionnode( )
else
    d = c2 – c1
    get_intersectionnode( )
end if

get_intersectionnode( )
p1 = head1
p2 = head2
p1 moves until distance d
while( p1 ! =NULL && p2 ! = NULL)
     if ( p1 →data = = p2 → data)
          return p1 → data
     end if
p1 = p1 → next
p2 = p2 → next
end while
```

5. Function to display n^{th} node in a linked list

Nthnode( )

```
p = head       // head and p are the pointers to the 1st node in the linked list
for (i = 1 ; i < n ; i ++)
      p = p → next
end for
write p → data
```

6. Function to copy one linked list to another.

Procedure:

- head 1 and head2 are the pointers to 1^{st} node in Liked list 1 (L1) and Linked list 2(L2).
- Initially pointer p1 points to 1^{st} node in L1.
- While there is a node in L1, data in L1 is copied to newnode. That newnode is attached to the L2. This process is performed until there is no node in L1

Algorithm:

```
p1 = head1
head2 = NULL
while( p1 ! = NULL)
     create newnode
     newnode → data = p1 → data
     newnode → next = NULL
     if( head = = NULL)
           head2 = newnode
     else
           tail → next = newnode
           tail = newnode
p1 = p1 → next
end while
```

UNIT – 2 TWO MARKS

1. What is Stack?

- Stack is a linear and static data structure.
- Stack is an ordered collection of elements in which insertion and deletion of elements is performed at only one end called Top.
- Initial condition of the stack Top=-1.
- It is otherwise called as **LIFO** (Last In First Out).

2. What are the various operations that can be performed on stack?

- In Stack, we can perform two operations namely Push and Pop.

Push: Push means inserting a new element into the stack. Insertion can be done by incrementing top by 1.

Pop: Pop means deleting an element from the stack. Deletion can be done by decrementing top by 1.

3. What do you mean by Top in Stack?

- Top is the pointer which always points the top element of the Stack.
- If Top =-1, then Stack is Empty.
- We can insert a new element into stack by incrementing top by 1.
- We can delete an element from stack by decrementing top by 1.

4. What are the applications of Stack? (Nov 13)

i. Matching of nested parenthesis in a mathematical expression.

ii. Conversion of infix to postfix.

iii. Evaluation of postfix.

iv. Towers of Hanoi

v. Shipment in cargo

vi. Arrangement of books/plates in table

5. What are the conditions that should be satisfied in the matching of nested parenthesis?

Matching of nested parenthesis should satisfy the following two conditions:

i. Number of opening parenthesis should be equal to the number of closing parenthesis.

ii. The closing parenthesis should be preceded by the matching opening parenthesis.

6. Define Expression, Operator and Operand?

Expression: Expression is the collection of operators and operands.

Operator: Operator is the symbol which performs mathematical operations on variables.

Operand: Operand is the constant or variable whose values may be int, real, char.

7. What are the various types of operators based on number of Operands?

Based on number of operands, there are three types of operators.

i. **Unary Operator:** Unary operator depends upon only one operand.

ii. **Binary Operator:** Binary operator depends upon two operands.

iii. **Ternary Operator:** Ternary operator depends upon three operands.

8. What are types of Expression?

Expression is classified as three types according to position of operator with respect to the operands. They are:

i. **Infix:** The operator is placed in between two operands. E.g.: A+B.

ii. **Postfix:** The operator is placed after the operands. E.g.: AB+.

iii. **Prefix:** The operator is placed before the operands. E.g.: +AB.

9. What do you mean by Hierarchy of Operators?

The order in which the arithmetic operators are evaluated is called Hierarchy of Operators. There are two types. They are:

i. High Level Priority.

ii. Low Level Priority.

10. What is Queue?

- Queue is a linear and static data structure.
- Queue is an ordered collection of elements in which we can insert an element at one end called Rear and delete an element at another end called Front.
- Initial condition of the stack Rear=Front=-1.
- It is otherwise called as **FIFO** (First In First Out).

11. What are the various operations that can be performed on Queue?

In Queue, we can perform two operations namely Insertion and Deletion.

Enqueue: Insertion can be done by incrementing Rear by 1.

Dequeue: Deletion can be done by incrementing Front by 1.

12. What do you mean by Queue empty?

If Queue has no data, then it is called as Queue Empty.

Queue may be empty on two conditions.

i. Front = Rear = -1 and

ii. Front = = Rear while performing dequeue.

13. What is Linked list?

- Linked list is a dynamic and linear data structure.
- Linked list is an ordered collection of elements in which each element is referred as a ***Node***.
- Node has two parts. Data field and link field.

Data	link

Node

(Or)

Data	address of next node

Node

14. What are various fields in a Linked list?

Each node has two fields namely

i. Data field or Information field and
ii. Address field or Link field.
Data Field: Data field contains the actual data.
Address Field: Address field contains the address of next node in list.

15. Define Head Pointer?

- Head pointer is the pointer which always points the first node in the list.
- Head pointer holds the address of the first node.
- Using Head pointer only we can move from first node to last node.

16. List the advantages and drawbacks of Double linked list.

Advantages:

- The elements in double linked list can be traversed both in forward and in backward directions
- The deletion operation in double linked list is more efficient if pointer to the node to be deleted in given

Drawbacks:

It requires extra space for storing the previous pointer.

Insertion and deletion takes more time than single linked list

17. What are the types of Linked List?

There are three types of Linked list. They are
i. Single Linked list, ii. Double Linked list iii. Circular Linked list.

18. What is Single linked list?

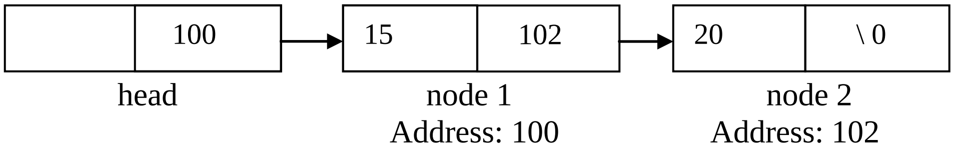
- In Single linked list, each node has data to be stored and one link to the next node.
- In Single linked list, we can move only in one direction from head pointer to Null pointer.

Each node has two fields namely

i. Data field or Information field and
ii. Address field or Link field.

It is otherwise called as Linear Linked list.

Representation of node in a Single linked list



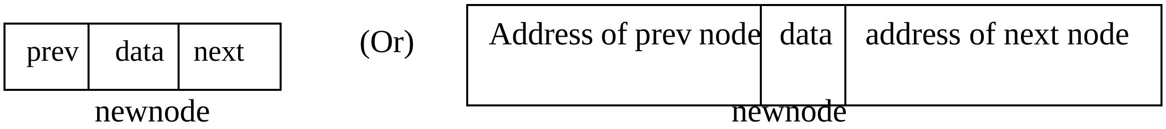
19. List out the application of a linked list

- Manipulation of polynomials
- Implementation of Stacks and Queues
- Sparse matrices

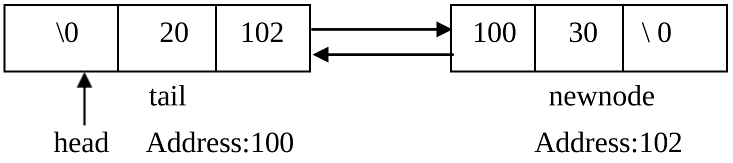
20. What is Double Linked list?

- Double linked list is a linked list where the elements can be traversed both in forward direction and also in backward direction

Representation of node in a Double linked list



Example:



21. What is Circular Single Linked list?

- In Circular Single linked list, the last node is connected to the first node.
- In Circular Single linked list, we can move only in one direction from head pointer to Null pointer.

22. List the advantages and drawbacks of circular linked list.

Advantages:

- If we are at a node, we can to go to any node.But in linear list,it's not possible to go to previous node.
- It saves time in traversing from last node to 1st node than in double linked list which traverse entire list backwards

Drawbacks:

- It is not easy to reverse a list
- Accessing previous element cannot be done in single step.

23. What is Circular Double Linked list?

- In Circular Double linked list, the last node is connected to the first node.
- In Circular Double Linked list, we can move in both the direction from head pointer to Null address or vice versa.

24. List out the advantages in using a linked list.

- It is a dynamic data structure
- It is not necessary to specify the number of elements in a linked list during its declaration
- Insertion and deletion is done at any place in linked list easily, only when it is needed, which avoids wastage of memory

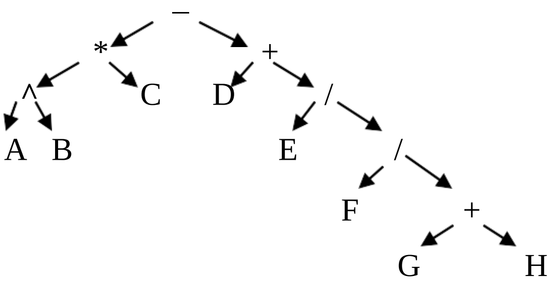
25. List out the disadvantages in using a linked list.

- Searching a particular element in list is difficult and time consuming
- Additional storage space is used for storing pointers

26. State the difference between arrays and linked list.

Arrays	Linked List
Size of any arrays is fixed.(Static DS)	Size of a linked list is variable based on requirement(Dynamic DS).
It is necessary to specify the number of elements during the declaration.	It is not necessary to specify the number of elements during the declaration.
Insertion and deletion of element at random position in array is complex	Insertion and deletion of element at random position in linked list is simpler.

27. Convert the following infix expression to postfix expression/reverse polish notation
(a) A^B*C-D+E/F/(G+H)
Expression tree:



Postorder traversal gives a postfix expression => AB^C*DEFGH+//+–

b) (A+B)* (C^(D–E)+F) – G

- If parenthesis present, operator with high priority is moved out of parenthesis.
- If nested parenthesis present, innermost parenthesis is first executed.

(A+B)   * (C^(D–E)+F) – G

=> (AB+) *** ((**C^(**D–E**) + F) – G)

=> (AB+) * (**(C ^ (DE-) + F)** – G)

=> (AB+) * (**(CDE–) + F)^** – G)

=> (AB+) * (**(CDE– F)^+** – G)   =>(AB+) * (CDE–F^+G–)

=> **AB+CDE–F^+G–***

28. List the Applications of queue

- Printing
- CPU scheduling
- Mail service
- Elevator
- Keyboard buffering

29. Mention the advantages of representing stacks using linked list than arrays.

- It is not necessary to specify the number of elements to be stored in a stack during its declaration.
- Insertions and deletions can be handled easily and efficiently.
- Linked list representation of stacks can grow & shrink in size without wasting the memory space, depending upon the insertion and deletion that occurs in the list.
- Multiple stacks can be represented efficiently using a chain for each stack.

30. Define Priority Queue.

- Priority Queue is the ordered collection of elements which are placed on the priority.
- Insertion and deletion are done according to the priority.

There are two types of Priority Queue. They are

i. Ascending Priority Queue and
ii. Descending Priority Queue.

31. Define Dequeue.

- Dequeue is otherwise called as Double ended queue.
- In Dequeue, we can insert and delete at both ends either front or rear.

32. Define Ascending Priority Queue.

- In this elements are placed in ascending order.
- The first smallest element is placed in first position and second smallest element in second position and so on.
- The new data item is inserted in priority queue without affecting the ascending order of queue.

33. Define Descending Priority Queue.

- In this elements are placed in descending order.
- The first highest element is placed in first position and second highest element is placed in second position and so on.
- The new data item is inserted in priority queue without affecting the descending order of queue.

34. Define Input restricted Dequeue.

- It means we can insert the element only at one end and delete the elements at both ends.

35. Define Output restricted Dequeue.

- It means we can insert the elements in both ends and delete the elements at only one end.

36. List the basic operations carried out in a linked list.

The basic operations carried out in a linked list include.

- Creation of list, Insertion of an element, Deletion of an element, Searching of an element, Traversal of the list.

37. What are the different ways to implement list?

- Array implementation of list
- Linked list implementation of list

UNIT – 3

NON - LINEAR DATA STRUCTURES:

- In Non-linear data structure, elements will have a non-linear relationship with the other elements.
- The elements are said to be Non-linear, if there is a possible way to skip some of the elements in data structure, while traversing the elements.
- Example: Trees, Graph

TREES:

A tree *T* is defined as a finite set of one or more nodes such that

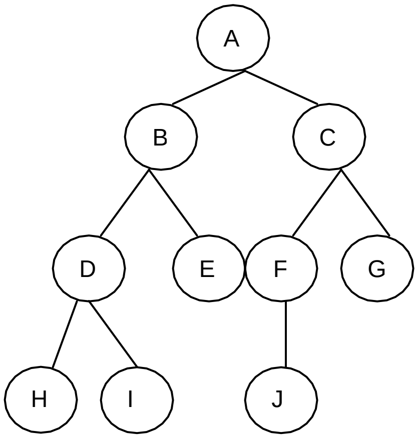
✓ There is one specially designated node called ROOT.

✓ The remaining nodes are partitioned into a collection of sub-trees (T_1 ,T_2 ,T_3……T_n) of the root, each of which is also a tree.

Properties of a Tree

1. Any node can be the root of the tree and each node in a tree has the property that there is exactly one path connecting that node with every other node in the tree.
2. The tree in which the root is identified is called a rooted tree; a tree in which the root is not identified is called a free tree.
3. Each node, except the root, has a unique parent.
4. A tree *T* can never be empty.

Example



LEVEL	Height or Depth
0	1
1	2
2	3
3	4

Fig. Tree

<u>**Tree terminologies:**</u>

- The nodes of a tree have a parent-child relationship. The **root** does not have a parent; but each one of the other nodes has a **parent node** associated to it.
- A node which doesn‟t have children is called a <u>**leaf node**</u> or <u>**terminalnodes**</u>.
- A line from a parent to a child node is called a <u>**branch**</u>**.** If a tree has **n** nodes, one of which is the root there would be ***n-1 branches***.

Degree of a node:

- The number of sub-trees of a node is called its degree. The degree of A,B,C,D is 2, F is 1 and E,G,H,I, J is zero.

The degree of a tree :

- The degree of the tree is the maximum degree of the nodes in the tree.

Siblings:

- Nodes with the same parent are called siblings. Here B & C, D & E, F & G, H & I are all siblings.

Level of the node:

- The level of a node is defined by initially letting the root at level zero. If a node is at level L, then its children will be at level L+1.

Height of the tree:

- The height or depth of a tree is defined to be the maximum level of tree(L) plus 1.
- Height of tree H= *L* +1.

Forest:

- A set of trees is called forest; if we remove the root of a tree we get a forest. In the below fig, if we remove A, we get a forest with three trees.

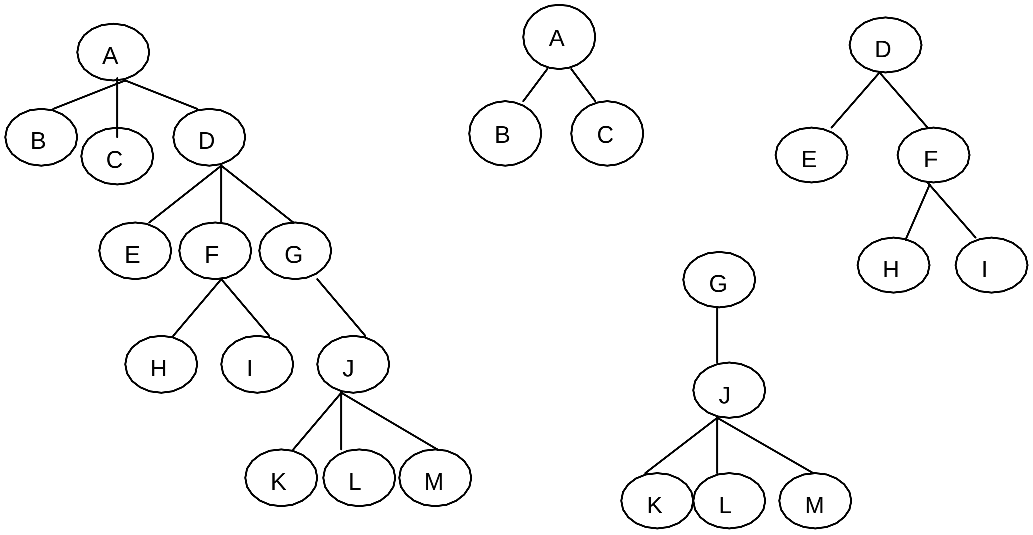


Fig.Forest (Sub-trees)

BINARY TREES

- A binary tree is a tree, which is, either empty or consists of a root node and two disjoint binary trees called the left sub-tree and right sub-tree.

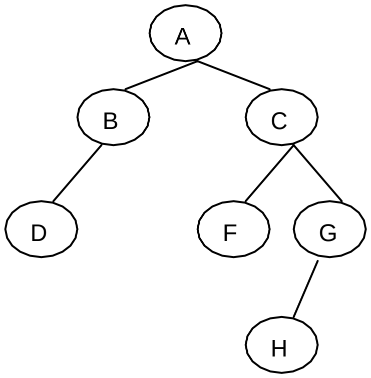


Fig. Binary Tree

- In a binary tree, ***no node can have more than two children***.
- So every binary tree is a tree, not every tree is a binary tree.
- A tree can never be empty but a binary tree may be empty.

Properties of Binary tree:

- In any binary tree, the maximum number of nodes on level L is 2^L , where $L \geq 0$.
- The maximum number of nodes possible in a binary tree of height H is 2^H -1.
- The minimum number of nodes possible in a binary tree of height H is h.
- The height of a complete binary tree with n number of nodes is $\log_2(n+1)$.

Full binary tree

- A **Full binary tree** is a binary tree in which all intermediate nodes have same degree as 2 and all leaves are at the same level.

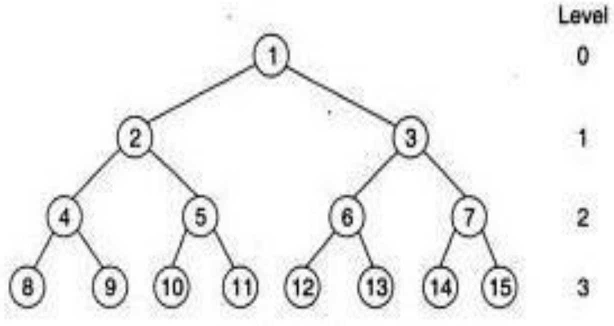


Fig. Full binary tree

Complete binary tree

- A binary tree is said to be a complete binary tree if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible

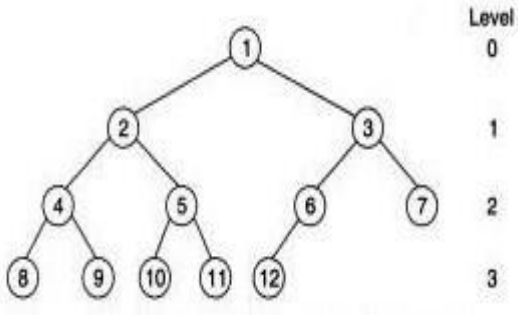


Fig. Complete binary tree

Representation of a Binary Tree:

1. Linear Representation of a Binary Tree

- The linear representation of implementing a binary tree uses a one-dimensional array of size (2^H)-1 where *H* is the depth or height of the tree.
- This type of representation is static (i.e) memory allocated initially is fixed. Hence the size of the tree cannot be changed dynamically.
- In this representation, the nodes are stored level by level, starting from the zero level where only the root node is present. The root node is stored in the first memory location.

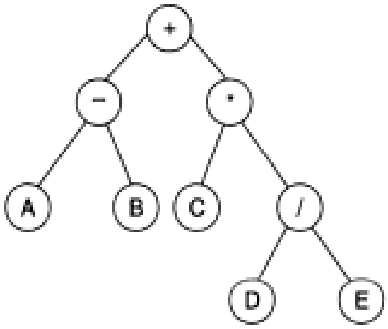


Fig. Binary tree

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
+	−	*	A	B	C	/	·	·	·	·	·		D	E

Fig. Array representation of above binary tree

Following rules can be used to decide the location of any node of a tree in the array assuming that the *array index starts from 1*.

1. Store the root in 1st location of the array.
2. If a node is in location i , where 1<i≤n

   (a) PARENT(i)=[i/2] , for the node when i=1,ther is no parent.
   (b) LCHILD(i)=2*i , If 2*i> *n* then *i* has no left child.
   (C) RCHILD(i)=2*i +1 , If 2*i +1> *n* then *i* has no right child.

Advantages

- Given a child node, its parent node can be determined immediately. If a child node is at location *i* in the array, then its parent is at location i/2.
- It can be implemented easily in languages in which only static memory allocation is directly available.

Disadvantages

- Insertion or deletion of a node causes considerable data movement up and down the array, using an excessive amount of processing time.
- Wastage of memory due to partially filled trees.

2. Linked List Representation

- The problems of sequential representation can be easily overcome through the use of a linked list representation. Each node will have three fields LCHILD, DATA and RCHILD. LCHILD points to left sub-tree and RCHILD points to the right sub-tree of node.

The following figure is an example of linked storage representation of a binary tree.

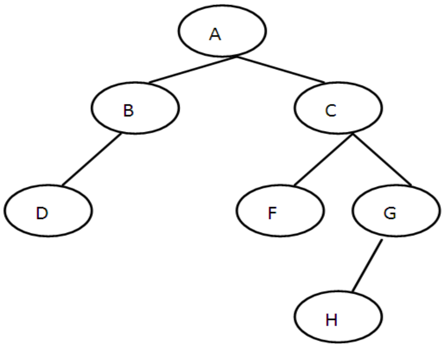


Fig. Binary tree

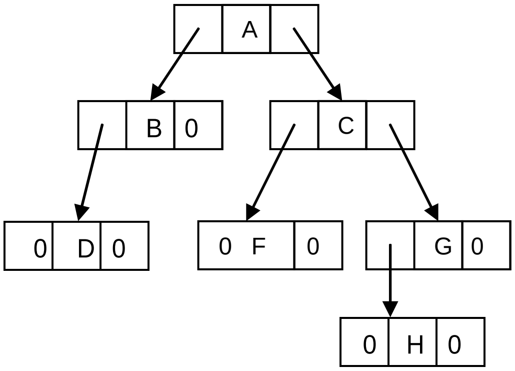


Fig. Linked representation of a binary tree

Advantages:

- If one knows the address of the root node then from it any other node can be accessed.
- It allows dynamic memory allocation

Disadvantages

- Linked representation requires extra memory to maintain pointers.
- Its implementation algorithm is more difficult in languages that do not offer dynamic storage techniques.

EXPRESSION TREE:

An expression tree is a tree which represent the arithmetic expression as a tree.In expression tree,

- Each operator node has exactly two branches.
- Each operand node has no branches; such trees are called expression trees.

The expression tree is constructed as follows:

i) In a given expression, find the priority of each operator.

ii) The operator with lowest priority is chosen as a root/parent node

iii) Left sub-tree of that operator will be expression to the left of operator and Right sub-tree of that operator will be expression to the right of operator.

iv) Repeat the above step until expression tree is formed.

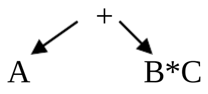
Example:

Consider an expression

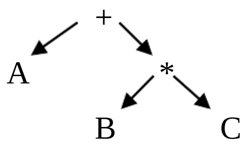
A + B * C

In the above expression, + has low priority compared to *. Hence + taken as root node

Step 1:



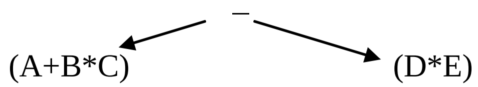
Step 2:



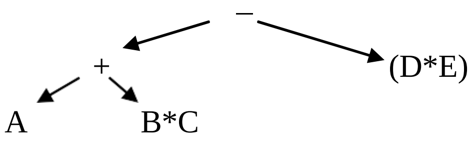
Example 2:

Construct an expression tree for following expression (A+B*C) – (D*E).

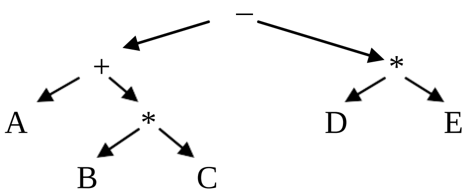
Since – has a low priority, – is chosen as root.



In expression (A + B * C), + has low priority. Hence + is chosen as parent.



In B*C and D*E, there is only one operator, hence that operator act as a parent.

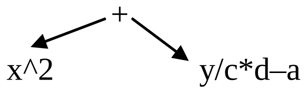


Example 3:

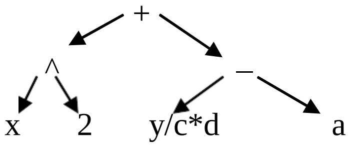
Construct an expression tree for following expression x^2+y/c*d–a.

Priority	Operators	Association
1	^	Right to Left
2	* , /	Left to Right
3	+ , –	Left to Right

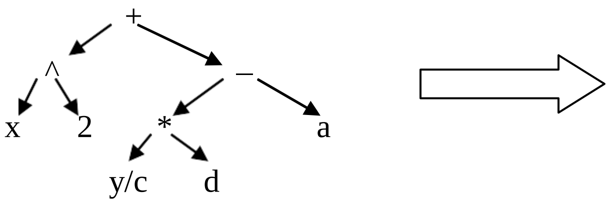
- The operator with lowest priority in expression is + and - . Since the Association is left to right. Expression should be processed from left to right.



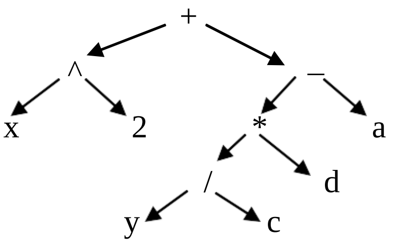
- Since x ^ 2 has only one operator, that operator is chosen as parent. In y/c*d–a, – has low priority. Hence – is chosen as parent. Left child of – is y/c*d and right child is a



- In y/c*d, / and * have same priority. Since association is Left to Right, the expression is processed from Left to Right



- Since y/c has only 1 operator, it act as parent



TRAVERSALS OF A BINARY TREE

- Traversing a tree means processing the tree such that each node is visited only once.
- Let T be a binary tree, such that there are number of different ways to visit the nodes.
- The methods differ primarily in the order in which they visit the nodes.
- The three different traversals of T are Inorder, Preorder and Postorder traversals.
- In C, each node is defined as a structure of the following form:

```
struct node
  {
      int data;
      struct node *lchild;
      struct node *rchild;
  }
typedef struct node NODE;
```

i) INORDER TRAVERSAL:

- It follows the strategy **Left-Root-Right**.
- In this traversal, if T is not empty, we first traverse (in order) the left sub-tree; Then visit the root node of T and then traverse the right sub-tree.
- *We may note that expression is in infix notation. The in-order traversal produces a left expression then prints out the operator at root and then a right expression.*

Algorithm

***Inorder*(*T*)**

```
//Initially T points to the root node of binary tree, where each node has three fields lchild, data and rchild.
if T ≠ NULL then
      call Inorder(T→ lchild)
      print T →data
      call Inorder(T→ rchild)
end if
```

Example 1:

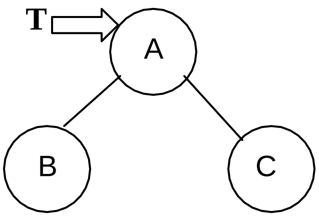
Recursion tree for the following tree:

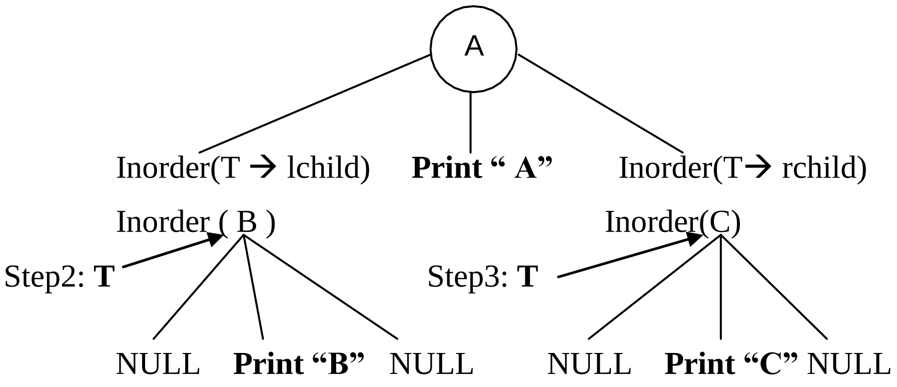
A recursion tree is a tree, which

i) Traverse a tree from *root node to the bottom* till the leaf node

ii) Then, traverse the tree from *left to right.*

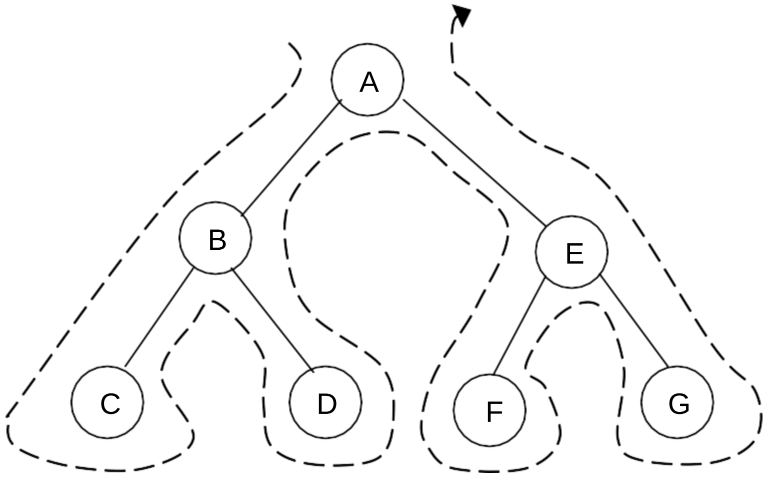
step1





- In above recursion tree, if we traverse from root to bottom and from left to right,We get Inorder of the given tree is **BAC**

Example 2:



The Output is : C → B → D →A →F → E→ G

Example 3:

Consider the binary tree given in the following figure.

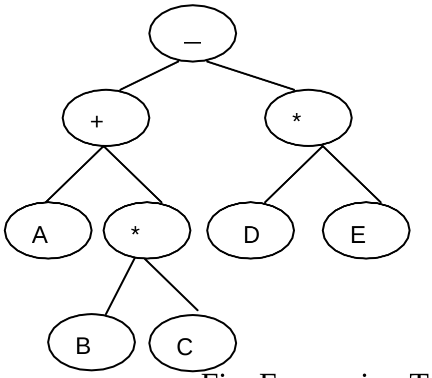


Fig. Expression Tree

Tree, T initially is rooted at „– „;

- Since left(T) is not NULL; currently T points to „+‟;
- Since left(T) is not NULL; currently T points to „A‟;
- Since left (T) is NULL; we print data in T (i.e) **A**.

- We then print root T i.e. „+‟. Now T points to „+″
- We now perform in-order traversal of right (T) current T becomes rooted at„*‟.
- Since left(T) is not empty; current T becomes rooted at „B″ since left(T) is empty; we visit data in T i.e. **B**; check for right (T) which is empty, therefore we move back to parent tree. We then print root T i.e. **„*‟**.
- Now Inorder traversal of right(T) is performed; which would give us „**C**‟.We then print root – Then, in-order traversal of right(T) is performed which would give us **„D‟, „*‟** and „**E**‟.

Therefore the complete listing is: A+B*C-D*E

ii) PREORDER TRAVERSAL:

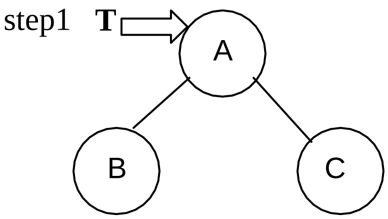
- It follows the strategy **Root-Left-Right**.
- In this traversal, if T is not empty, we print the root node of T, then traverse the left sub-tree and then traverse the right sub-tree.
- ***Preorder traversal of a given tree gives the prefix expression.***

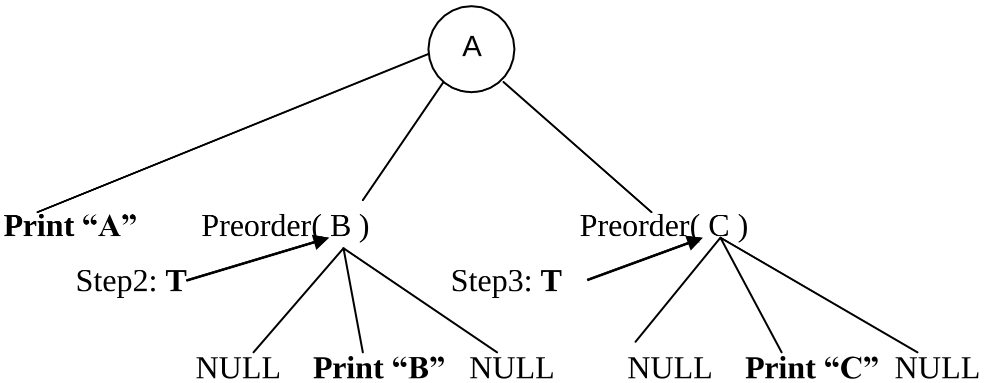
Algorithm

***Preorder*(*T*)**

```
//Initially T points to the root node of binary tree, where each node has three fields lchild, data and rchild.
if T ≠ NULL then
    print T →data
    call  Inorder(T→ lchild)
    call  Inorder(T→ rchild)
end if
```

Example 1:

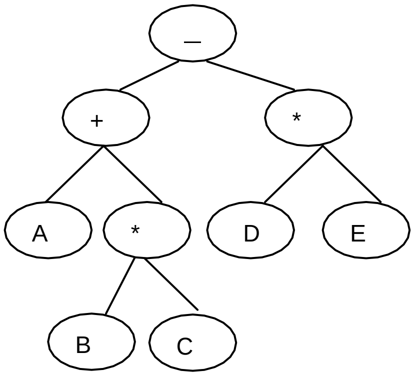




- In above recursion tree, if we traverse from root to bottom and from left to right,We get Preorder of the given tree as **ABC**

Example 2:

Consider the binary tree given in the following figure.



Preorder traversal of above binary tree gives –**+A*BC*DE** **iii)**

POSTORDER TRAVERSAL:

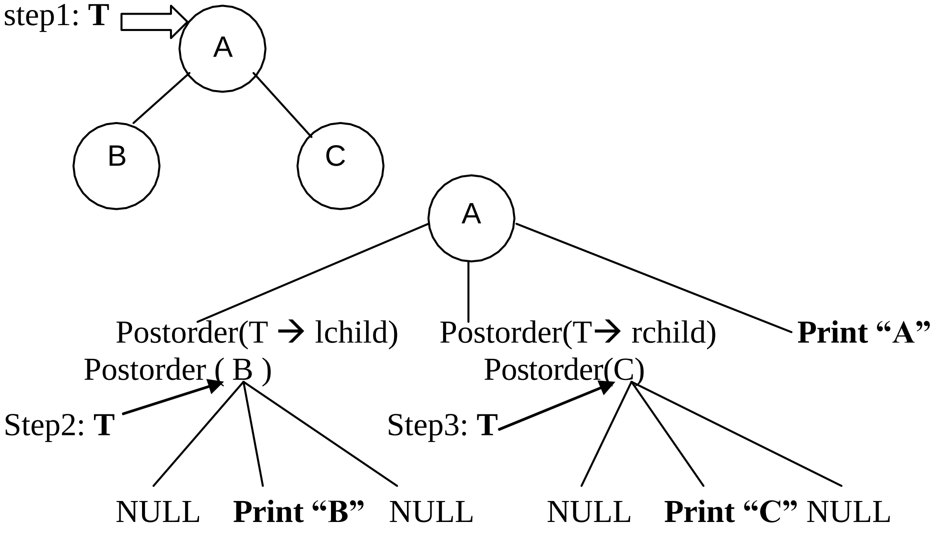
- It follows the strategy **Left-Right-Root**.
- In this traversal, if T is not empty, we traverse the left sub-tree and then traverse the right sub-tree and then we print the root node of T
- ***Postorder traversal of a given tree gives the postfix expression.***

Algorithm

***Postorder*(*T*)**

```
//Initially T points to the root node of binary tree, where each node has three fields lchild, data and rchild.
if T ≠ NULL then
        call Postorder(T→ lchild)
        call Postorder(T→ rchild)
        print T →data
end if
```

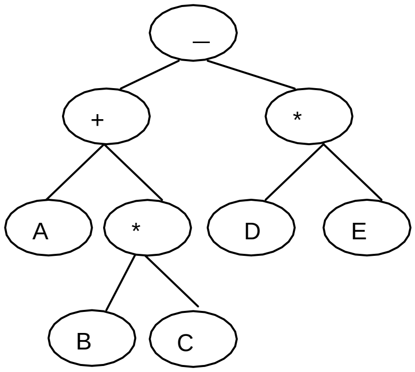
Example 1:



- In above recursion tree, if we traverse from root to bottom and from left to right, we get Postorder of the given tree as **BCA.**

Example 2:

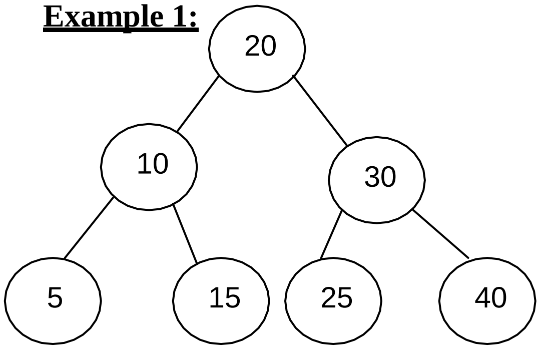
Consider the binary tree given in the following figure.



Postorder traversal of above binary tree gives **ABC*+DE*–**

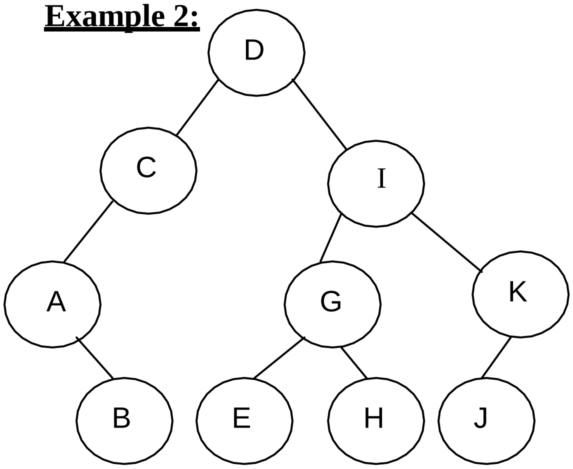
<u>EXAMPLES FOR TREE TRAVERSAL:</u>

<u>Example 1:</u>



Inorder Traversal : 5 10 15 20 25 30 40
Preorder Traversal: 20 10 5 15 30 25 40
Postorder Traversal :5 15 10 25 40 30 20

<u>Example 2:</u>



Inorder Traversal : A B C D E G H I J K
Preorder Traversal: D C A B I G E H K J
Postorder Traversal :B A C E H G J K I D

BINARY SEARCH TREE(BST)

- A binary tree T is termed binary search tree or binary sorted tree, if each node N of T satisfies the following property:
    - *"The value at N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N".*

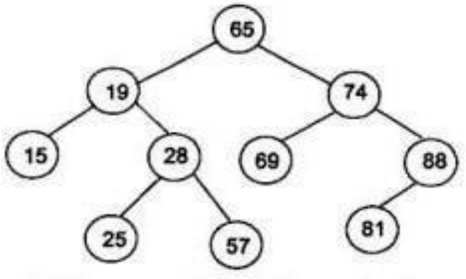


Fig. Binary Search Tree

Binary search tree operations:

The basic operation on a binary search tree(BST) include,

- Inserting data
- Deleting data
- Searching data
- Finding minimum element
- Finding maximum element
- Size of tree (or) Counting number of nodes in tree

i) Insertion of an element in BST:

- The insertion operation on a BST is one step more than the searching operation.
- To insert an element into BST, the tree required to be searched starting from the root node. If „element‟ is found do nothing, otherwise „element‟ is to be inserted at the dead end where the search halts.

Algorithm:

```
InsertBST(node *T, int element)
// „T‟ is the pointer to the root of the tree and „data‟ is the element to be inserted
if (T= = NULL)
    //Create a node and return
     T=newnode
     T→data = element
     T→lchild = NULL
     T→rchild = NULL
else if (element<T→data)
    T→lchild =InsertBST(T→lchild, element)
else if(element> T→data)
    T→rchild =InsertBST(T→rchild, element)
else
    print" element is already exist
end if
return T
```

Example:

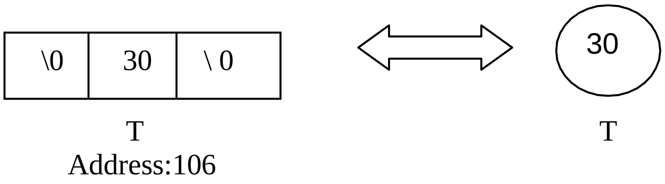
Insert 30

Initially T = NULL

InsertBST(T, 30)

T = = NULL => True

Create newnode and make it as T



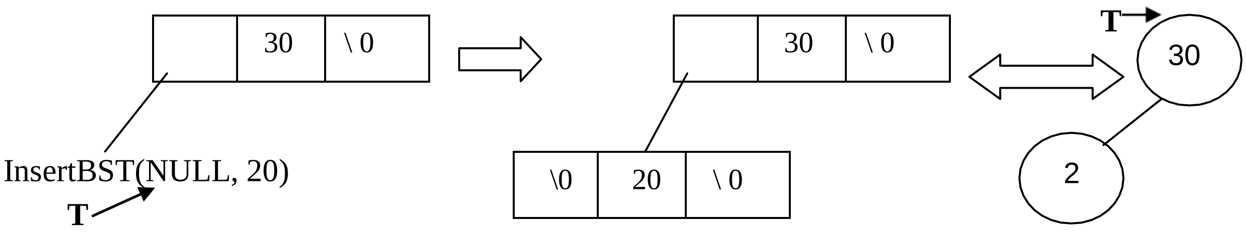
Here the left child of T and right child of T is NULL

Insert 20:

InsertBST(T, 20)

T = = NULL => False

20 < 30, so traverse left sub-tree of 30

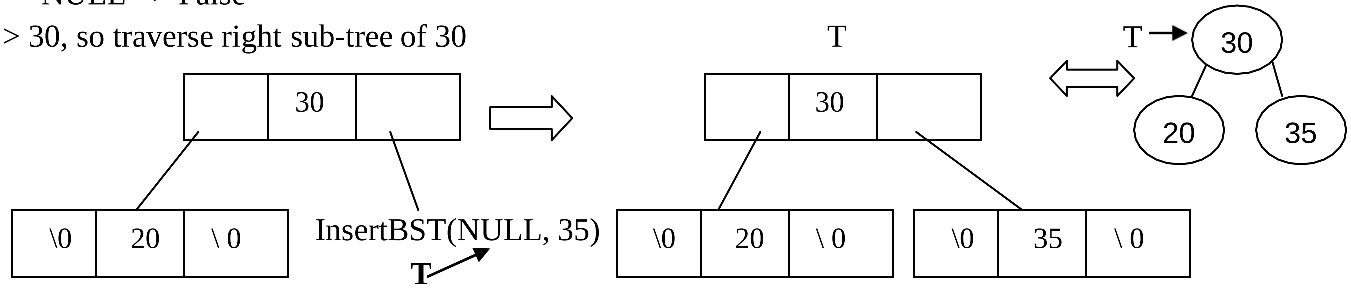


Insert 35:

InsertBST(T, 35)

T = = NULL => False

35 > 30, so traverse right sub-tree of 30



Insert 40

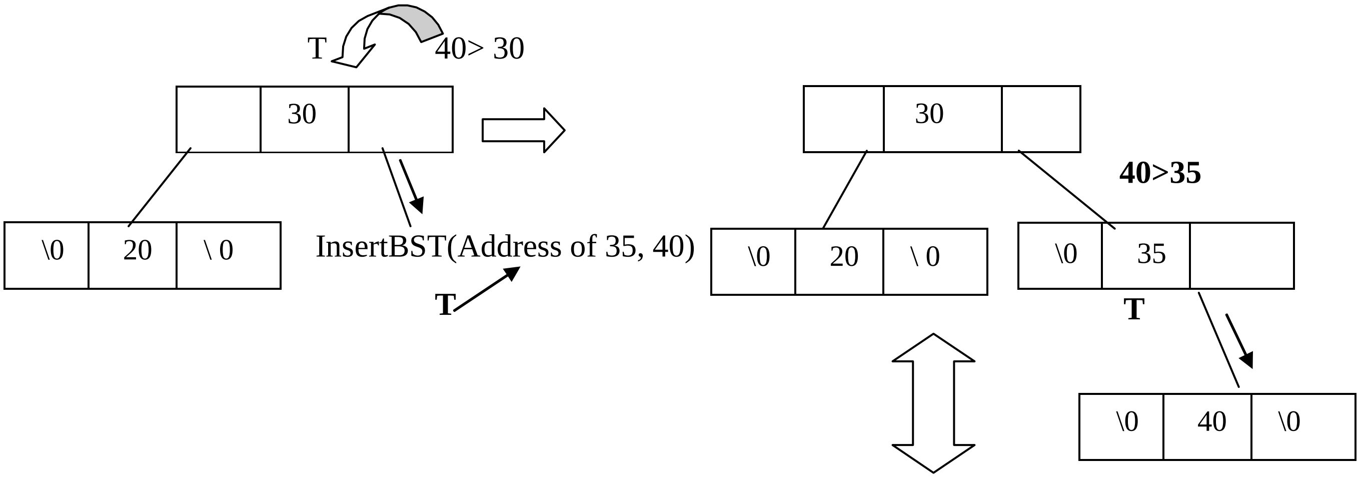
InsertBST(T, 40)

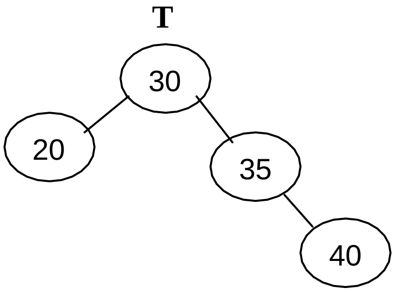
T= = NULL => False

40 > 30, so traverse right sub-tree of 30

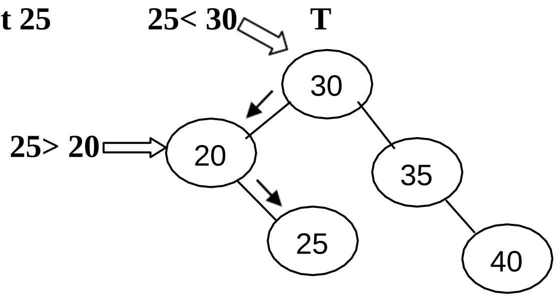
T→ rchild = InsertBST(T→ rchild, 40)

30→ rchild = InsertBST(Address of 35, 40)

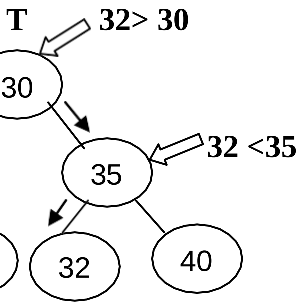




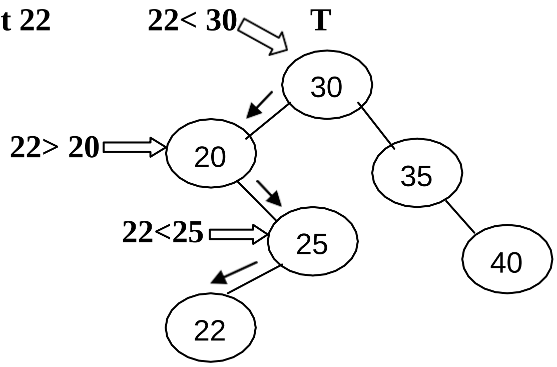
Insert 25



Insert 32



Insert 22



ii) Deletion of an element in BST:

- Deletion is the process whereby a node is deleted from the tree. Only certain nodes in a binary tree can be deleted easily. (i.e) the node with 0 or 1 children can be deleted easily, but, the node with 2 childrens cannot be deleted easily.

The node to be deleted can fall into any one of the following categories

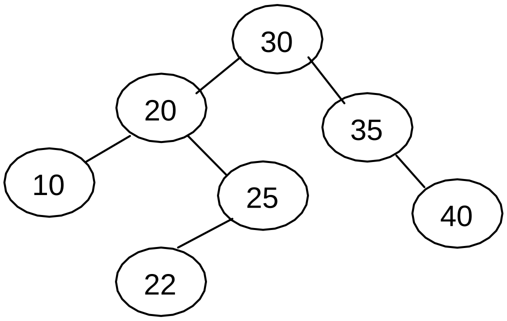
- Case 1: Node may not have any children (ie, it is a leaf node)
- Case 2: Node may have only one child (either left / right child)
- Case 3: Node may have two children (both left and right child)

Algorithm:

DeleteBST(node *T, int element)

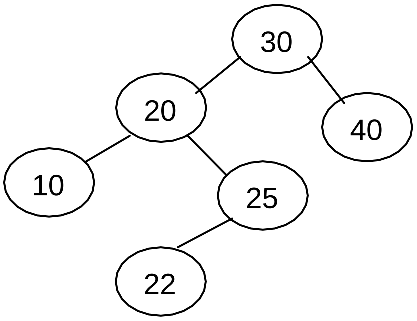
```
// „T‟ is the pointer to the root of the tree and „data‟ is the element to be inserted
if ( T = = NULL)
    print “ root empty”
if( element < T → data)
    T → lchild = deleteBST( T → lchild, element)
else if( element > T → data)
    T → rchild = deleteBST( T → rchild, element)
else
     if( T → lchild = = NULL && T → rchild = = NULL)     // Node to be deleted is leaf node
         free(T)
     else if( T → lchild = = NULL)          // Node to be deleted have only right child
         temp = T→ rchild
         free(T)
         return temp
     else                                   // Node to be deleted have only left child
         temp = T→ lchild
         free(T)
         return temp
     end if
end if
// Node to be deleted have both left and right child
Find the minimum value at the right sub-tree of the node, which is to be deleted.
Replace that minimum value with the node to be deleted.
Free the minimum value node
```

Example: Consider a Binary Search Tree



Case 2: Delete 35.

Since 35 is a node having only one child(right child), the right child of node 35 is copied and it is returned. Node 35 is freed.



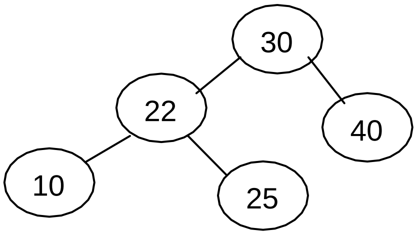
Case 3: Delete 20

Node 20 have both left and right sub- tree.

Hence, the minimum value in right sub tree is found, which is 22.

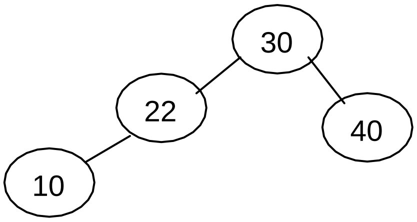
Now 22 should be replaced in place of 20.

Then, prev location of 22 is freed from tree.



Case 1 : Delete 25

Since 25 is a leaf node, it can be directly deleted and the node is freed from memory



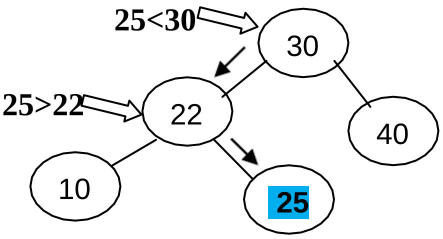
iii) Searching of an element in BST:

Searching an element in BST is much faster than searching data in arrays or linked lists.

- Searching starts from the root of the tree
- If the search key value is less than the value in root, then the search is done at left sub-tree
- If the search key value is greater than root, then the search is done at right sub-tree
- This searching should continue till the node with the search key value or null pointer(end of the branch) is reached.

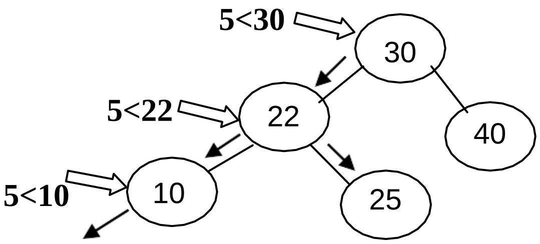
In case null pointer is reached, it is an indication of the absence of the node.

Example: Search an element 25 in following BST



Search element found at specified address

Example: Search an element 5 in following BST



NULL(Element not present)

Algorithm:

SearchBST((node *T, int element)

```
// „T‟ is the pointer to the root of the tree and „data‟ is the element to be inserted
if ( T = = NULL)
    print " root empty"
if( element < T → data)
    SearchBST( T → lchild, element)
else if( element > T → data)
    SearchBST( T → rchild, element)
else
     return T        // The address of node T where the element found is returned
end if
```

iv) Finding a minimum value in a BST

- The node with minimum value can be found in a BST by traversing the tree along the left sub-tree until, left child of the node is not equal to null value.

Algorithm:

Minvaluenode(node * T)

```
while( T → lchild ! = NULL)
    T = T → lchild
end while
return T → data
```

v) Finding a maximum value in a BST

- The node with maximum value can be found in a BST by traversing the tree along the right sub-tree until, right child of the node is not equal to null value.

Algorithm:

Maxvaluenode(node * T)

```
while( T → rchild ! = NULL)
    T = T → rchild
end while
return T → data
```

vi) Size of the tree:

- The size of the tree gives the number of nodes in a given tree.
- It is given by 1 + size of the left sub-tree + size of the right sub-tree

Algorithm:

size(node * T)

```
if T = = NULL
   return 0
else
   return ( 1 + size( T → lchild) + size( T → rchild) )
end  if
```

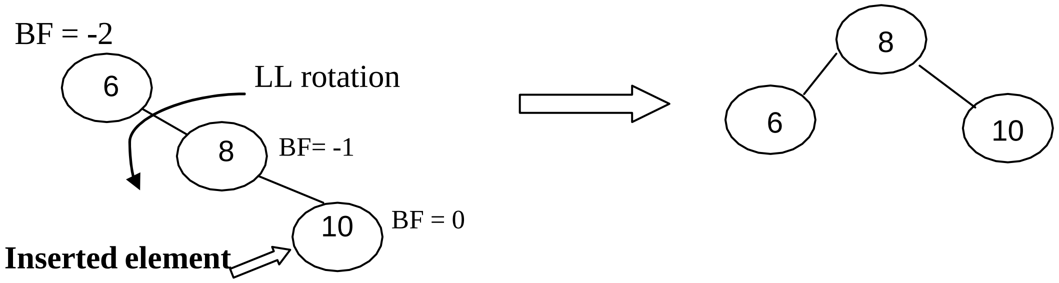
AVL TREES

- AVL tree is a Binary Search Tree(BST), except that for every node in a tree, height of the left sub-tree and right sub-tree differ by atmost 1
- Balance Factor(BF) = Maximum Height(Left sub-tree) - Maximum Height(Right sub-tree)
- BF can be 1, 0 or -1. If not, tree need to be balanced by making either single or double rotations.

ROTATIONS IN AN AVL TREE:

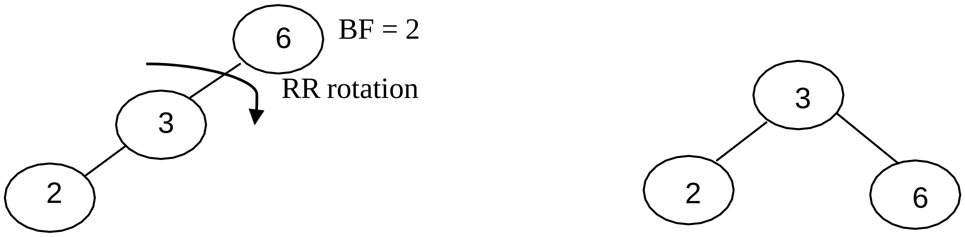
i) LL rotation (or) Single Rotation with Left(or) Rotateleft

- It is performed , If insertion is performed in the right sub-tree of right child of T. We Consider ***T is a node where imbalance occurs.***



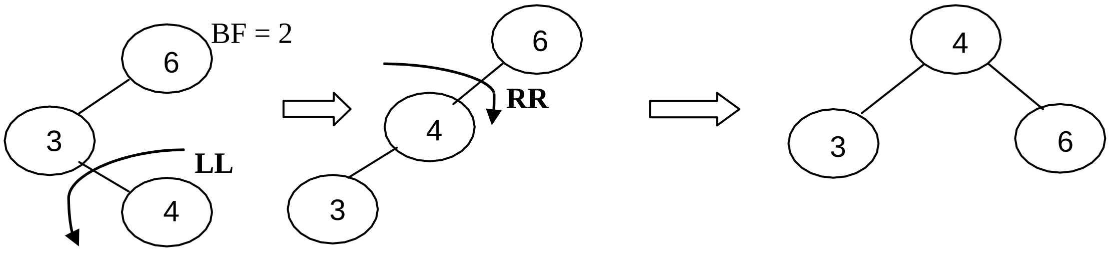
ii) RR rotation (or) Single Rotation with Right(or) Rotateright

It is performed , If insertion is performed in the left sub-tree of left child of T.



iii) LR(Left – Right rotation)

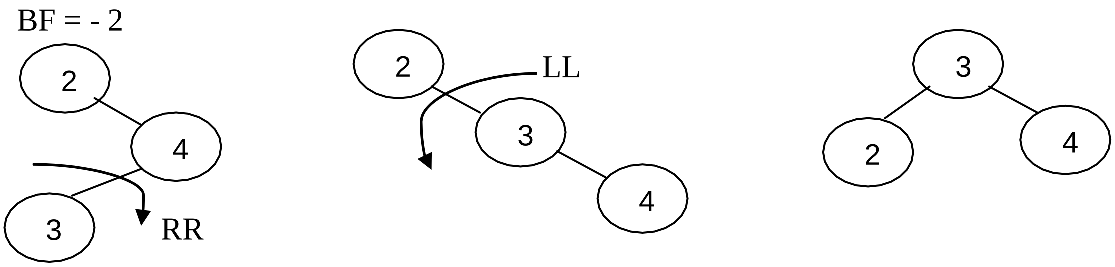
- In this double rotation is performed. Left rotation followed by right rotation.



- LR is performed , If insertion is performed in the right sub-tree of left child of T.

iv) RL(Right – Left rotation)

- In this double rotation is performed. Left rotation followed by right rotation.
- It is performed , If insertion is performed in the left sub-tree of right child of T.



<u>AVL tree rotations Algorithm:</u>

```
Balance Factor(BF) = Maximum Height(Left sub-tree) – Maximum Height(Right sub-tree)
If( BF = = - 2)
    If(element > T→rchild→data)
        T = LL(T)
    Else
        T = RL(T)
If( BF = = 2)
    If(element < T→lchild→data)
        T = RR(T)
    Else
        T = LR(T)
```

LL(node *T)

```
Rotateleft(T)
Return(T)
```

RR(node *T)

```
Rotateright(T)
Return(T)
```

LR(node *T)

```
T→lchild = Rotateleft(T→lchild)
T= Rotateright(T)
Return(T)
```

RL(node *T)

```
T→rchild = Rotateright(T→rchild)
T= Rotateleft(T)
Return(T)
```

Rotateleft(node * x)

```
node *y
y = x → rchild
x → rchild = y → lchild
y → lchild = x
x → height = height(x)
y → height = height(y)
return(y)
```

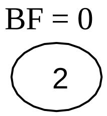
Rotateright(node * x)

```
node *y
y = x → lchild
x → lchild = y → rchild
y → rchild = x
x → height = height(x)
y → height = height(y)
return(y)
```

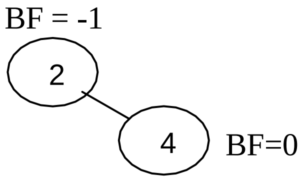
Example:

Insert the elements 2, 4, 1, 42, 10, 12, 9, 22, 15 in a AVL tree

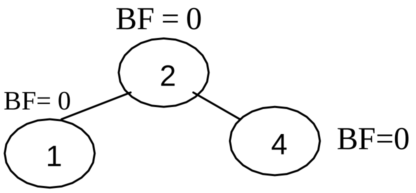
Insert 2



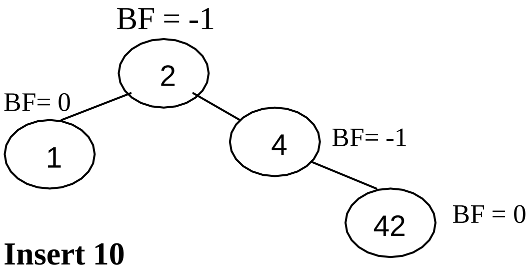
Insert 4



Insert 1

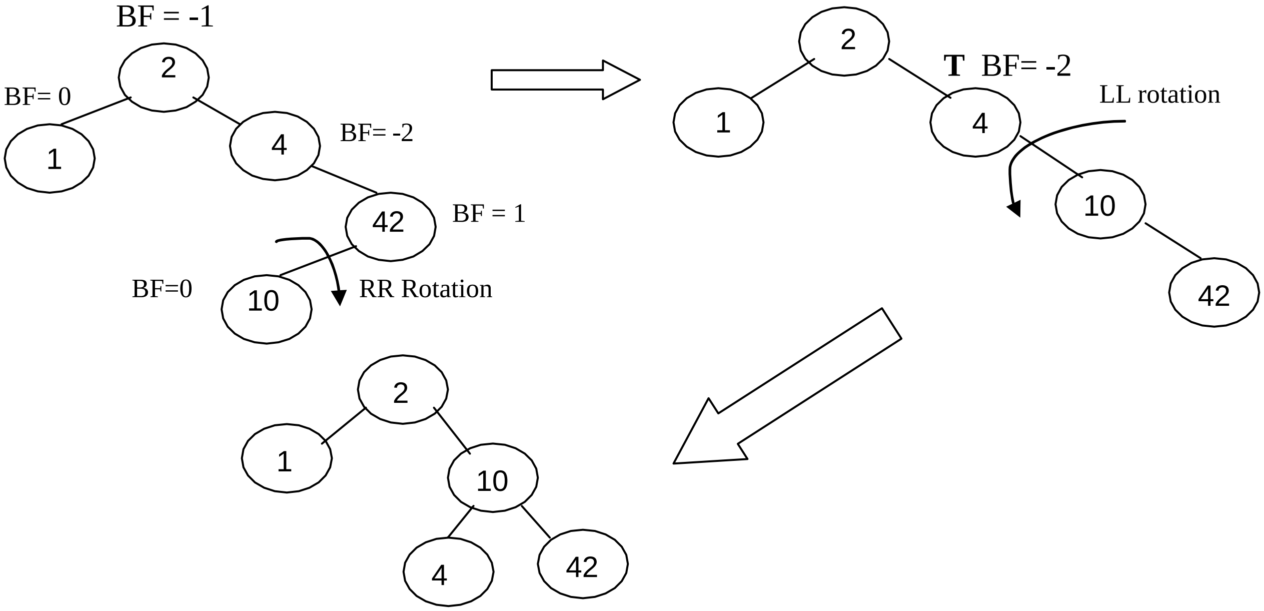


Insert 42



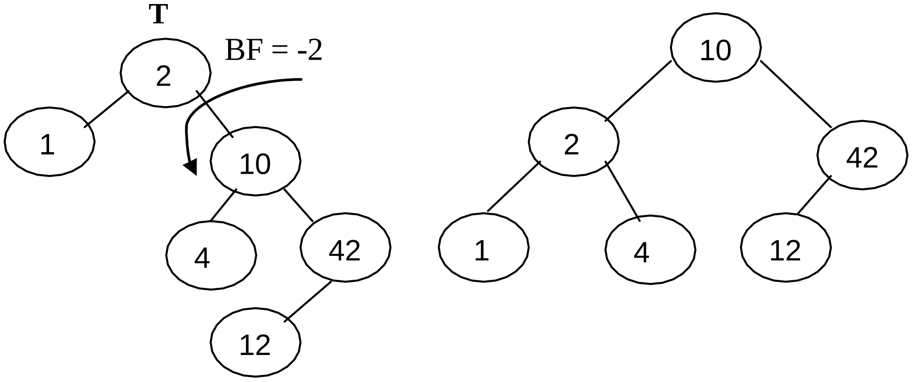
Insert 10

In the below tree, since the element 10 is inserted as left sub-tree of right child of T, **RL rotation is performed**

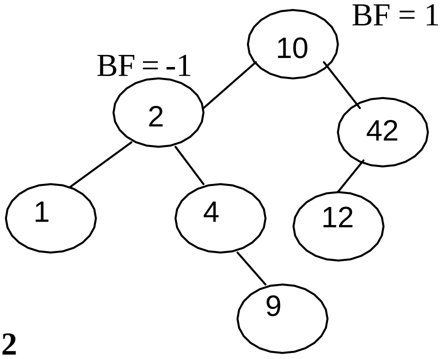


Insert 12

In the below tree, since the element 12 is inserted as the right sub-tree of right child of T, **LL rotation** is performed

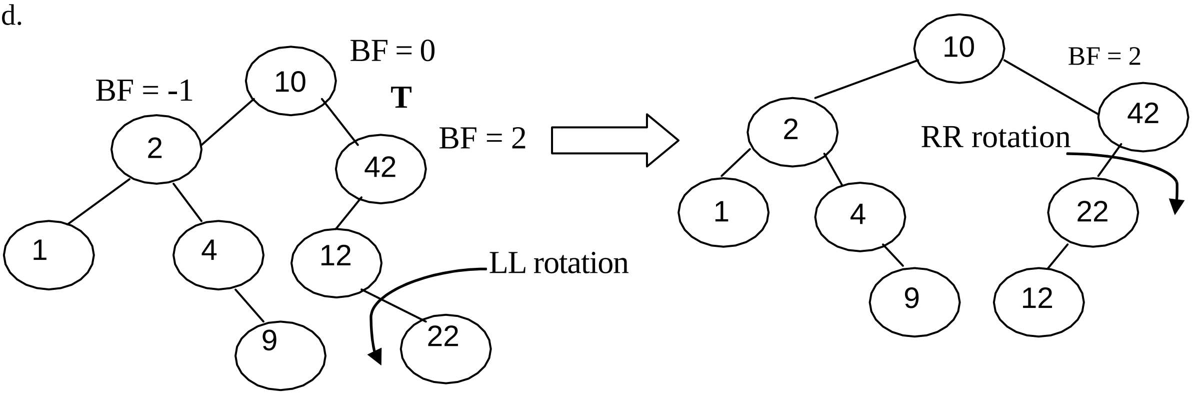


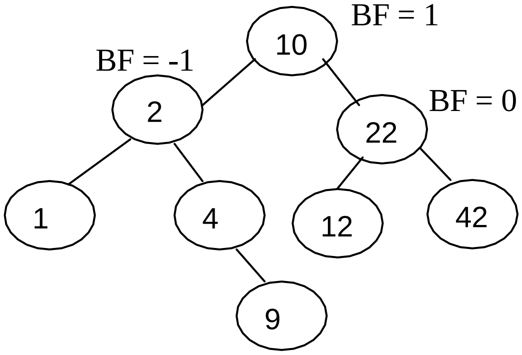
Insert 9



Insert 22

In the below tree, since the element 22 is inserted as right sub-tree of left child of T, **LR rotation** is performed.





B TREE

B tree of order m is a m-way search tree that is either empty or satisfies the following properties

- Root node has atleast 2 children, unless it act as a leaf node
- All internal nodes have $\lceil m/2 \rceil$ to m children
- All leaf nodes must be at same level
- The number of keys is 1 less than the number of children for non-leaf nodes and m/2 to m -1 for leaf nodes

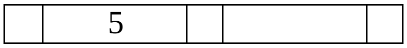
INSERTION AND DELETION IN B TREE OF ORDER M

- m represents a maximum number of pointers in a node.
- For n key , n+1 pointers will be available.

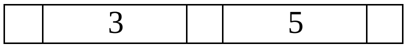
INSERTION:

Insert 5, 3, 2, 9, 7, 8, 6, 10, 12, 1 in a B tree of order 3 and delete 8, 2, 5, 1

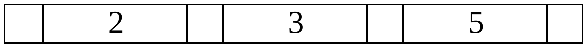
Insert 5:



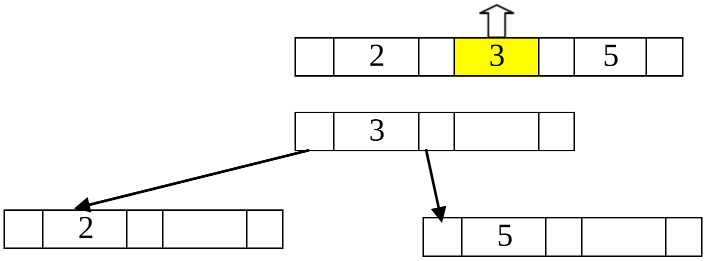
Insert 3:



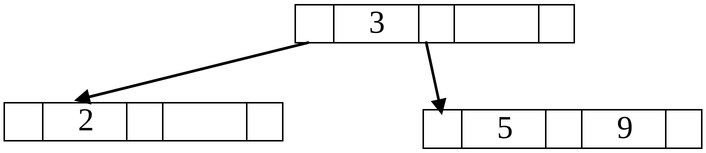
Insert 2:



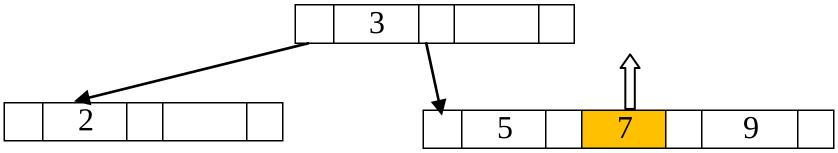
Since B tree of order 3, can store only 2 key elements, we find the middle element and it is moved one level upward



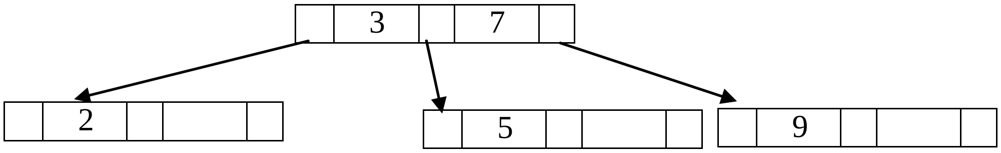
Insert 9:



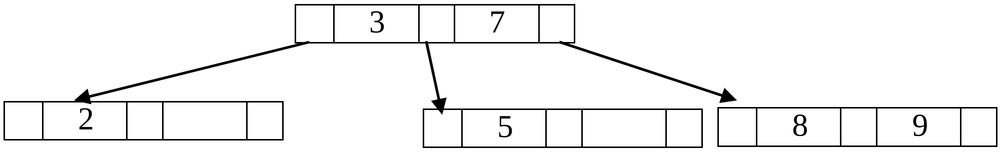
Insert 7:



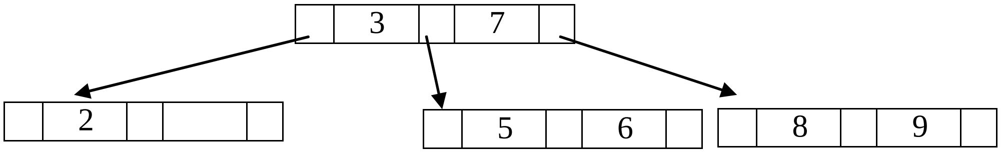
Here the middle element 7 is moved one level upward and 5 and 9 are attached as left child and right child of 7



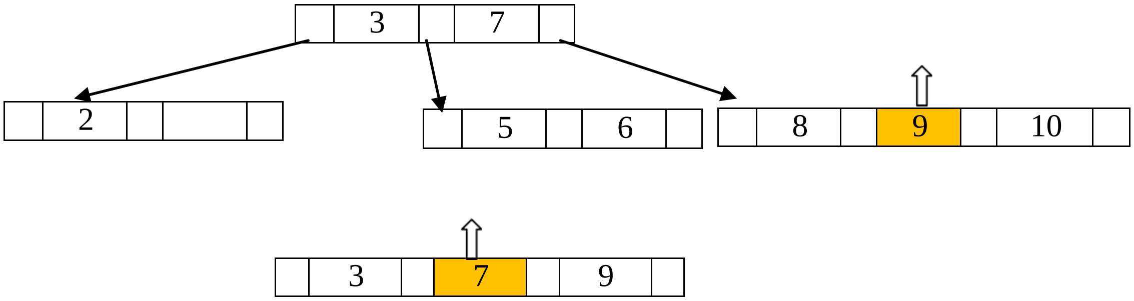
Insert 8:



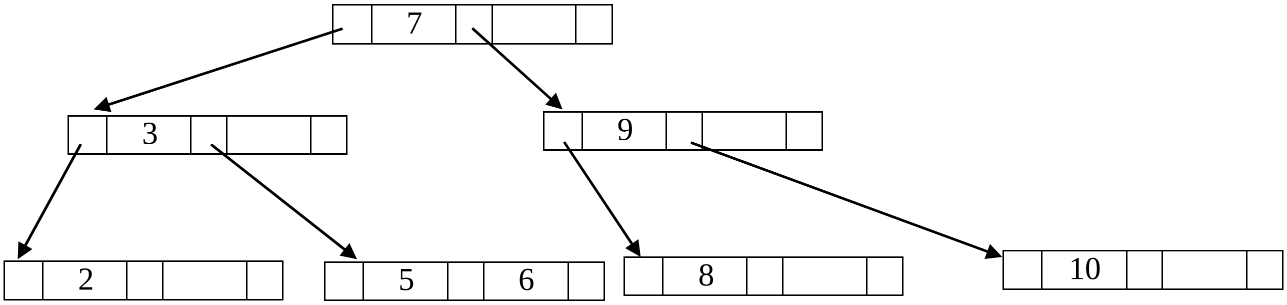
Insert 6:



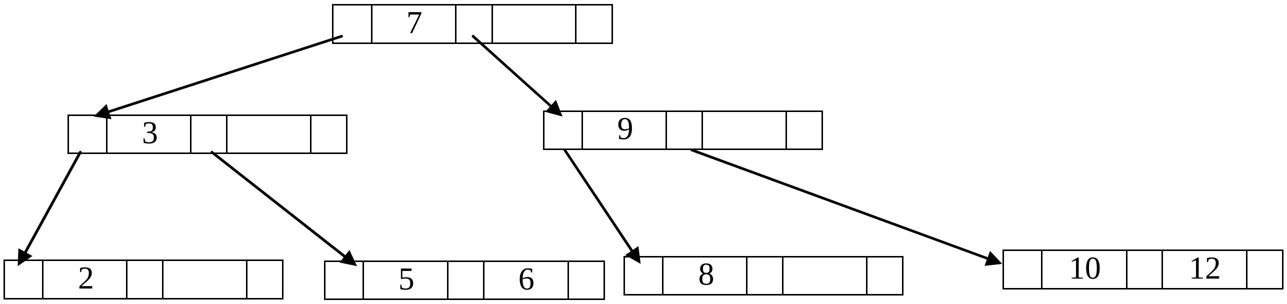
Insert 10:



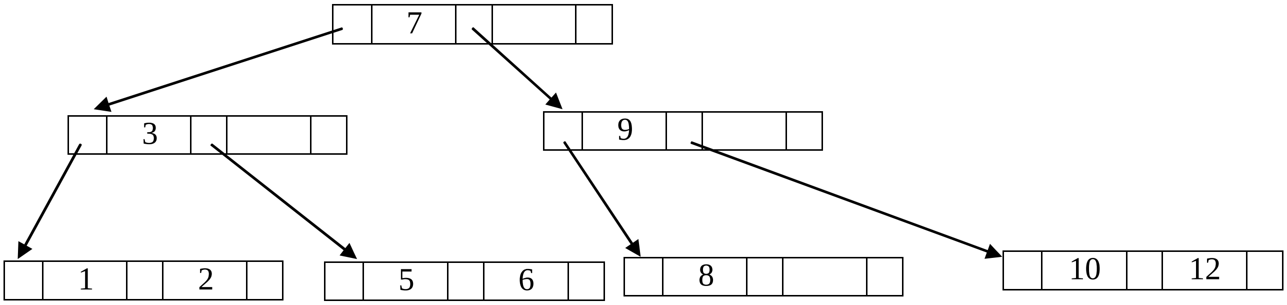
- 7 is moved up level and 3 and 9 are attached as left and right child and other values are attached to their respective parent. 8 and 10 are left and right child of 9.



Insert 12:



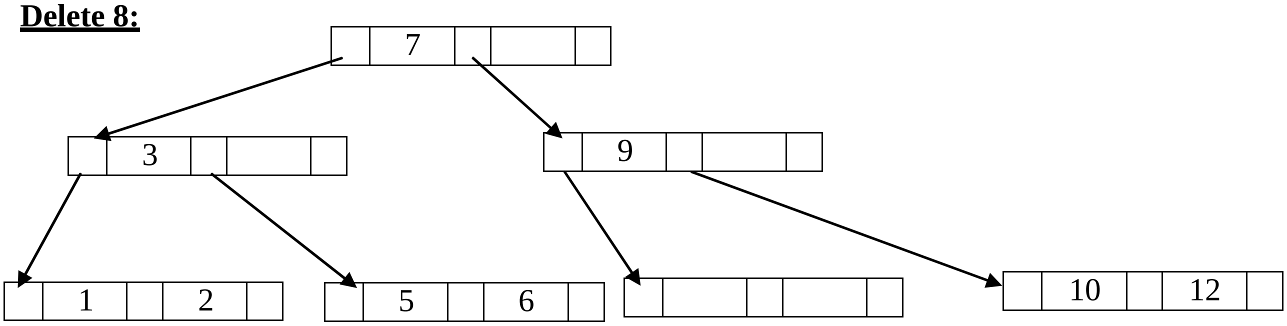
Insert 1:

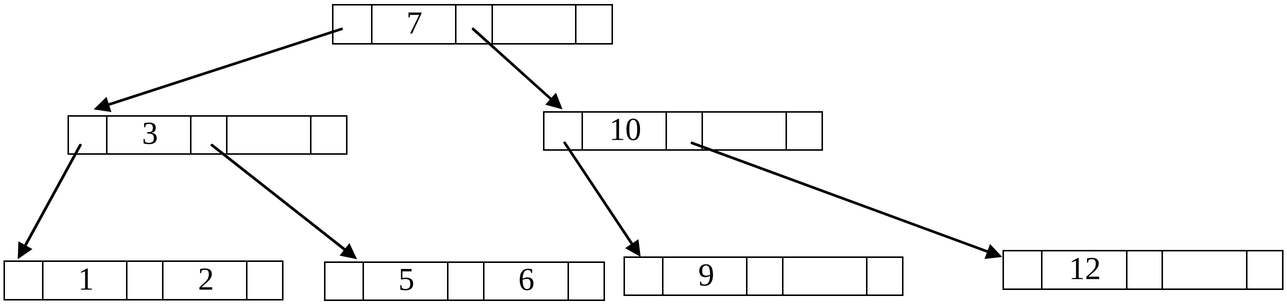


DELETION:

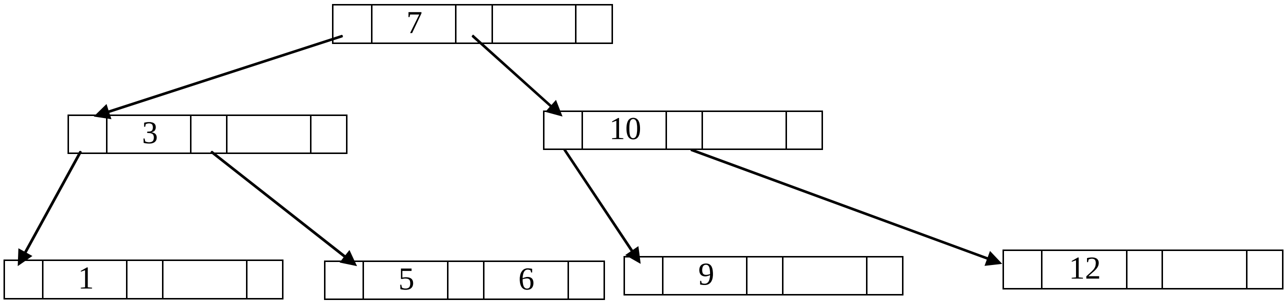
Delete 8, 2, 5, 1

Delete 8:

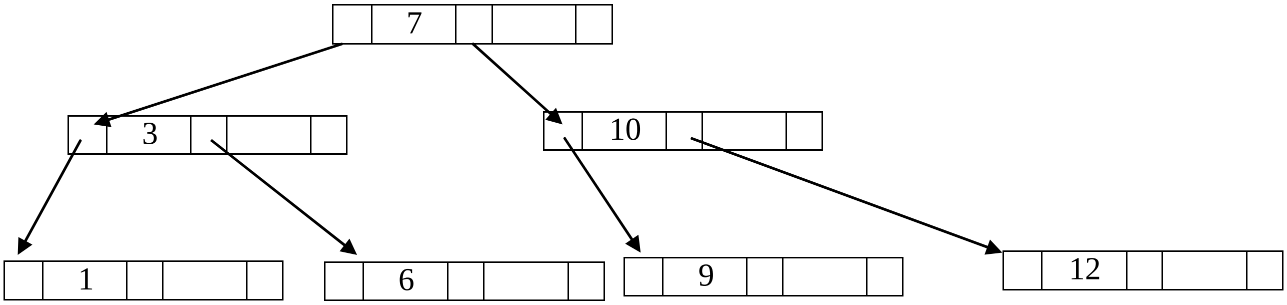




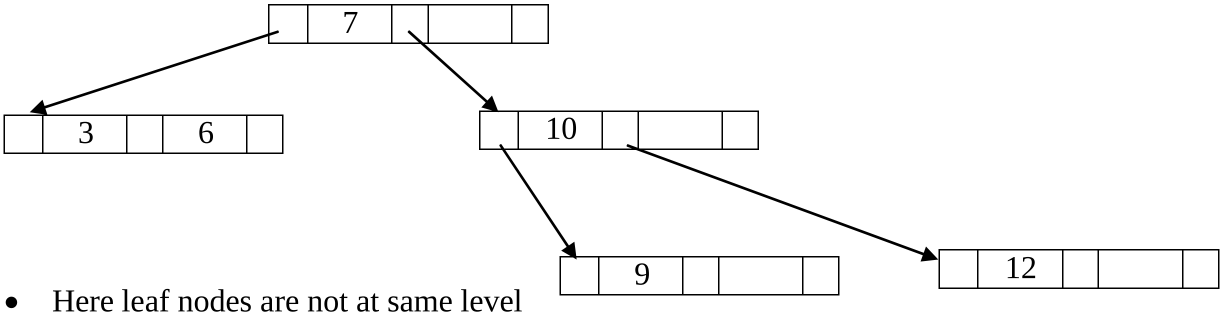
Delete 2:



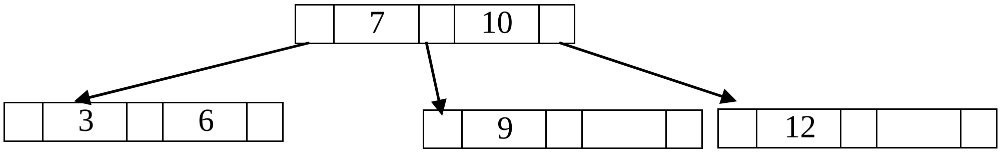
Delete 5:



Delete 1:



- Here leaf nodes are not at same level

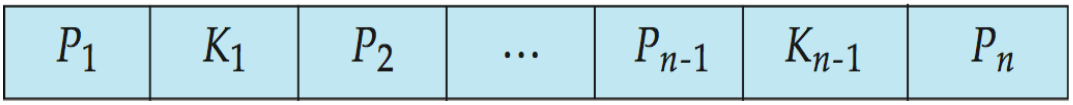


B+ TREE INDEX FILES

A B+ tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Leaf nodes store the actual record. Other nodes only have an index that is used to access that record
- Each internal node has $\lceil n/2 \rceil$ to n children.
- Each leaf node has between $\lceil (n-1)/2 \rceil$ and n–1 values
- Left child of parent is strictly less than the parent node and the right child of parent is greater than or equal to parent node.

B+ Tree Node Structure:



P_1	K_1	P_2	…	P_{n-1}	K_{n-1}	P_n

- K_i are the search-key values
- P_i are pointers to children (for non-leaf nodes) or pointers to records(for leaf nodes).
- The search-keys in a node are ordered
  - $K_1 < K_2 < K_3 < \ldots < K_{n-1}$

Insertion in B+ trees:

Consider a B+ tree of order 3. It means each node in tree can store maximum of 3 key values and have maximum of 4 children.

(i.e) If a node can store n key, it have n+1 pointers (or) child nodes

Insert the values 1, 4, 7, 10, 17, 21, 28 in B + tree.

Insert 1

	1					

Insert 4

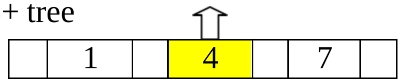
	1		4			

Insert 7

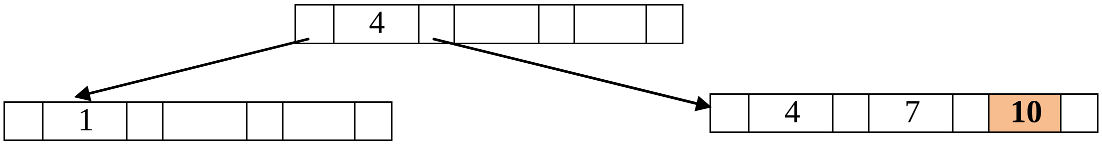
	1		4		7	

Insert 10

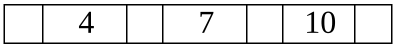
- When we try to insert 10, there is no space in node. Hence the middle value in node is found and it is moved one level upward
- Here 4 is moved one level upward and remaining values are attached according to the 5th property of B+ tree



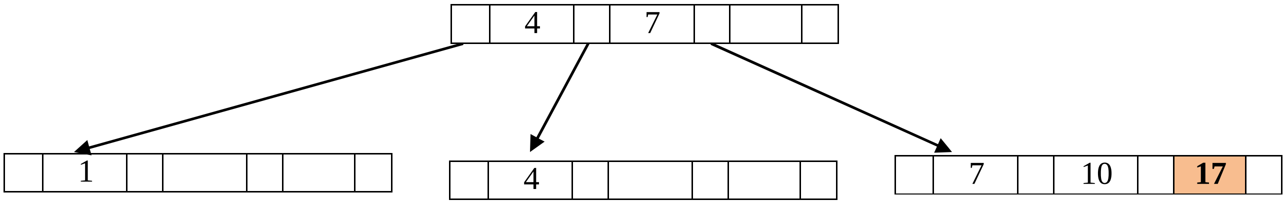
	1		4		7	



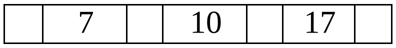
Insert 17

When we try to insert 17, there is no space in node 

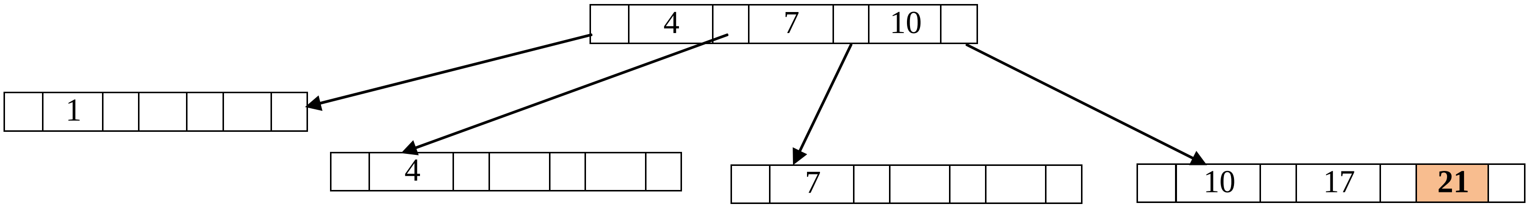
Hence middle value in the node 7 is moved one level upward and remaining elements are adjusted according to the 5th property of B+ tree



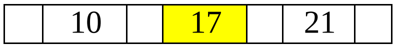
Insert 21

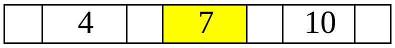
When we try to insert 21, there is no space in node 

Hence middle value in the node 10 is moved one level upward and remaining elements are adjusted according to the 5th property of B+ tree

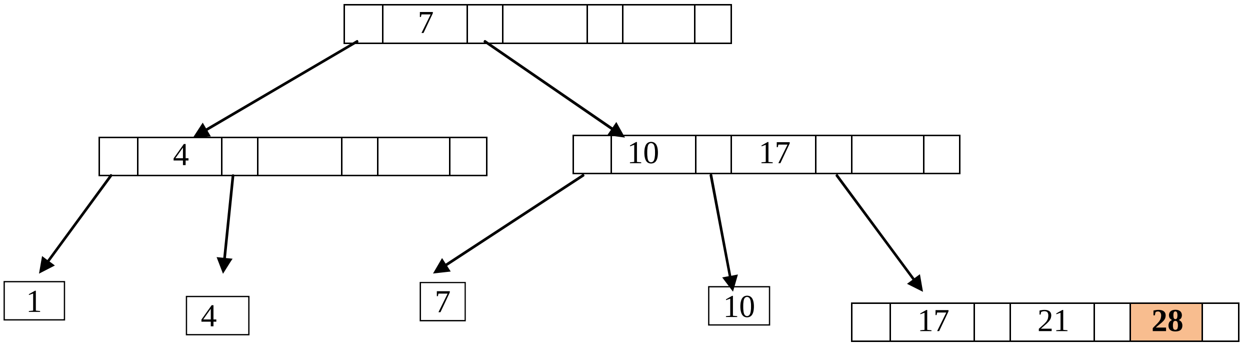


Insert 28

When we try to insert 28, there is no space in node 

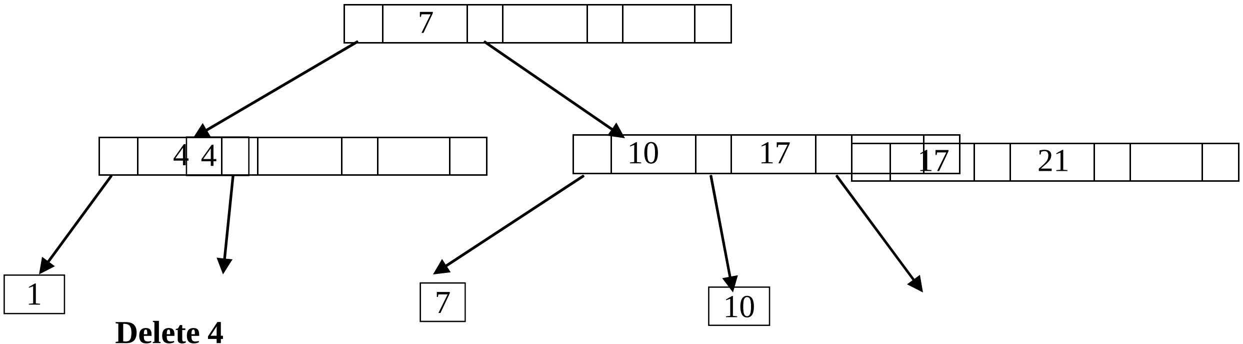
Hence middle value in the node 17 is moved one level upward. When 17 is moved one level upward, again there is no space in root node 

Hence, middle value in the node 7 is moved one level upward

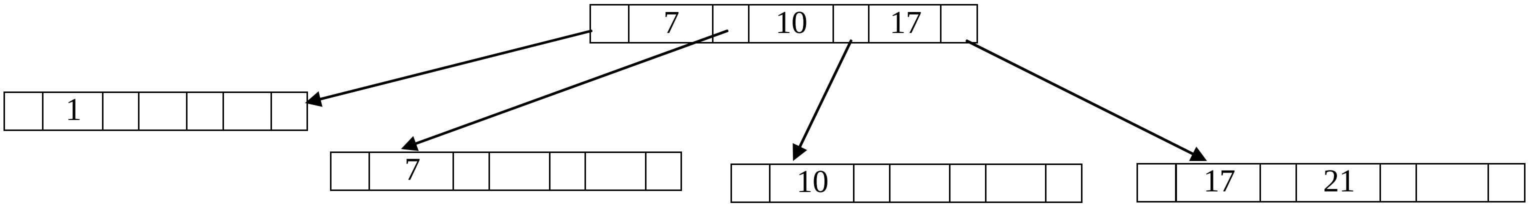


Deletion 28 and 4 :

Delete 28



Delete 4



Difference between B tree and B + tree:

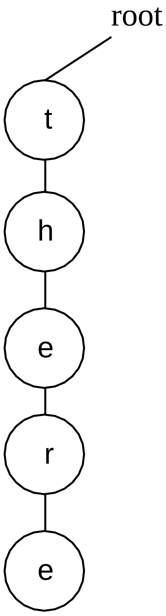
B tree	B + tree
More height compared to width	More width compared to height
Each node have some record. No need to traverse until leaf node	Root/internal node contain only pointer to leaf node. Leaf node only contain record

TRIE TREE INDEXING

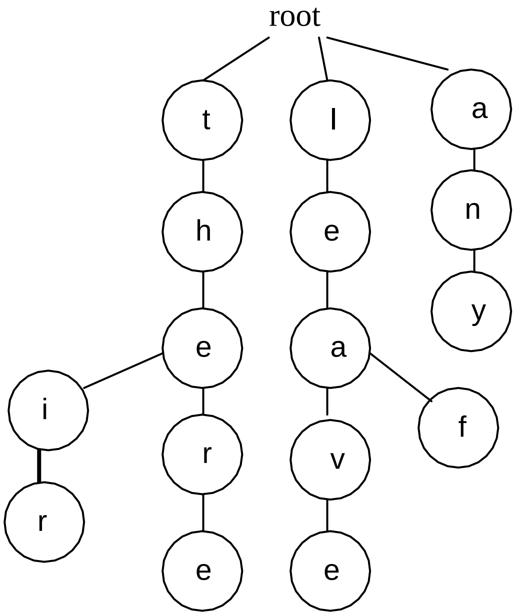
- Trie is an efficient information retrieval data structure, using which search complexities can be brought into optimal limit

Insertion in trie:

- In trie, every character of input key is inserted as a trie node.
- The children is an array of pointers to next level trie node.
- If K is the length of key, N is the number of keys in trie, Alphabet size is 26,
    - Memory requirement for trie is O(alphabet size * K * N)
- Initialize root with NULL
- Insert the key " **there**" in trie



- Last character in trie node act as leaf node
- Leaf node determines the end of the key
- In this above trie, there is only one child for each node
- If we insert a key, in which some of the characters of the key already exists in trie, then nodes for the new characters alone is constructed and they are joined with already existing characters.
- Insertion of keys **their, leave, leaf, bye** in trie



Searching in trie:

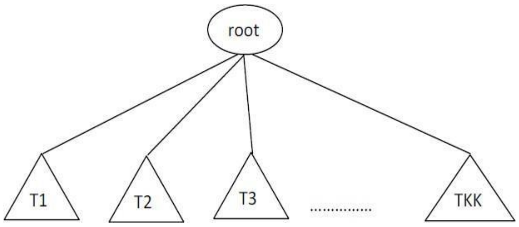
- When performing search in a trie, 1st character in search key is compared with the character in trie starting from the child of root. If it matches, searching continued with comparing 2nd character of search key with the that key in trie etc.
- Search may terminate due to the end of the string(i.e) search key exists in trie.
- Search may also terminate when the last character in the search key is not a leaf node in trie. (i.e) Search key doesn‟t exist
- **Searching** of key **"there".** We start from the child of root node. When 1st character „t‟ is found, the next character h is searched and so on. If all the characters of the search key is found and the last character in search key „e" is the leaf of the trie.
    - Hence the search key „ **there**" is present in trie.
- **Searching** of key **"thei".** In this all the characters of the search key are present in trie, but the last character in search key doesn‟t act as a leaf node intrie.
    - Hence, the search key **thei** is not present in trie.

UNIT – 3 TWO MARKS

1. Define Tree .Give an example.

- Tree is a non linear data structure, where there is no linear relation between the data items.
- It can be defined as finite set of more than one node.
- There is a special node designated as root node.
- The remaining nodes are partitioned into sub-trees(T1, T2,…,Tn) of a tree T.

Example: Directory structure hierarchy



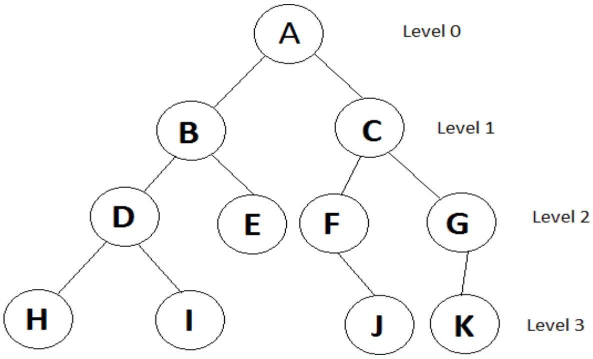


Figure: Tree

2. Define a path in a tree. Give example.

- The path in a tree is referred as the nodes in which the successive nodes are connected by the edge in a tree. **Example**: In the above tree,the path from A to I is A – B, B – D, D – I.
- A path from a node n_1 to n_k is defined as the sequence of nodes n_1, n_2…..n_k such that n_i is the parent of n_{i+1} for 1<i<k.

3. Define height of the node in a tree. Give example.

- The height of node n_i in a tree is the length of the longest path from n_i to a leaf
- In the above tree, Height of node B is 2. Height of node A is 3.

4. List the applications of trees.

- Decision making in games, Routing algorithms where the next path of packet is determined, Directory/ Folder traversal in a system, Auto-correct applications/ Spell checker, Syntax tree in compilers, Undo function in text editor
- Binary search trees, Expression trees, Threaded binary trees

5. Define terminal nodes in a tree

- A node that has no children is called as a terminal node. It is also referred as a leaf node. These nodes have degree has zero.

6. Define non-terminal nodes in a tree?

- All intermediate nodes that traverse the given tree from its root node to the terminal nodes are referred as non terminal nodes.

7. Define branch, siblings & ancestors?

- **Branch or edge** of a tree is called as the link or connection between two nodes.
- The nodes having the same parent are called **siblings**.
- The **ancestor** of a node is referred as all nodes along the path of root node to the node.

8. State the properties of tree?

- Any node can be the root of the tree.
- There is only one path exists between any node to every other node in a tree
- Every node, expect the root node has a unique parent.
- If the root is identified; then that tree is called as the rooted node, else tree is called as the free tree.

9. Define degree?

- The degree of a node is referred as the number of sub-trees of a particular node.
- Example: Degree of A, B, C, D are 2. Degree of F and G is 1 and Degree of H, I, E, J, K are 0.

10. Define height of the node and tree?

- The height of a node is the length of the longest downward path between the node and a leaf.
- The height of a tree is the length of the longest downward path between the root and a leaf.

11. Define binary tree

- A binary tree is a tree, which is, either empty or consists of a root node and two disjoint binary trees called the left sub-tree and right sub-tree.
- In a binary tree, ***no node can have more than two children***.

12. List the properties of binary tree.

- In any binary tree, the maximum number of nodes on level L is 2^L , where $L \geq 0$.
- The maximum number of nodes possible in a binary tree of height H is 2^H -1.
- The minimum number of nodes possible in a binary tree of height H is h.
- The height of a complete binary tree with n number of nodes is $\log_2(n+1)$.

13. What is meant by full binary tree?

- A **Full binary tree** is a binary tree in which all intermediate nodes have same degree as 2 and all leaves are at the same level.

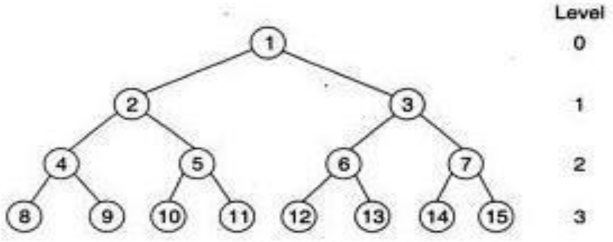


Fig. Full binary tree

14. What is meant by complete binary tree?

A binary tree is said to be a complete binary tree if all its levels, except possibly the last level, have the maximum number of possible nodes, and all the nodes in the last level appear as far left as possible

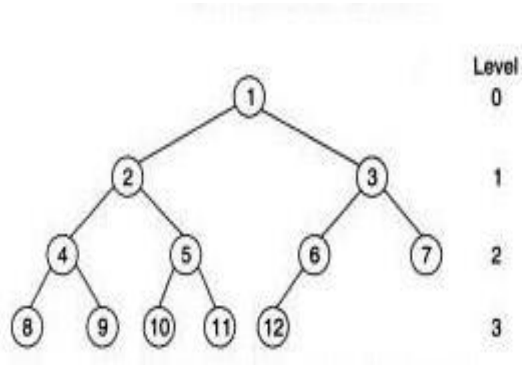


Fig. Complete binary tree

15. Define a left skewed binary tree?

- A left skewed binary tree is a tree, which has only left child nodes

16. Define a right skewed binary tree?

- A right skewed binary tree is a tree, which has only right child nodes

17. What are the different ways of representing a binary tree?

- Linear representation using arrays
- Linked representation using pointers

18. What is meant by binary tree traversal?

- Traversing a binary tree means moving through all the nodes in the binary tree visiting each node in the tree only once.

19. What are the difference binary tree traversal techniques?

- Inorder traversal
- Preorder traversal
- Postorder traversal

20. State the merits and demerits of linear representation of binary trees

Merits:

- Storage method is easy and can be easily implemented in arrays
- When the location of a parent /child node is known other one can be determined easily.

Demerits:

- Insertions and deletions in a node, taker an excessive amount of processing time due to data movement up and down the array.

21. State the merits and demerits of linked representation of a binary tree

Merits:

- Insertions and deletions in a node, involves no data movement except the re arrangement of pointers, hence less processing time.

Demerits:

- Given a node structure, it is difficult to determine its parent node.
- Memory spaces are wasted for storing null pointers for the nodes, which have one or no subtrees.

22. Define a binary search tree.

- A binary tree T is termed binary search tree or binary sorted tree, if each node N of T satisfies the following property:

"The value at N is greater than every value in the left sub-tree of N and is less than every value in the right sub-tree of N".

23. What do you mean by general trees?

- General tree is a tree with nodes having any number of children

24. Define Trie tree.

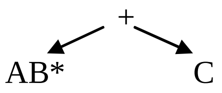
- A trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.
- It is an efficient information re***trie***val data structure. Using trie, search complexities can be brought to optimal limit (key length)

25. What is meant by Expression Tree?

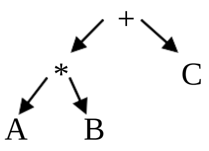
- An expression tree is a binary tree in which the operands are attached as leaf nodes and operators become the internal nodes.

26. Construct an expression tree for a following postfix expression AB*C+

<u>Step 1:</u>



<u>Step 2:</u>



27. List the tree traversal applications.

- Listing a directory in an hierarchal file system (preorder)
- Calculating the size of a directory (post order)

28. What is meant by full node in binary tree.

- Full node is a node which have both children as not NULL. (i.e) a node with two children
- In binary tree, the ***number of full node + 1 = Number of leaf nodes***

29. What are the applications of binary tree?

- Binary Search Tree
- Binary Space Partition
- Syntax tree
- Tries
- Heaps

30. Define B – Tree

- B - tree is a ***self-balancing tree*** data structure that *keeps data sorted* and allows searches, sequential access, insertions, and deletions in logarithmic time.
- The B-tree is a generalization of a binary search tree in that a node can have more than two children

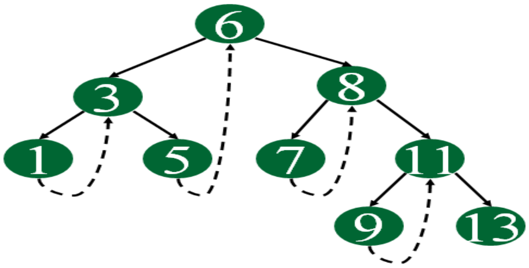
31. List the properties of B-trees.

B tree is a m-way search tree, such that

1. All leaf nodes are at same level
2. All internal nodes have m/2 to m children
3. Root have atleast two children unless it act as leaf node
4. The number of keys is one less than the number of children for non-leaf nodes and m/2 to m for leaf nodes

32. What is meant by Threaded binary tree?

- A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node
- The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion



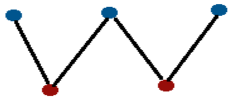
33. List the types of threaded binary tree.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

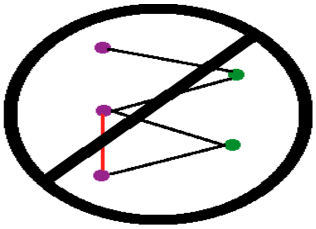
Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

34. Show that every tree is a bipartite graph.

- If a vertices of graph is divided into 2 disjoint subset, such that, edges should not connect vertices of same subset.
- Similarly, in tree, the child will be connected only to the parent and not to their siblings.



Bipartite



NOT Bipartite

35. Give the application/use of B- trees.

- B-tree is optimized for systems that read and write ***large blocks of data***.
- B-trees are a good example of a data structure for external memory. It is commonly ***used*** in ***databases and filesystems.***

36. Define B+ tree and its properties.

A B+ tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Leaf nodes store the actual record. Other nodes only have an index that is used to access that record
- Each internal node has $\lceil n/2 \rceil$ to n children.
- Each leaf node has between $\lceil (n-1)/2 \rceil$ and $n-1$ values
- Left child of parent is strictly less than the parent node and the right child of parent is greater than or equal to parent node.

37. Compare B tree and B + tree

B tree	**B + tree**
More height compared to width	More width compared to height
Each node have some record. No need to traverse until leaf node	Root/internal node contain only pointer to leaf node. Leaf node only contain record

38. Define AVL tree.

- AVL tree is a self-balancing binary search tree. In an AVL tree, the heights of the left and right sub-trees of any node differ by at most one
- Balance Factor(BF) = Height(Left Sub-tree) – Height(Right sub-tree)
- BF can be either -1, 0 or +1. If not tree should be balanced by making single or double rotations.

39. List out the advantages of trees.

- It is used in manipulation of hierarchical data
- Provides efficient searching of data
- Like linked list, there is no upper limit on number of elements to be stored in tree

UNIT – 4

GRAPH:

- A graph G consists of a set of Vertices (nodes) and a set of Edges (arcs).
- Graph can be represented as
  - **G(V,E)**. V is a finite and non-empty set of vertices.
  - E is a set of pair of vertices; these pairs are called as edges.
- V(G) = Set of vertices of graph G and E(G) = Set of edges of graph G
- An Edge e= (v, w) is a pair of vertices v and w, and to be incident with v and w.
- A graph can be pictorially represented as follows,

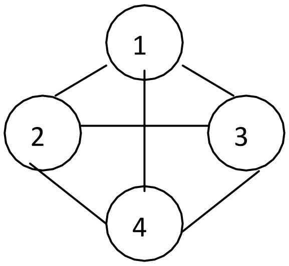


Fig. ***Graph G***

✓ We have numbered the vertices of the graph as 1,2,3,4

✓ Therefore, V(G)=(1,2,3,4) and E(G) = {(1,2),(1,3),(1,4),(2,3),(2,4), (3,4)}

BASIC TERMINOLOGIES OF GRAPH

Undirected Graph (or) Unqualified graph:

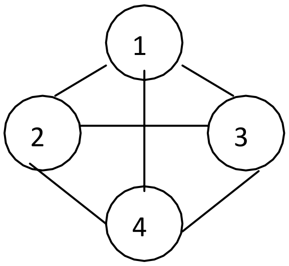
- Undirected graph is a graph in which, the pair of vertices representing the edges is unordered.
- Edges in undirected graph doesn"t specify the direction

Directed graph (or) Digraph:

- Directed graph is a graph in which, the pair of vertices representing the edges is ordered.
- Edges in directed graph specify the direction

Path in a graph:

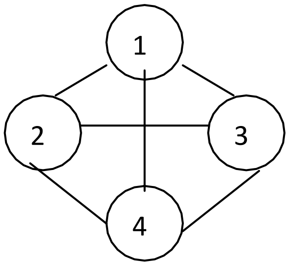
- Path between vertices u,v is a sequence of edges that connects u and v



- Path between 1 to 4 are {1 – 2 –4, 1–4, 1–3–4, 1–2–3–4, 1–3–2–4}

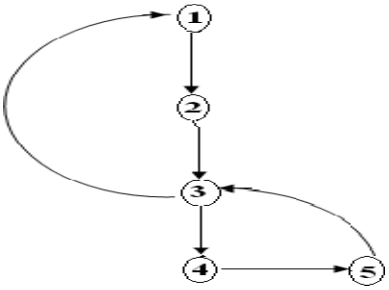
Connected graph:

- In Undirected graph, If there is a path from any vertex to every other vertex in a graph, then the graph is called connected graph.



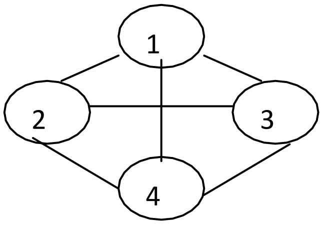
Strongly Connected graph:

- In Directed graph, If there is a path from any vertex to every other vertex in a graph, then the graph is called strongly connected graph.



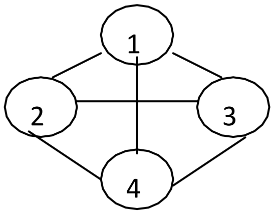
Complete Graph

- A Graph is a complete, if there is an edge between every pair of vertices.
- It has exactly n(n-1)/2 number of edges.



Sub Graph

A sub-graph of G is a graph G„ such that V(G‟) ⊆ V(G) and E(G „) ⊆E(G).



(a)

Graph G

Some of the Subgraphs of G

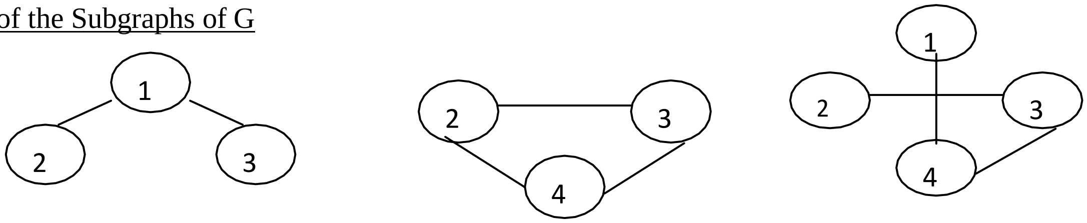
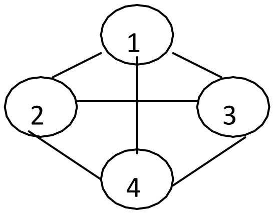


Fig. Sub graphs of G

Adjacent Vertices:

A vertex v1 is said to be a adjacent vertex of v2, if there exist an edge (v1,v2) or (v2,v1).

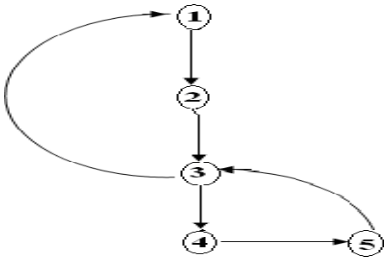


(a)

Graph G

Adjacent(1) = {2,3,4}, Adjacent(2) = {1,3,4}, Adjacent(3) = {1,2,4}, Adjacent(4) = {1,2,3}

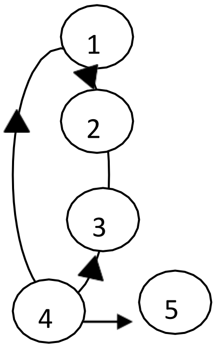
Length of the graph:



The length of the graph is the number of edges in it.

Weakly Connected Graph

If there does not exist a directed path from one vertex to another vertex then it is said to be a weakly connected graph.



Cycle

A cycle is a path in which the first and the last vertices are the same.

Degree

- The number of edges incident on a vertex determines its degree. There are two types of degrees for directed graph **In-degree** and **out-degree**.

➢ **In-Degree** of the vertex V is the number of edges for which vertex V is a head. (or) Number of edges whose direction is towards the vertex

➢ **Out-Degree** of vertex V is the number of edges for which vertex is a tail. (or) Number of edges whose direction is outside that vertex

A GRAPH IS SAID TO BE A TREE, IF IT SATISFIES THE TWO PROPERTIES:

a. It is connected

b. There are no cycles in the graph.

GRAPH REPRESENTATION

A graph can be represented by some of the following three methods,

1. Set representation
2. Adjacency matrix.
3. Adjacency list.

1. SET REPRESENTATION

This is one of the straightforward method of representing a graph. With this method, two sets are maintained:

(1) V, the set of vertices,

(2) E, the set of edges

But if the graph is weighted, the set E is the ordered collection of three tuples, that is, E=(cost of edge, source vertex, destination vertex).

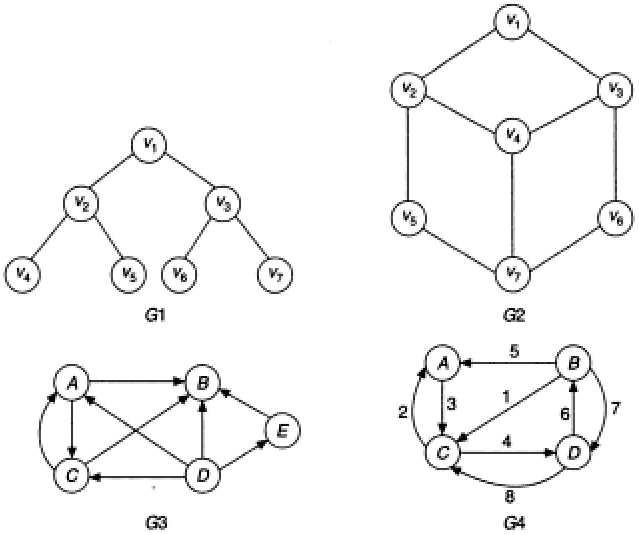
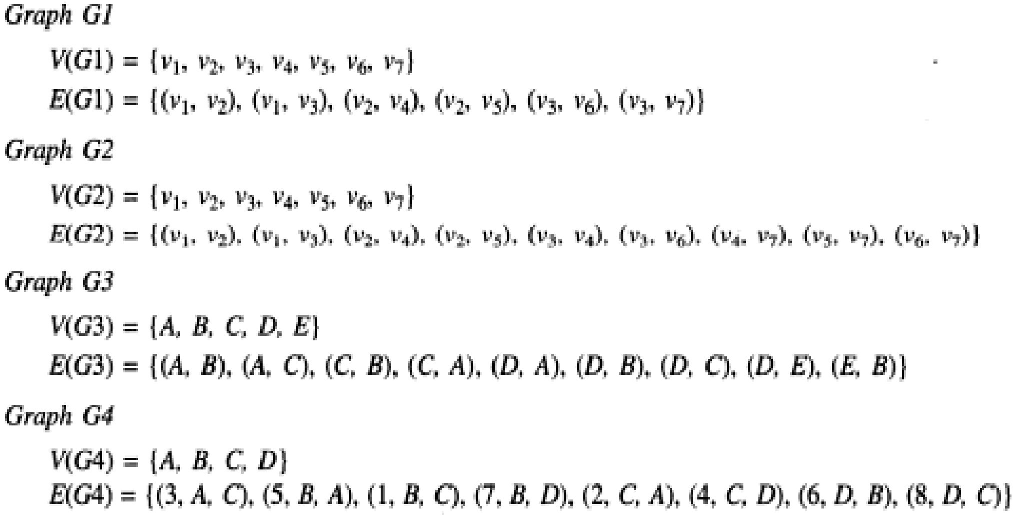


Fig. Types of Graphs

Above Graphs can be represented as follows,



Graph G1

$V(G1) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E(G1) = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_6), (v_3, v_7)\}$

Graph G2

$V(G2) = \{v_1, v_2, v_3, v_4, v_5, v_6, v_7\}$

$E(G2) = \{(v_1, v_2), (v_1, v_3), (v_2, v_4), (v_2, v_5), (v_3, v_4), (v_3, v_6), (v_4, v_7), (v_5, v_7), (v_6, v_7)\}$

Graph G3

$V(G3) = \{A, B, C, D, E\}$

$E(G3) = \{(A, B), (A, C), (C, B), (C, A), (D, A), (D, B), (D, C), (D, E), (E, B)\}$

Graph G4

$V(G4) = \{A, B, C, D\}$

$E(G4) = \{(3, A, C), (5, B, A), (1, B, C), (7, B, D), (2, C, A), (4, C, D), (6, D, B), (8, D, C)\}$

Although, it is a straight forward representation and the most efficient one from the memory point of view, this method of representation is not useful so far as the manipulation of graph is concerned.

2. ADJACENCY MATRIX REPRESENTATION

The adjacency matrix A for a graph G(V,E) with n vertices, is an n* n matrix of bits ,such that

A_{ij} = 1 , if there is an edge from vi to vj and

A_{ij} = 0, if there is no such edge.

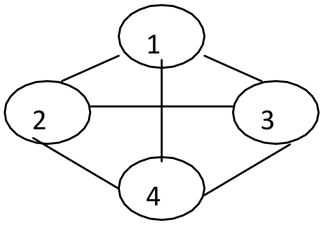


Fig. Graph (G)

The adjacency matrix for the graph G is,

$$A= \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

- The space required to represent a graph using its adjacency matrix is n* n bits.
- From the adjacency matrix, one may readily determine if there an edge connecting any two vertices i and j.

For directed graph, A_{ij} = 1 , if there is an directed edge from vi to vj and

A_{ij} = 0, if there is no directed edge from vi to vj.

The adjacency matrix is a simple way to represent a graph, but it has 2 disadvantages,

- It takes O(n*n) space to represent a graph with n vertices ; even for sparse graphs and
- It takes O(n*n) times to solve most of the graph problems.

iii) ADJACENCY LIST OR LINKED REPRESENTATION

- The representation of the n rows of the adjacency matrix are represented as n linked lists. There is one list for each vertex in G. The nodes in list I represent the vertices that are adjacent from vertex I. Each node has at least 2 fields: VERTEX which contain vertex name and LINK to next vertex in list.

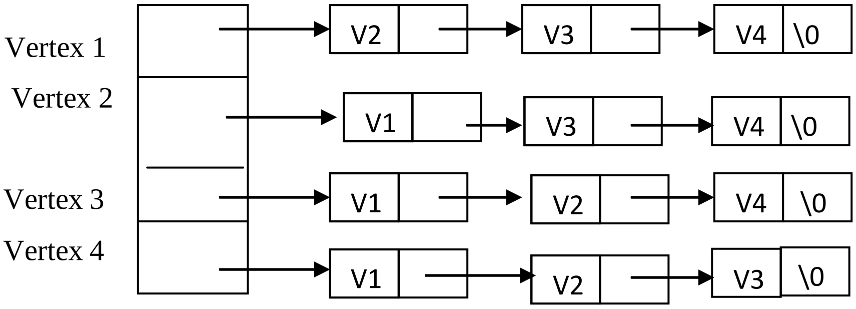


Fig. Linked list representation of Graph

- The total number of edges in G can be determined in time O(n+e).

Comparison among Various Representations

- The adjacency matrix representation has the disadvantage that it always requires an n×n matrix with *n* vertices, regardless of the number of edges.
- The set representation of graphs is very concise but manipulation with this representation has a lot of difficulties.
- Insertion, deletion, searching, merging operations in a graph can be easily done using linked list representation.
- But , when we consider overall performance, the matrix representation is more powerful than all.

GRAPH TRAVERSAL

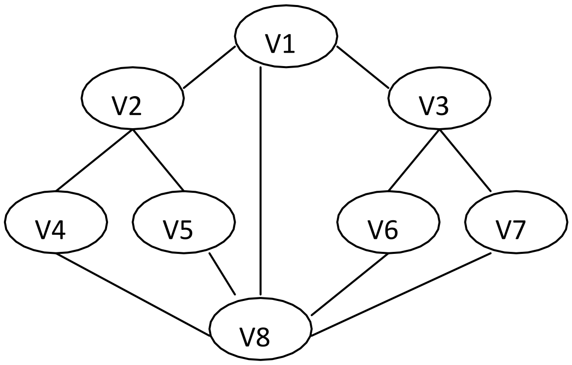
Given an undirected graph G(V.E) and a vertex v in V(G) we are interested in visiting all vertices in G that are reachable from v (that is all vertices connected to v). We have two ways to do the traversal. They are

- **Depth First Search**
- **Breadth First Search.**

DEPTH FIRST SEARCH

- In DFS we pick on one of the adjacent vertices; visit all of its adjacent vertices and back track to visit the unvisited adjacent vertices.
- In graphs, we do not have any start vertex or any special vertex Therefore the traversal may start from any arbitrary vertex.

Let us see an example, consider the following graph.



Let us start with **V_1**,

1. Its adjacent vertices are V2, V8, and V3. Let us visit **V2**.
2. V2‟s adjacent vertices are V1, V4, V5. Since V1 is already visited, Let us visit **V4**.
3. V4‟s adjacent vertices are V2, V8. Since, V2 is already visited, Let us visit **V8**.
4. V8‟s adjacent vertices are V4, V5, V1, V6, V7. Since V4 and V1 are visited. Let us visit **V5**.
5. V5‟s adjacent vertices are V2, V8.
6. Since Both vertices V2 and V8 are already visited, backtracking is performed.
7. We had V6 and V7 unvisited in the list of V8, visit **V6**.
8. V6‟s adjacent vertices are V8 and V3. Since V8 already visited, visit **V3**.
9. V3‟s adjacent vertices are V1, V7. Since V1 is already visited, visit **V7**.
10. V7‟s adjacent vertices are already visited, we back track and find that we have visited all the vertices of G.

Therefore the sequence of traversal is

V1, V2, V4, V8, V5, V6, V3, V7.

This is not a unique or the only sequence possible using this traversal method.

We may ***implement*** the ***Depth First search*** by using a **stack** and also by using **recursion**

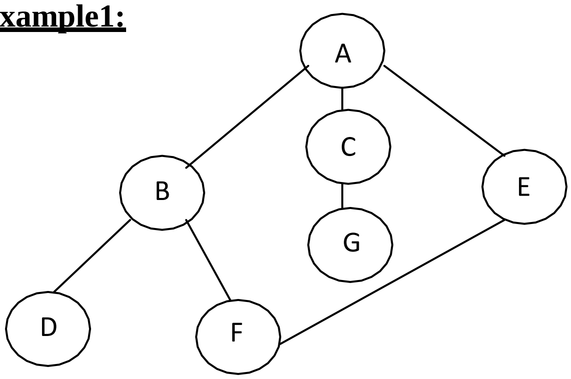
DFS is best described recursively as

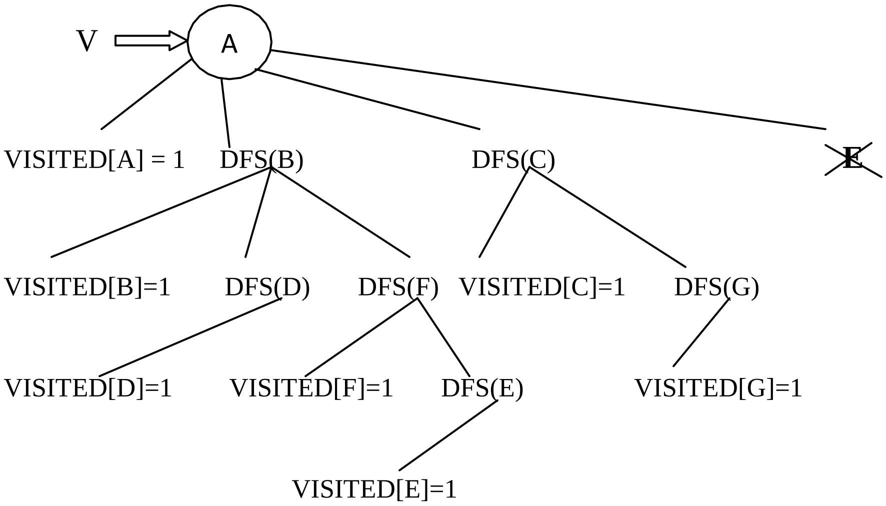
Algorithm:

DFS(v)

```
//Given an undirected graph G(V, E) with n vertices and an VISITED array initially set to zero
//This algorithm visits all vertices reachable from v .
    VISITED[v] = 1
    for each vertex w adjacent to v do
        if VISITED [w] =0 then
            call DFS (w)
        end if
    end for
```

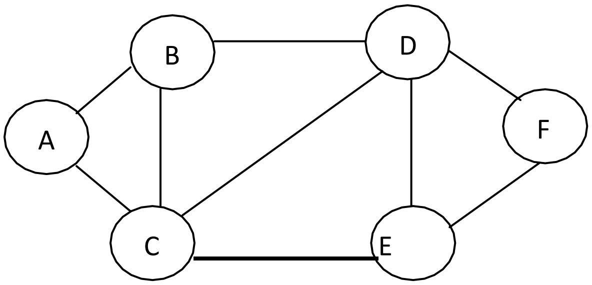
Example1:

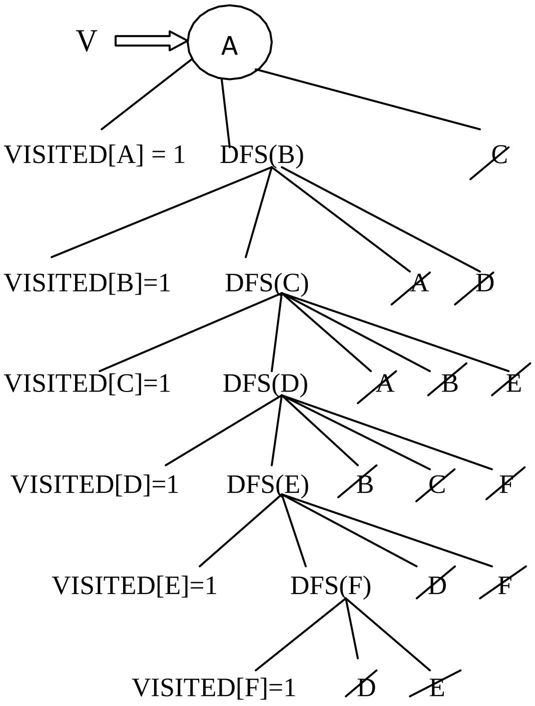




ORDER OF DFS TRAVERSAL: ABDFECG

Example 2:

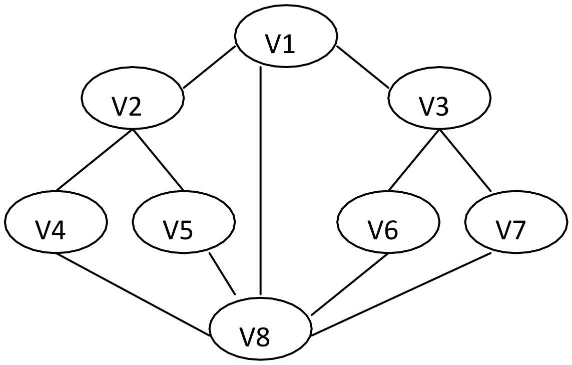




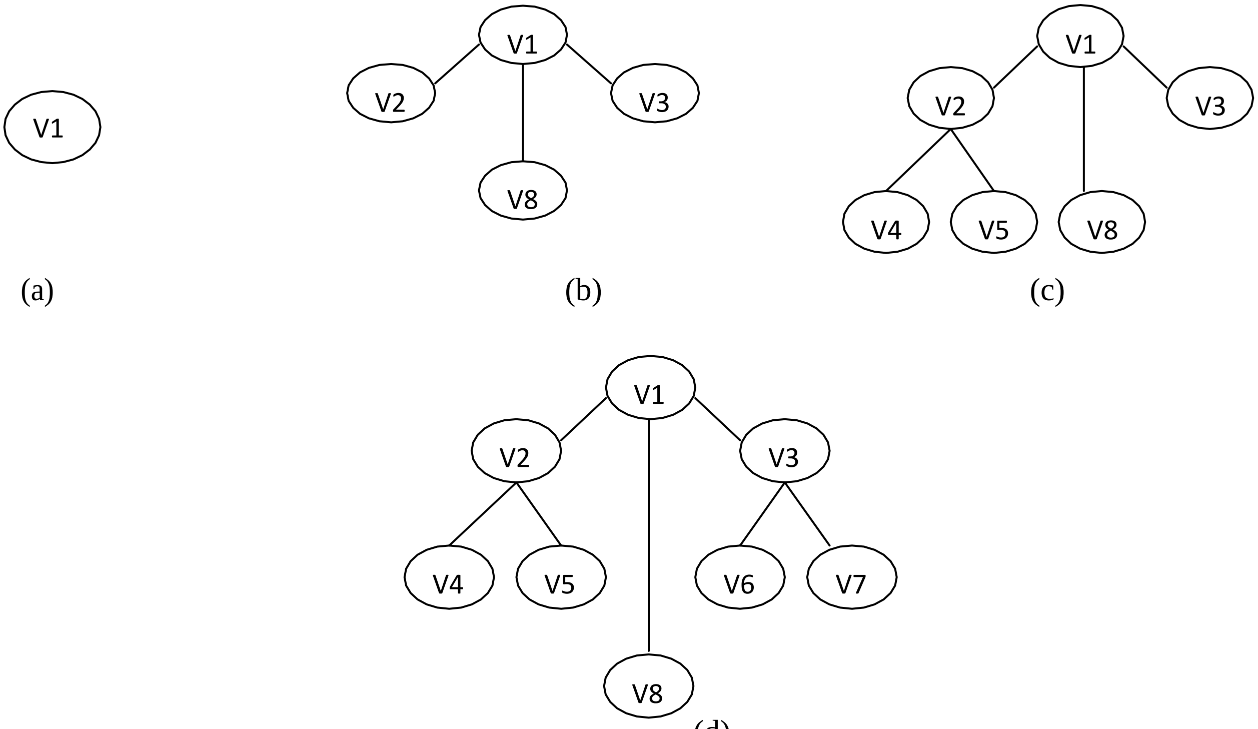
ORDER OF DFS TRAVERSAL: A,B,C,D,E,F

BREADTH FIRST SEARCH

Consider the following graph.



- In BFS, we first visit all the adjacent vertices of the start vertex and then visit all the unvisited vertices adjacent to these and so on.
- Let us consider the same example, given in figure. We start say, with V_1. Its adjacent vertices are V_2, V_8, V3.
- We visit all one by one. We pick on one of these, say V_2. The unvisited adjacent vertices to V_2 are V_4, V_5 . we visitboth.
- We go back to the remaining visited vertices of V_1 and pick on one of this, say V_3. The unvisited adjacent vertices to V_3 are V_6,V_7. We visit both. Thus all the vertices are visited.



(a) (b) (c)

(d)

Fig. Breadth First Search

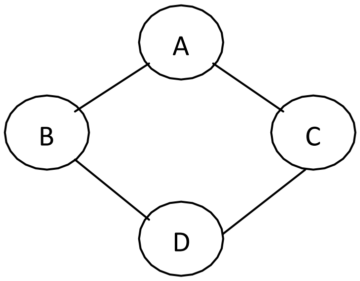
- Thus the sequence so generated is V_1,V_2, V_8, V_3,V_4, V_5,V_6, V_7. Here we need a queue instead of a stack to implement it.
- We add unvisited vertices adjacent to the one just visited at the rear and read at from to find the next vertex to visit.

ALGORITHM:

BFS(v)

```
   //BFS of G is carried out beginning at vertex v. All vertices are marked as visited[i]=1
   Visited[v]=1
   Enqueue vertex v to queue
   While(queue is not empty)
          Dequeue()
     For all vertices w adjacent to v
        If(visited(w)==0)
              Enqueue(w)
              Visited[w]=1
        End if
      End for
 End while
```

Example:



0	0	0	0
Visited[A]	Visited[B]	Visited[C]	Visited[D]

Consider, BFS started at vertex A.

BFS(A)

Visited[A] = 1

1	0	0	0
Visited[A]	Visited[B]	Visited[C]	Visited[D]

Enqueue vertex A in Queue

Queue	A			

Q!= empty => Perform dequeue => ***A is dequeued***

Adjacent(A) = { B, C} and both B and C are not visited.

Enqueue B and mark it as visited and then enqueue C and mark it as visited

Queue	B	C		

1	1	1	0
Visited[A]	Visited[B]	Visited[C]	Visited[D]

Q!= empty=> perform dequeue => ***B is dequeued***

Adjacent(B) = {A, D}

Here A is already visited. Hence ***D is enqueued and it is marked as visited***

Queue

C	D		

1	1	1	1
Visited[A]	Visited[B]	Visited[C]	Visited[D]

Q!=empty => Perform dequeue => ***C dequeued***

Adjacent(C) = {A,D}. Here both A and D are already visited

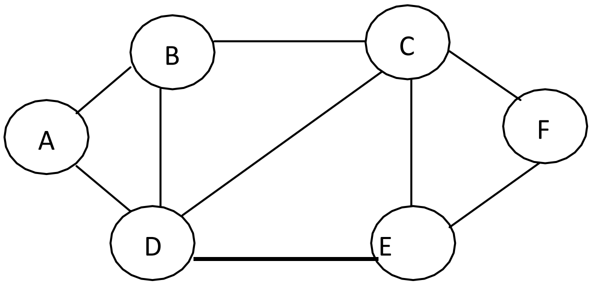
Q!=empty=> ***D dequeued***

Adjacent(D) = { B, C}. Here both B and C are already visited

Q!=empty => False

ORDER OF BFS TRAVERSAL : ABCD

EXAMPLE 2:



0	0	0	0	0	0
Visited[A]	Visited[B]	Visited[C]	Visited[D]	Visited[E]	Visited[F]

Consider, BFS started at vertex A.
BFS(A)
Visited[A] = 1

1	0	0	0	0	0
Visited[A]	Visited[B]	Visited[C]	Visited[D]	Visited[E]	Visited[F]

Enqueue vertex A in Queue

Queue

A			

Q!= empty => Perform dequeue => ***A is dequeued***

Adjacent(A) = { B, D} and both B and D are not visited.

Enqueue B and mark it as visited and then enqueue D and mark it as visited

Queue	B	D		

1	1	0	1	0	0
Visited[A]	Visited[B]	Visited[C]	Visited[D]	Visited[E]	Visited[F]

Q!= empty=> perform dequeue => ***B is dequeued***

Adjacent(B) = {A, C, D}

Here A and D are already visited. C is not visited. Hence ***C is enqueued and marked as visited***

Queue	D	C		

1	1	1	1	0	0
Visited[A]	Visited[B]	Visited[C]	Visited[D]	Visited[E]	Visited[F]

Q!=empty => Perform dequeue => ***D dequeued***

Adjacent(D) = {A,B,C,E}. Here both A, B and C are already visited. E is not visited. Hence ***E is enqueued and marked as visited***

Queue	C	E		

1	1	1	1	1	0
Visited[A]	Visited[B]	Visited[C]	Visited[D]	Visited[E]	Visited[F]

Q!=empty=> ***C dequeued.***

Adjacent(C) = { B, D, E, F}. Here B, D and E are already visited. F is not visited. Hence ***F is enqueued and marked as visited***

Queue	E	F		

1	1	1	1	1	1
Visited[A]	Visited[B]	Visited[C]	Visited[D]	Visited[E]	Visited[F]

Q!=empty=> ***E dequeued.***

Adjacent(E) = {C,D,F}. Here all vertices C, D and F are already visited.

Q!=empty => ***F dequeued***

Adjacent(F) = { C,E}. Here C and E are already visited.

ORDER OF BFS TRAVERSAL: ABDCEF

MINIMUM SPANNING TREE

Spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected

(Or)

Spanning tree: The property of spanning tree is that all the vertex of Graph G is visited exactly once.

Minimum Spanning tree: The property of minimum spanning tree is that all the vertex of Graph G is visited exactly once with minimum cost.

(or)

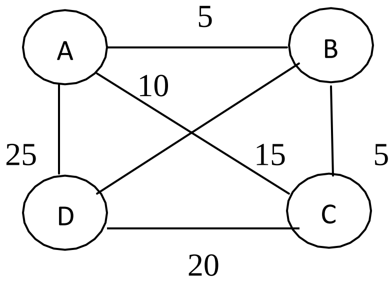
Minimum spanning tree is a spanning tree that has minimum weight/cost than all other spanning trees of the same graph.

PRIM'S ALGORITHM TO FIND THE MINIMUM SPANNING TREE OF A GRAPH

Algorithm:

1. Construct a cost matrix for a given graph G.
   // Cost matrix contains a cost of each vertex to every other vertex in graph G
2. Start at any arbitrary vertex. Add that vertex to the tree T and mark it as visited
3. The edge (u,v) can be added to the tree T, if
   i) It is an edge with minimum cost
   ii) It should not create a cycle in T
   iii) The newly added vertex doesn‟t present in T already. If so, that vertex v is marked as visited.

Example:



Cost matrix:

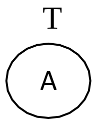
	A	B	C	D
A	**-**	5	10	25
B	5	**-**	5	15
C	10	5	**-**	20
D	25	15	20	**-**

Initially spanning tree T is Null

Consider, the arbitrary vertex chosen is A

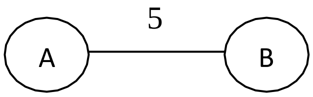
Hence A is added to T and it is marked as visited

Step 1:



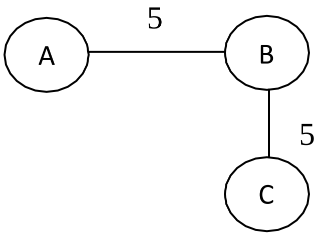
Step 2:

The edge A,B is an edge with minimum cost in row A. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



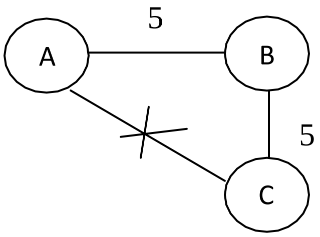
Step 3:

The edge B,C is an edge with next minimum cost in row A and B. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T.



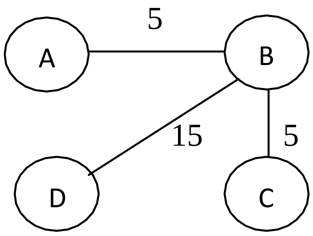
Step 4:

The edge A,C is an edge with next minimum cost in row A, B and C. Adding an edge between A and C creates a loop in spanning tree T. Hence it is ignored.



Step 5:

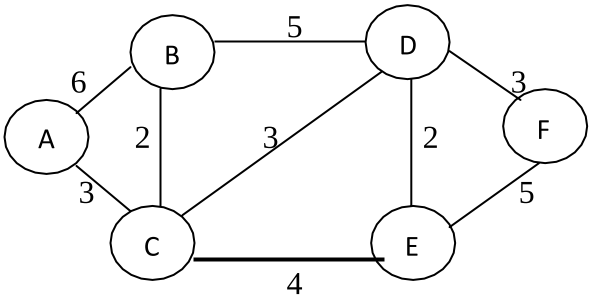
The edge B,D is an edge with next minimum cost in row A, B and C. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T.



In minimum spanning tree T, All vertices are visited exactly once.

Minimum cost of visiting all vertices of given graph G exactly once is 25.

Example 2:



Cost matrix:

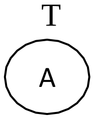
	A	**B**	**C**	**D**	**E**	**F**
A	**-**	6	3	∞	∞	∞
B	6	**-**	2	5	∞	∞
C	3	2	**-**	3	4	∞
D	∞	5	3	**-**	2	3
E	∞	∞	4	2	**-**	5
F	∞	∞	∞	3	5	**-**

Initially spanning tree T is Null

Consider, the arbitrary vertex chosen is A

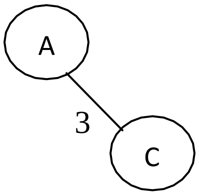
Hence A is added to T and it is marked as visited

Step 1:



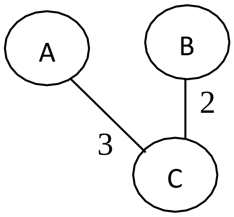
Step 2:

The edge A,C is an edge with minimum cost in row A. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



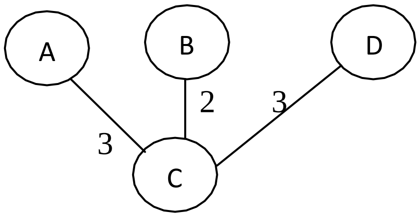
Step 3:

The edge B,C is an edge with next minimum cost in row A and B. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T.



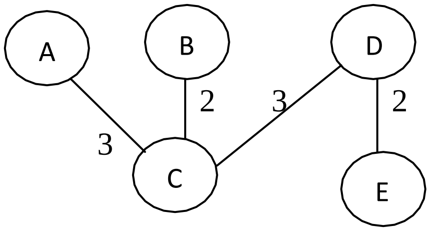
Step 4:

The edge C,D is an edge with next minimum cost in row A, B and C. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T.



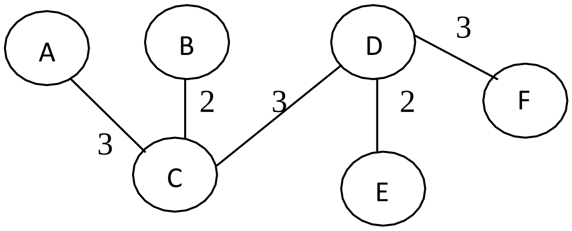
Step 5:

The edge D,E is an edge with next minimum cost in row A, B, C and D. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T.



Step 6:

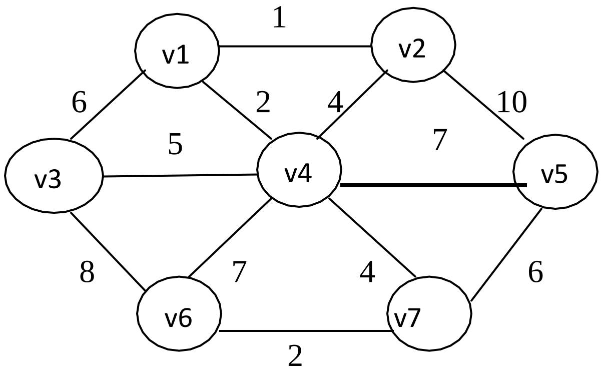
The edge D,F is an edge with next minimum cost in row A, B, C, D and E. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T.



In above minimum spanning tree T, All vertices are visited exactly once.

Minimum cost of visiting all vertices of given graph G exactly once is 13.

Example 3:



Cost matrix:

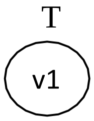
	v1	v2	v3	v4	v5	v6	v7
v1	-	1	6	2	∞	∞	∞
v2	1	-	∞	4	10	∞	∞
v3	6	∞	-	5	8	∞	∞
v4	2	4	5	-	7	7	4
v5	∞	10	∞	7	-	∞	6
v6	∞	∞	8	7	∞	-	2
v7	∞	∞	∞	4	6	2	-

Initially spanning tree T is Null

Consider, the arbitrary vertex chosen is v1

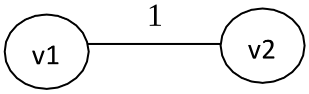
Hence v1 is added to T and it is marked as visited

Step 1:



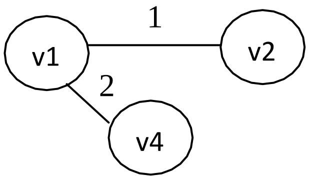
Step 2:

The edge v1,v2 is an edge with minimum cost in row v1. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



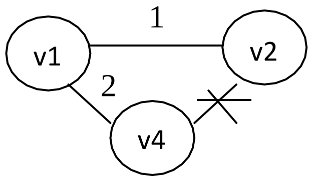
Step 3:

The edge v1,v4 is an edge with minimum cost in row v1 and v2. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



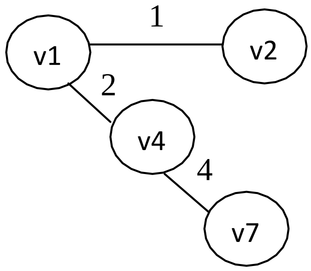
Step 4:

The edge v2,v4 is an edge with minimum cost in row v1, v2 and v4. **Adding edge v2,v4 form a cycle.** Hence this edge is ignored.



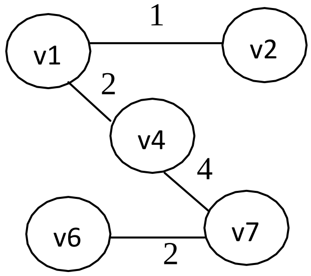
Step 5:

The edge v4,v7 is an edge with minimum cost in row v1, v2 and v4. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



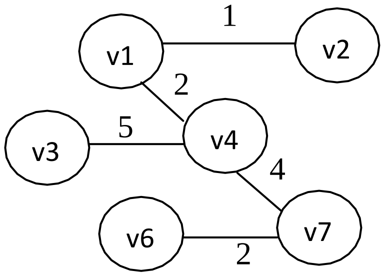
Step 6:

The edge v7,v6 is an edge with minimum cost in row v1, v2, v4 and v7. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



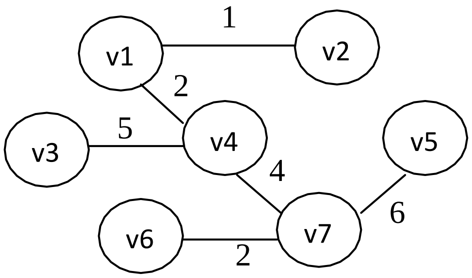
Step 7:

The edge v4,v3 is an edge with minimum cost in row v1, v2, v4, v6 and v7. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



Step 8:

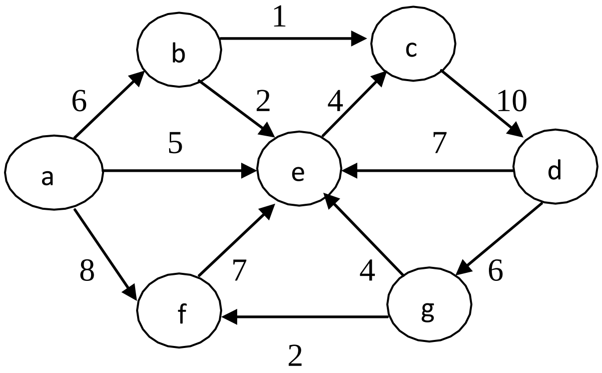
The edge v7,v5 is an edge with minimum cost in row v1, v2, v3, v4, v6 and v7. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



In above minimum spanning tree T, All vertices are visited exactly once.

Minimum cost of visiting all vertices of given graph G exactly once is 20.

Example 4:



Cost matrix:

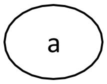
	a	b	c	d	e	f	g
a	-	6	∞	∞	5	8	∞
b	∞	-	1	∞	2	∞	∞
c	∞	∞	-	10	∞	∞	∞
d	∞	∞	∞	-	7	∞	6
e	∞	∞	4	∞	-	∞	∞
f	∞	∞	∞	∞	7	-	∞
g	∞	∞	∞	∞	4	2	-

Step 1:

Initially spanning tree T is Null

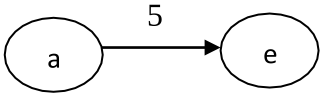
Consider, the arbitrary vertex chosen is **a**

Hence **a** is added to T and it is marked as visited



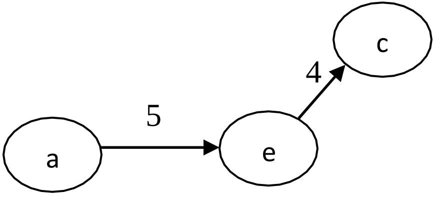
Step 2:

The edge **a,e** is an edge with minimum cost in row ***a***. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



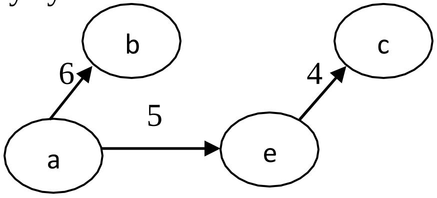
Step 3:

The edge **e,c** is an edge with minimum cost in row ***a and e***. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



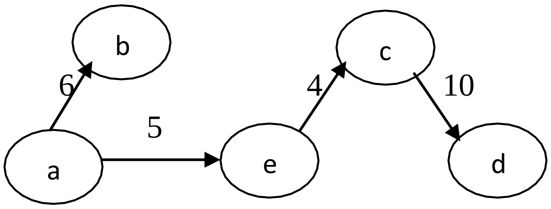
Step 4:

The edge **a,b** is an edge with minimum cost in row ***a, c and e***. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



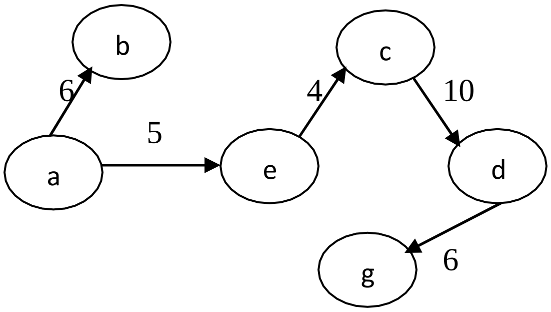
Step 5:

The edges **b,c** and **b,e** cannot be added to T, since each vertex can be visited exactly once. The edge **c,d** is an edge with minimum cost in row ***a, b, c and e***. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



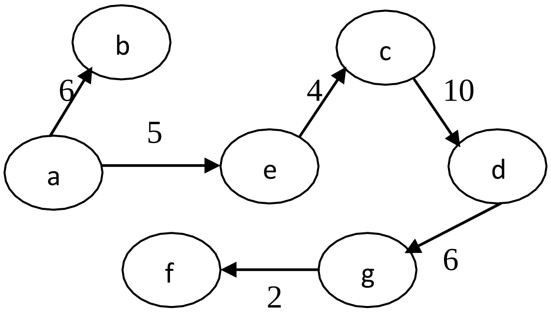
Step 6:

The edge **d,g** is an edge with minimum cost in row ***a, b, c, d and e***. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



Step 7:

The edge **g,f** is an edge with minimum cost in row ***a, b, c, d, e and g***. Adding of that edge doesn‟t form any cycle. Hence it is added to the tree T and it is marked.



In above minimum spanning tree T, All vertices are visited exactly once.

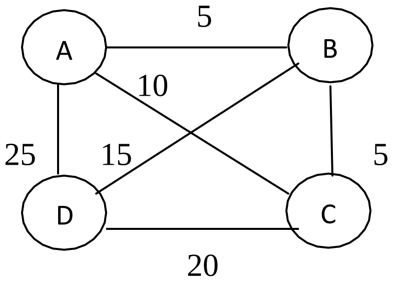
Minimum cost of visiting all vertices of given graph G exactly once is 33.

KRUSKAL ALGORITHM TO FIND MINIMUM SPANNING TREE OF GRAPH

Algorithm:

```
T = Null. // T is a minimum spanning tree T
E is an edge set which contains all edges in G from lowest cost to highest cost
While((Number of edges in T < n – 1 ) && ( E ! = empty))
    Choose an edge (u, v) from E with lowest cost
    Delete (u,v) from E
    If (u, v) doesn‟t creates a cycle in T
        Add that edge edge (u,v) to T
    Else
        Discard that edge (u,v)
    End if
End while
```

Example:

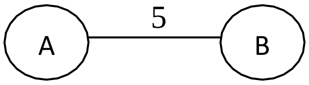


Edge set:

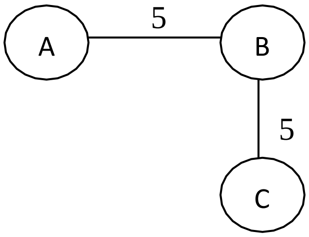
Edges	AB	BC	AC	BD	DC	AD
Cost	5	5	10	15	20	25

Step 1:

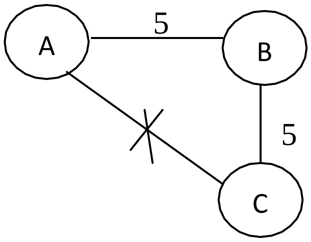
Initially T is null. Number of vertices n = 4. Hence n-1 = 3
Since, Number of edges in T is 0 < 3 and E ! = empty
Edge AB is chosen from Edge set E and it is deleted.
Since adding of edge AB doesn‟t form cycle, it is added to T



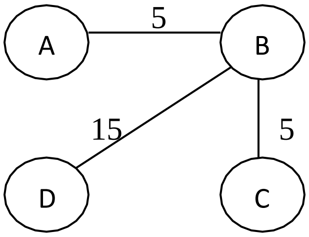
Now, Number of edges in T is 1 < 3 and E ! = empty
Edge BC is chosen from Edge set E and it is deleted.
Since adding of edge BC doesn‟t form cycle, it is added to T



Now, Number of edges in T is 2 < 3 and E ! = empty
Edge AC is chosen from Edge set E and it is deleted.
Since adding of edge AC forms cycle, it is ignored



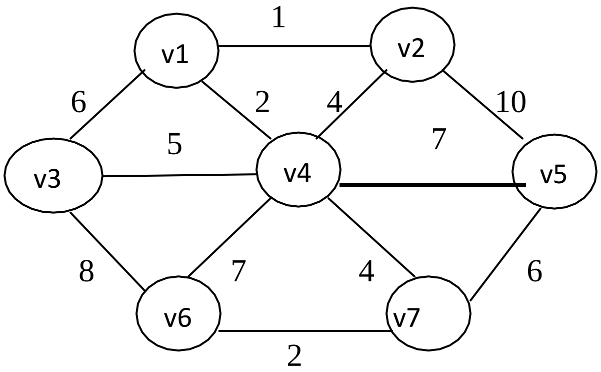
Edge BD is chosen from Edge set E and it is deleted.
Since adding of edge BD doesn‟t forms cycle, it is added to T



In above minimum spanning tree T, All vertices are visited exactly once.

Minimum cost of visiting all vertices of given graph G exactly once is 25.

Example 2:



Edge set:

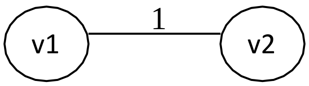
Edges	v1,v2	v1,v4	v6,v7	v2,v4	v4,v7	v3,v4	v1,v3	v5,v7	v4,v5	v4,v6	v3,v6	v2,v5
Cost	1	2	2	4	4	5	6	6	7	7	8	10

Step 1:

Initially T is null. Number of vertices n =7. Hence n-1 = 6

Edge v1,v2 is chosen from Edge set E and it is deleted.

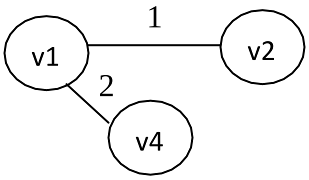
Since adding of edge v1v2 doesn‟t form cycle, it is added to T



Step 2:

Edge v1,v4 is chosen from Edge set E and it is deleted.

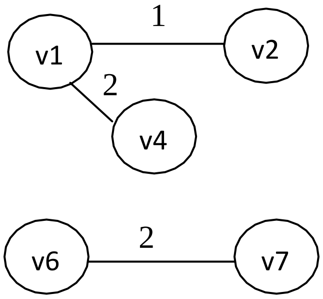
Since adding of edge v1v4 doesn‟t form cycle, it is added to T



Step 3:

The edge v6,v7 is chosen from Edge set E and it is deleted.

Since adding of edge v6,v7 doesn‟t form cycle, it is added to T



Step 4:

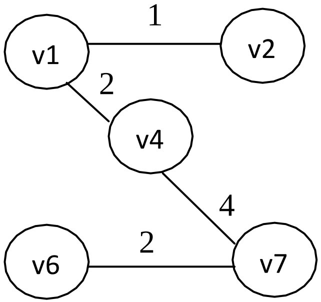
The edge v2,v4 is chosen from Edge set E and it is deleted.

Since adding of edge v2,v4 form cycle, it is ignored

Step 5:

The edge v4,v7 is chosen from Edge set E and it is deleted.

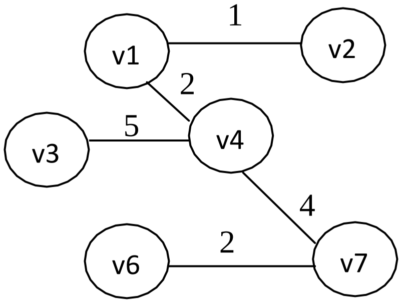
Since adding of edge v4,v7 doesn‟t form cycle, it is added to T



Step 6:

The edge v3,v4 is chosen from Edge set E and it is deleted.

Since adding of edge v3,v4 doesn‟t form cycle, it is added to T



Step 7:

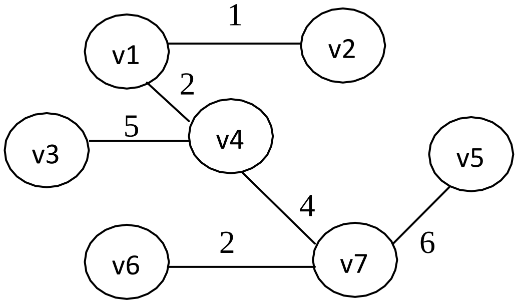
The edge v1,v3 is chosen from Edge set E and it is deleted.

Since adding of edge v1,v3 form cycle, it is ignored

Step 8:

The edge v5,v7 is chosen from Edge set E and it is deleted.

Since adding of edge v5,v7 doesn‟t form cycle, it is added to T



In above minimum spanning tree T, All vertices are visited exactly once.

Minimum cost of visiting all vertices of given graph G exactly once is 20.

SINGLE SOURCE SHORTEST PATH ALGORITHM

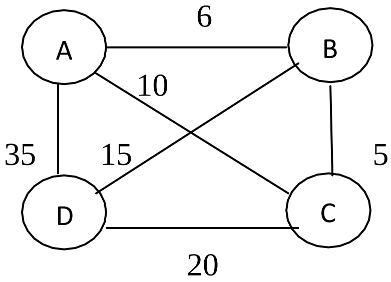
Dijkstra algorithm:

Given a single source vertex, dijkstra algorithm will find the shortest path to all other vertex of the graph G.

Dijkstra(G,S)

```
1. i = 1 // i represents the row of shortest path table
2. In row i of shortest path table, write 0 for the source vertex and Infinity for other vertex
3. While(row i has unvisited vertices)
       U = extract minimum value from row  i  and mark it.
       i ++
       Copy already marked values and U in new row
       For (each unvisited neighbor V of U)
           Tempdist = distance[U] + Edgecost(U,V)
           If( tempdist < distance[V]
               Distance[V] = tempdist
           End if
       End for
   End while
4. When destination vertex is marked, perform backtracking to find the shortest path.
5. Move upward from the destination vertex.
           i)    If value doesn‟t changes, move upward further
           ii)   Else, mark that row and from that row move upward. Goto step i) until the source
                 vertex is marked
   Similarly we can find the shorted path from source vertex to all other vertex in graph by using
   backtracking.
6. When shortest path is found, display the path from source to destination.
```

Example:



Consider the Source vertex is A

Create a shortest path table. Since there are 4 vertices, the table has 4 columns.

A	B	C	D
0	∞	∞	∞

Write 0 in column A since it is a source vertex.

Smallest unmarked value in row 1 is 0. Hence it is marked. Create another row and copy marked values alone to the new row.

A	B	C	D
0	∞	∞	∞
0			

- Find the minimum value of each edge from the marked vertex A, if there exists an direct edge. Otherwise copy the previous value of that vertex.
- Minimum value of any edge X, Y is found by

  min(value of destination vertex column, value in source vertex column + edge cost)

  Here X is a source vertex and Y is a destination vertex

A	B	C	D
0	∞	∞	∞
0	A to B => min(∞, 0 +6) **6**	A to C =>min(∞, 0+10) 10	A to D => min(∞, 0+35) 35

The smallest unmarked value in 2nd row is found and it is marked. (i.e) 6 is marked

Create another row and copy marked values alone to the new row.

A	B	C	D
0	∞	∞	∞
0	6	10	35
0	**6**	B to C=> min(10, 6 +5) **10**	B to D => min(35, 6+15) 21

The smallest unmarked value in 3rd row is found and it is marked. (i.e) 5 is marked

Create another row and copy marked values alone to the new row.

A	B	C	D
0	∞	∞	∞
0	**6**	10	35
0	**6**	**10**	21
0	**6**	**10**	C to D => min(21, 10+20) **21**

The minimum cost from A to B is 6

The minimum cost from A to C is 10

The minimum cost from A to D is 21

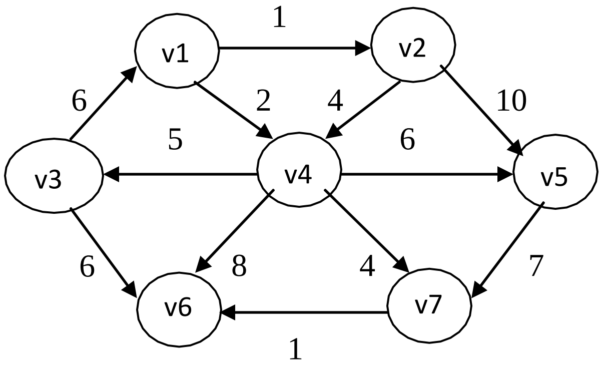
By applying back tracking,

The shortest path from A to B is found as A → B

The shortest path from A to C is found as A → C

The shortest path from A to D is found as A → B →C

Example 2:



Consider the Source vertex is v1

Create a shortest path table. Since there are 7 vertices, the table has 7 columns.

v1	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**
0	∞	∞	∞	∞	∞	∞

Write 0 in column v1 since it is a source vertex.

Smallest unmarked value in row 1 is 0. Hence it is marked. Create another row and copy marked values alone to the new row.

v1	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**
0	∞	∞	∞	∞	∞	∞
0	v1 to v2 => min(∞,0+1) **1**	v1 to v3 => No direct edge ∞	v1 to v4 => min(∞,0+2) 2	v1 to v5 => No direct edge ∞	v1 to v6=> No direct edge ∞	v1 to v7 => No direct edge ∞

The smallest unmarked value in 2nd row is found and it is marked. (i.e) 1 is marked=> ***v2 marked***

Create another row and copy marked values alone to the new row.

v1	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**
0	∞	∞	∞	∞	∞	∞
0	**1**	∞	2	∞	∞	∞
0	**1**	v2 to v3 => No direct edge ∞	v2 to v4 => min(2,1+4) **2**	v2 to v5 => min(∞,1+10) 10	v2 to v6 => No direct edge ∞	v2 to v7 => No direct edge ∞

The smallest unmarked value in 3rd row is found and it is marked. (i.e) 2 is marked =>***v4 marked***

Create another row and copy marked values alone to the new row.

v1	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**
0	∞	∞	∞	∞	∞	∞
0	**1**	∞	2	∞	∞	∞
0	**1**	∞	**2**	10	∞	∞
0	**1**	v4 to v3 => min(∞,2+5) 7	**2**	v4 to v5 => min(10,2+6) 8	v4 to v6 => min(∞,2+8) 10	v4 to v7 => min(∞,2+4) **6**

The smallest unmarked value in 4th row is found and it is marked. (i.e) 6 is marked => ***v7 marked***
Create another row and copy marked values alone to the new row.

v1	v2	v3	v4	v5	v6	v7
0	∞	∞	∞	∞	∞	∞
0	**1**	∞	2	∞	∞	∞
0	**1**	∞	**2**	10	∞	∞
0	**1**	7	**2**	8	10	**6**
0	**1**	v7 to v3 => No direct edge **7**	**2**	v7 to v5 => No direct edge 8	v7 to v6 => min(10, 6+1) 7	**6**

The smallest unmarked value in 5th row is found and it is marked. (i.e) 7 is marked => ***v3 marked***
Create another row and copy marked values alone to the new row.

v1	v2	v3	v4	v5	v6	v7
0	∞	∞	∞	∞	∞	∞
0	**1**	∞	2	∞	∞	∞
0	**1**	∞	**2**	10	∞	∞
0	**1**	7	**2**	8	10	**6**
0	**1**	**7**	**2**	8	7	**6**
0	**1**	**7**	**2**	v3 to v5 => No direct edge 8	v3 to v6 => min(7, 7+6) **7**	**6**

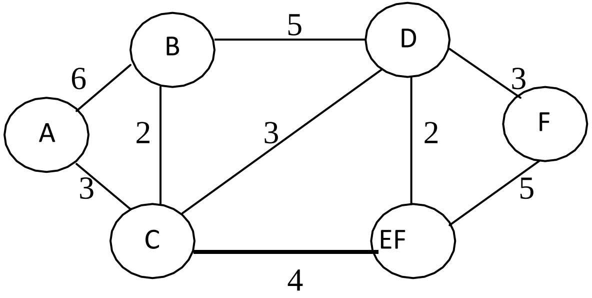
The smallest unmarked value in 6th row is found and it is marked. (i.e) 7 is marked => ***v6 marked***
Create another row and copy marked values alone to the new row.

v1	v2	v3	v4	v5	v6	v7
0	∞	∞	∞	∞	∞	∞
0	**1**	∞	2	∞	∞	∞
0	**1**	∞	**2**	10	∞	∞
0	**1**	7	**2**	8	10	**6**
0	**1**	**7**	**2**	8	**7**	**6**
0	**1**	**7**	**2**	v6 to v5 => No direct edge **8**	**7**	**6**

By applying backtracking, we can find that,

The shortest path from v1 to v2 is v1→v2 and its cost is 1
The shortest path from v1 to v3 is v1→v4→v3 and its cost is 7
The shortest path from v1 to v4 is v1→v4 and its cost is 2
The shortest path from v1 to v5 is v1→v4→v5 and its cost is 8
The shortest path from v1 to v6 is v1→v4→v7→v6 and its cost is 7
The shortest path from v1 to v7 is v1→v4→v7 and its cost is 6

Example 3:



Consider the Source vertex is A

Create a shortest path table. Since there are 6 vertices, the table has 6 columns.

A	B	C	D	E	F
0	∞	∞	∞	∞	∞

Write 0 in column A since it is a source vertex.

Smallest unmarked value in row 1 is 0. Hence it is marked. Create another row and copy marked values alone to the new row.

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
0	A to B => min(∞, 0+6) 6	A to C => min(∞, 0+3) **3**	A to D => No direct edge ∞	A to E => No direct edge ∞	A to F => No direct edge ∞

The smallest unmarked value in 2nd row is found and it is marked.(i.e) 3 is marked => ***C marked***

Create another row and copy marked values alone to the new row.

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
0	6	**3**	∞	∞	∞
0	C to B => min(6, 3+2) **5**	**3**	C to D => min(∞, 3+3) 6	C to E => min(∞, 3+4) 7	C to F => No direct edge ∞

The smallest unmarked value in 3rd row is found and it is marked.(i.e) 5 is marked => ***B marked***

Create another row and copy marked values alone to the new row.

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
0	6	**3**	∞	∞	∞
0	**5**	**3**	6	7	∞
0	**5**	**3**	B to D => min(6, 5+5) **6**	B to E => No direct edge 7	B to F => No direct edge ∞

The smallest unmarked value in 4th row is found and it is marked.(i.e) 6 is marked => ***D marked***

Create another row and copy marked values alone to the new row.

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
0	6	**3**	∞	∞	∞
0	**5**	**3**	6	7	∞
0	**5**	**3**	**6**	7	∞
0	**5**	**3**	**6**	D to E => min(7, 6+2) **7**	D to F => min(∞, 6+3) 9

The smallest unmarked value in 5th row is found and it is marked.(i.e) 7 is marked => ***E marked***

Create another row and copy marked values alone to the new row.

A	B	C	D	E	F
0	∞	∞	∞	∞	∞
0	6	**3**	∞	∞	∞
0	**5**	**3**	6	7	∞
0	**5**	**3**	**6**	7	∞
0	**5**	**3**	**6**	**7**	9
0	**5**	**3**	**6**	**7**	E to F => min(9, 7+5) **9**

All the vertices of the graph is marked.

By applying backtracking, we can find that,

The shortest path from A to B is A→C→B and its cost is 5

The shortest path from A to C is A→C and its cost is 3

The shortest path from A to D is A→C→D and its cost is 6

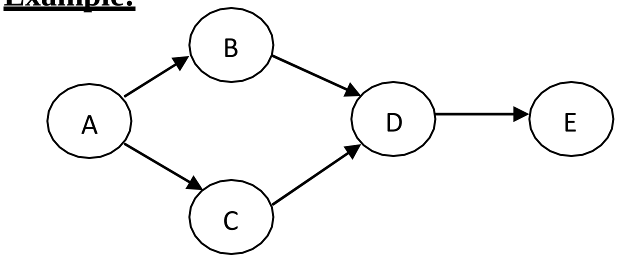
The shortest path from A to E is A→C→E and its cost is 7

The shortest path from A to F is A→C→D→F and its cost is 9

TOPOLOGICAL SORTING

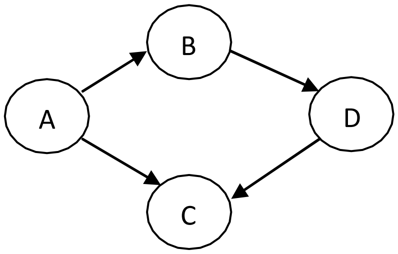
- Topological sorting applies only to Directed Acyclic Graph(DAG)
- Topological sort of DAG G(V,E) is a linear ordering of all its vertices such that, if G contains an edge (u,v), then u always appears before v in the ordering.
- Topological sort of a graph is not unique.(i.e) There may be multiple topological order for a given graph

Example:

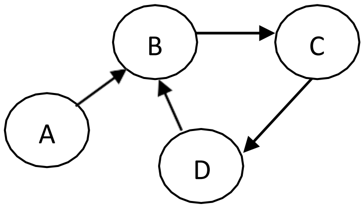


The topological order for a above graph are A,B,C,D,E (or) A,C,B,D,E

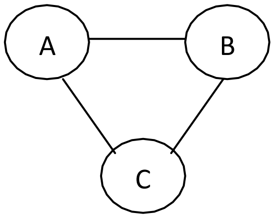
List of graph where topological sorting cannot be applied.



(a)



(b)



(c)

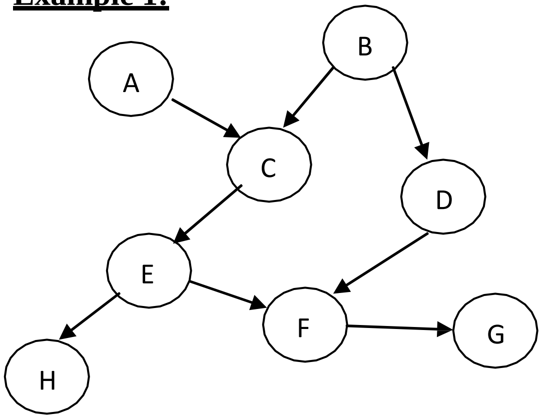
Algorithm:

```
Initialize Stack S and visited set as Empty
While(all vertex of G are not in visited set)
    Choose any arbitrary vertex v.
    If it is not in visited set,
    Call  topo(v)
End while
```

Topo(v)

```
//Given an undirected graph G(V, E) with n vertices and an VISITED array initially set to zero
    VISITED[v] = 1
    for each vertex w adjacent to v do
        if VISITED [w] =0 then
            call Topo (w)
        end if
    push(v)
    end for
```

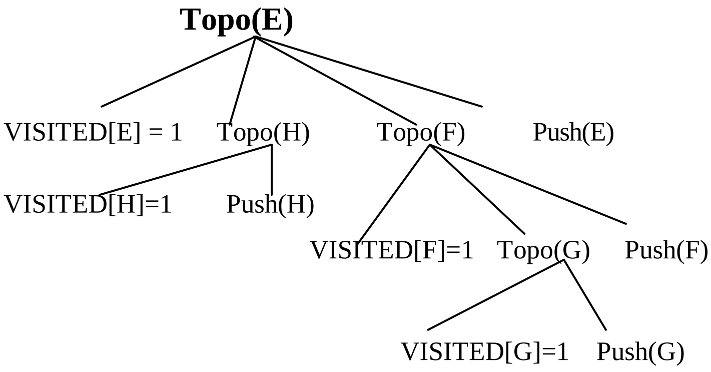
Example 1:



Initially Stack S and visited set is empty

Consider the arbitrary vertex chosen is E

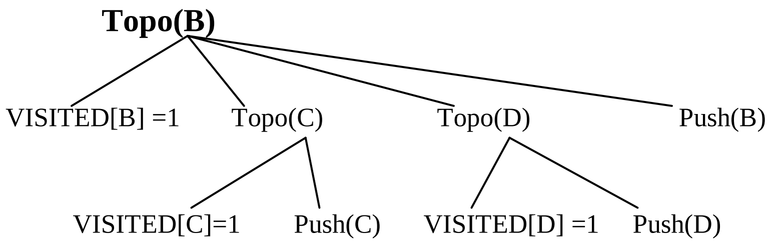
E is passed to the topo function



While all vertices of G are not visited.

Choose an another arbitrary vertex B

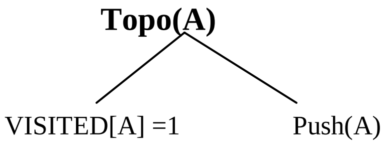
E is passed to the Topo function



While all vertices of G are not visited.

Choose an another arbitrary vertex A

A is passed to the Topo function

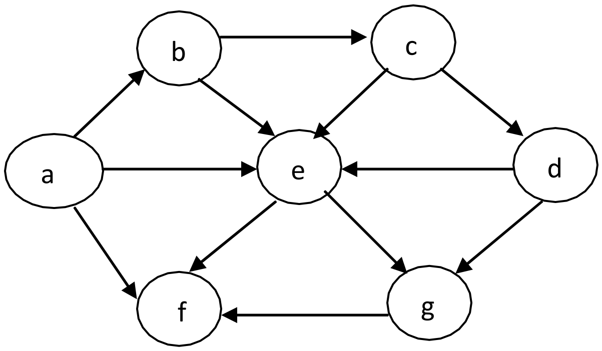


- Since all the vertices of G are visited, Process stops.
- Now the elements of the stack are

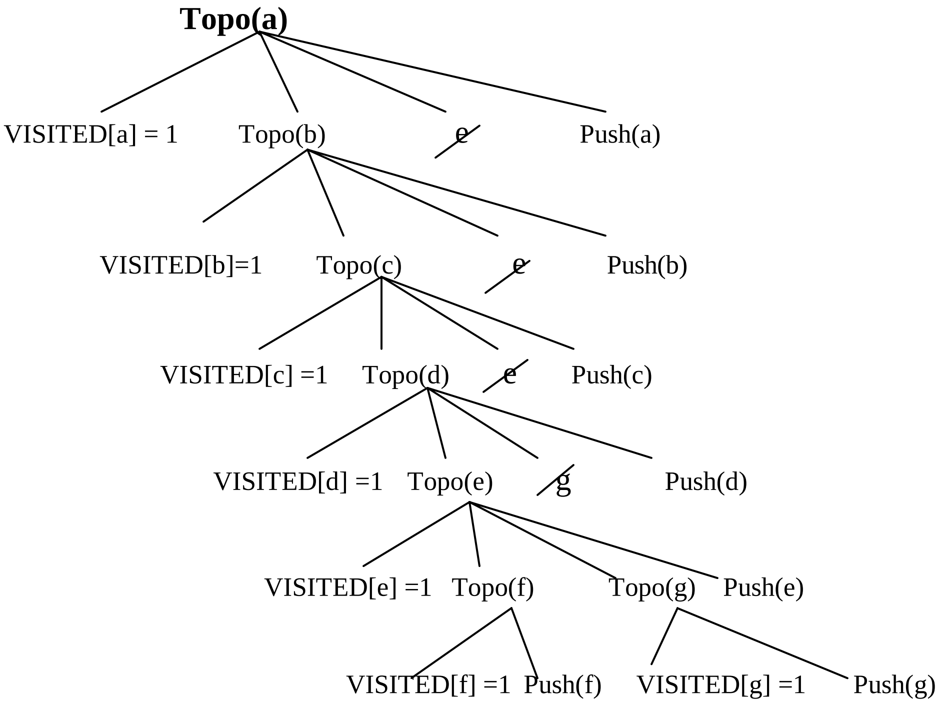
A
B
D
C
E
F
G
H

- The elements from the stack are popped out and it is displayed which gives a topological order of a graph G
- Topological order of the given graph is A,B,D,C,E,F,G,H

Example 2:



Initially Stack S and visited set is empty
Consider the arbitrary vertex chosen is a
a is passed to the topo function



- While all the vertices of graph are visited the contents of stack is popped and displayed, which gives a topological order of a graph

a
b
c
d
e
g
f

- Topological order of the given graph is a,b,c,d,e,g,f

SETS

- A set is an unordered collection of distinct, homogeneous elements.

Terminologies:

i) Bag:

- A bag is an unordered collection of homogeneous elements and all the elements are not necessarily distinct

ii) Union of sets:

- Union of two sets is formed by combining the elements of set 1 with set 2 such that no duplicate values exist.
- If Si and Sj are two sets, their union is given by Si U Sj

iii) Intersection of sets:

- Intersection of two sets is a collection of elements that appears common in both sets.
- If Si and Sj are two sets, their intersection is given by Si ∩ Sj

iv) Difference of sets:

- Difference of two sets is the set of elements from one set that do not appear in common in a second set.
- If Si and Sj are two sets, their difference is given by Si – Sj

v) Subset:

- A set is said to be a subset of another set, if all the elements in first set appears in second set

vi) Null set:

- A set is said to be null set, if it doesn‟t contain any element in it.
- Null set is denoted by Ø

vii) Disjoint set:

- Two sets Si and Sj are said to be disjoint, if the intersection of two sets is an empty set
- Si ∩ Sj = Ø

vii) Cardinality:

- The number of elements in a set is termed as cardinality

viii) Equality:

- Two sets are said to be equal, if all the elements of first set appear in the second set and all the elements from the second set also appears in the first set.
- If Si and Sj are equal, then it is denoted as Si ≡ Sj

REPRESENTATION OF SETS:

The various ways of representing sets are

i) Linked list representation of sets

ii) Hash table representation of sets

iii) Bit vector representation of sets

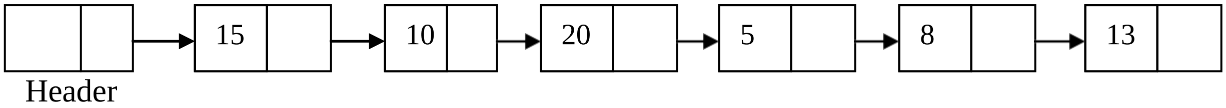
iv) Tree representation of sets

i) Linked list representation of sets:

It is a simplest and straight forward representation of a set. It allows dynamic storage facility.

Consider a Set S = { 15, 10, 20, 5, 8, 13}

- Head node doesn‟t store any data. Head points to the first element in aset



Operation on linked list representation of sets:

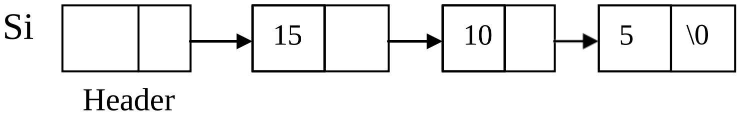
a) Union

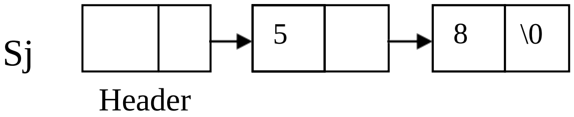
Union of two given sets Si and Sj are performed to get set S

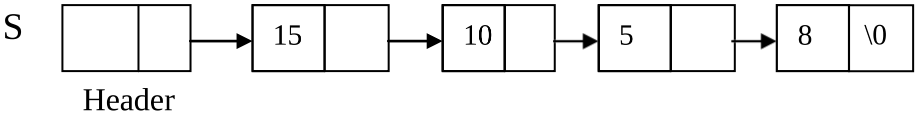
(i.e) Set S = Si U Sj

Initially S is empty. As a first step copy all the elements of Si to S. Then each element in Sj is inserted at the end of S only if that element is not already present.

Consider, Set Si ={ 15, 10, 5}and Sj = { 5 , 8 }

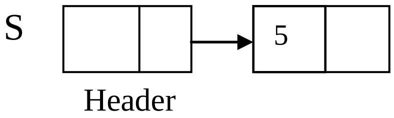






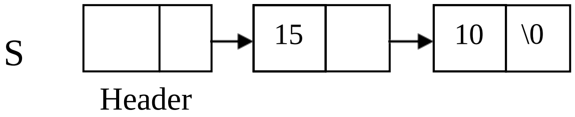
b) Intersection

- If Si and Sj are two sets, their intersection is given by S = Si ∩ Sj
- Initially S is empty. The values in Si and Sj are compared and the common value in both set is alone inserted to Set S. Consider, Set Si ={ 15, 10, 5}and Sj = { 5 , 8 }



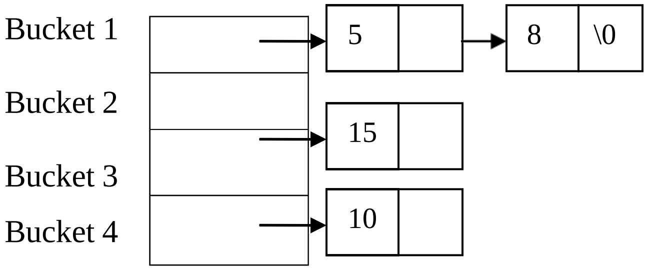
c) Difference:

- If Si and Sj are two sets, their difference is given by S = Si – Sj
- Initially S is empty. Find the common value in both Si and Sj. Remove that common value from Si and then copy the values in Si to S.
- Consider, Set Si ={ 15, 10, 5}and Sj = { 5 , 8 }



ii) Hash table representation of sets

- Hash table contains several buckets, where each bucket can store a list of elements
- Each bucket can hold an arbitrary number of elements.
- Consider Set S = { 15, 10, 5, 8}
- Consider H(x) is a hash function which can store elements of set in any one of the bucket
- Consider a hash table which contains 4 buckets.

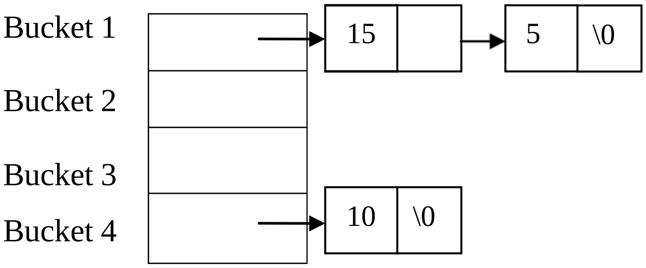


Operations on Hash table representation of sets

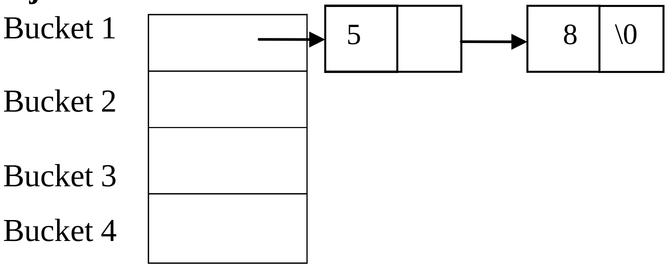
Let Si = {15, 5, 10}, Sj = {5, 8}

- The union operation can be performed by pairing the corresponding buckets.
- The union of hash table refers to union of corresponding buckets in the hash table.
- The intersection of hash table refers to common value in corresponding buckets in the hash table.
- The difference of hash table gives the Set Si after removing the common value between Si and Sj

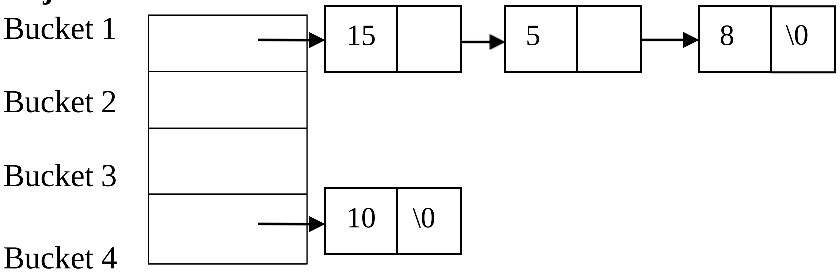
Let Set Si



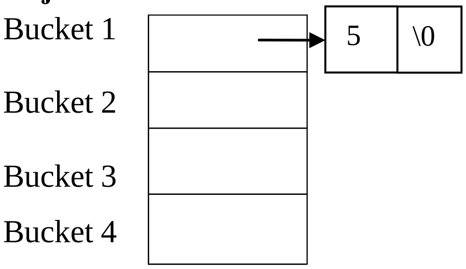
Let Set Sj



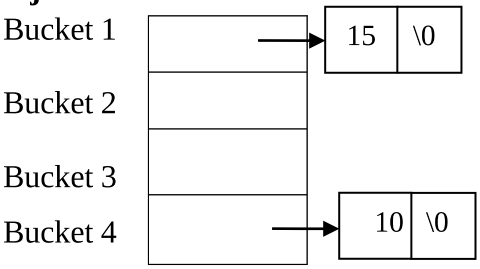
S = Si U Sj



S= Si ∩ Sj



S = Si - Sj



iii) Bit vector representation of sets:

Bit vector representation specifies the presence or absence of an element in set.

Example:

A set giving the record of a student who scored above 90 %

{ 0, 1, 1, 0, 0, 1, 0}. Here 0 specifies the absence of record of student who scored above 90 % and 1 specifies the presence of record of student who scored above 90 %

Operations on bit vector representation of set

Union

To obtain the union of sets Si and Sj, Bitwise OR operation can be performed

Consider Si = { 1 0 1 0 1 1 0 }

Sj = { 0 0 1 1 0 1 1 }

The set S = Si U Sj = { 1 0 1 1 1 1 1}

Intersection

To obtain the intersection of sets Si and Sj, Bitwise AND operation can be performed

Consider Si = { 1 0 1 0 1 1 0 }

Sj = { 0 0 1 1 0 1 1 }

The set S = Si ∩ Sj = { 0 0 1 0 0 1 0}

Difference

Difference of Si and Sj is the set of values that appear in Si but not in Sj

This can be done performing Bitwise AND on the Complement of Sj

Consider Si = { 1 0 1 0 1 1 0 }

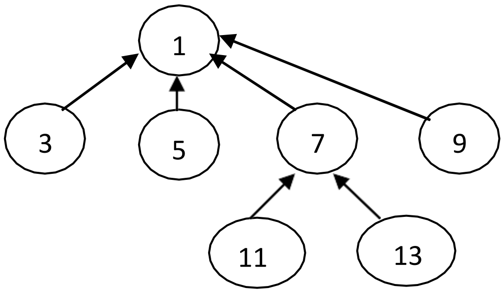
Sj = { 0 0 1 1 0 1 1 } => Sj‟ = { 1 1 0 0 1 0 0}

Si – Sj => Si ∩ Sj‟ = { 1 0 0 0 1 0 0}

iv) Tree representation of sets

- In tree representation of set, a tree is used to represent one set and each element in set has the same root.
- The trees are not necessarily binary tree

Consider a set S = { 1, 3, 5, 7, 9, 11, 13 }



Representation of set in an array (i.e) Parent array of the given tree

-	-	1	-	1	-	1	-	1	-	7	-	7	-	-
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Operation on tree representation of set:

a) Exclusion:

Case 1 : Exclusion of an element which is a leaf node

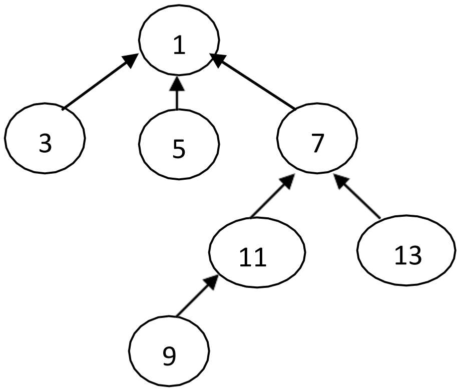
- Let the element to be excluded be i. Make the parent[i] = Null
- Reduce the number of elements of set by 1

Case 2 : Exclusion of an element which is not a leaf node

- Let the element to be excluded be i.
- Traverse the parent array to select an element j such that it is a successor of i and is a leaf node.
- Make parent[j] = parent [i]. Then, Make parent[i] = Null
- Traverse the parent array to replace all the occurrence of i by j

Example:

Exclusion of element 7



Parent array of the given tree

-	-	1	-	1	-	1	-	11	-	7	-	7	-	-
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Element to be excluded is 7 (i.e) i is 7

Traverse the parent array from 7 to select the successor which is a leaf node. (i.e) j is 9

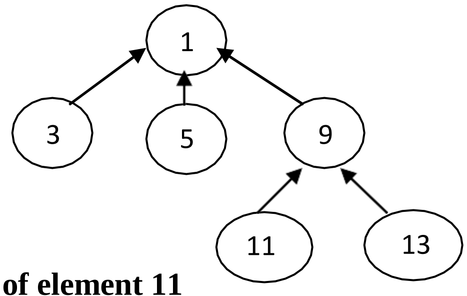
Parent[9] = 1

Parent[7] = Null

Traverse parent array and replace the occurrence of 7 by 9

Parent array of the given tree

-	-	1	-	1	-	-	-	1	-	9	-	9	-	-
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15



Exclusion of element 11

Parent array of the given tree

-	-	1	-	1	-	-	-	1	-	-	-	9	-	-
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

APPLICATION OF SETS:

i) Hash table representation of set is used in spelling check

ii) Bit array representation of set is used in information system

iii) Tree representation of set is used in Client – Server environment

i) Spelling checker:

The spelling checker system maintains the dictionary in a form of one hash table. The input words to be checked is maintained in another hash table. The intersection operation of these two hash tables is performed to check the misspelled word.

ii) Information system:

Bit strings are used for information storing and retrieval

Consider a student table

Register number	Sex	Department
101	Male	ECE
102	Male	CSE
103	Female	IT
104	Male	Mechanical
105	Female	CSE

In this technique, the length of bit string is equal to the number of records

We need a bit array to store a set of bit strings

Bit Array	**Bit String**
Male	11010
Female	00101
CSE	01001

Information retrieval using bit string

Consider, we need a record of Female student of CSE department

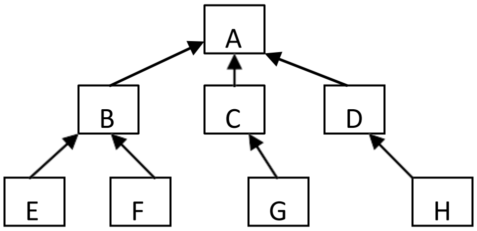
Female ∩ CSE => 00101 ∩ 01001

=> 0000**1**

It represents the register number 105 is a female student of CSE department

iii) Client – Server environment:

- Tree representation of sets is used in client-server environment
- A square symbol represents a client-server system. In this, any number of client system can be connected to the server system in an hierarchy.
- The clients of different server can communicate with each other provided that, both that servers belongs to the same set.
- Here B,C,D are client of server A. E,F are clients of server B. G is client of server C. H is a client of server D
- The client E can communicate with client G since the servers of E and G belong to same set.

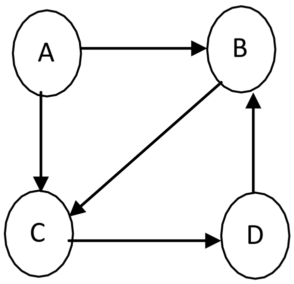


TRANSITIVE CLOSURE

- Given a vertex v, it is possible to find the reachability of all other vertices of graph G.
- Reachable means that there is a path from vertex u to v.
- If path exist between u to v, it is denoted as 1. If path doesn‟t exist, it is denoted as 0.
- The transitive closure of a given graph can be represented as Reachability matrix.

Example 1:

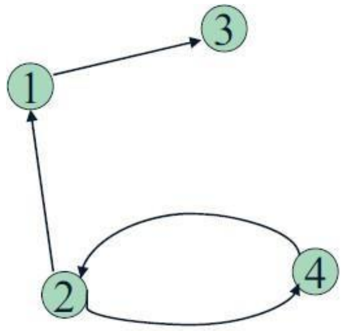
Consider a following graph



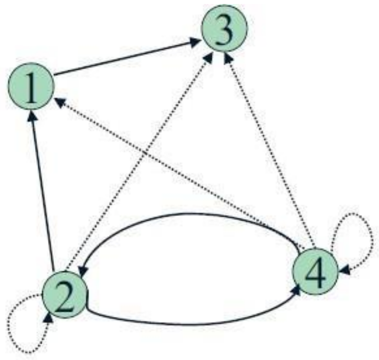
Reachability matrix:

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

Example 2:



$$\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{array}$$

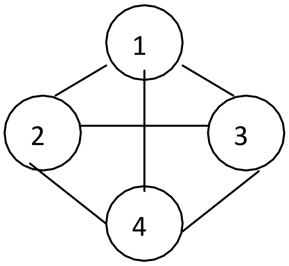


$$\begin{array}{cccc} 0 & 0 & 1 & 0 \\ 1 & \mathbf{1} & \mathbf{1} & 1 \\ 0 & 0 & 0 & 0 \\ \mathbf{1} & 1 & \mathbf{1} & \mathbf{1} \end{array}$$

UNIT – 4 TWO MARKS

1. Define graph. Give example

- A graph G consists of a set of Vertices (nodes) and a set of Edges (arcs).
- Graph can be represented as
  - **G(V,E)**. V is a finite and non-empty set of vertices.
  - E is a set of pair of vertices; these pairs are called as edges.
- V(G) = Set of vertices of graph G and E(G) = Set of edges of graph G



Here V(G)=(1,2,3,4) and E(G) = {(1,2),(1,3),(1,4),(2,3),(2,4), (3,4)}

2. Define Undirected Graph (or) Unqualified graph:

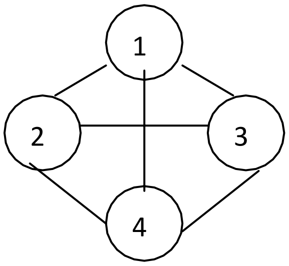
- Undirected graph is a graph in which, the pair of vertices representing the edges is unordered.
- Edges in undirected graph doesn"t specify the direction

3. Define Directed graph (or) Digraph:

- Directed graph is a graph in which, the pair of vertices representing the edges is ordered.
- Edges in directed graph specify the direction

4. Define Path in a graph:

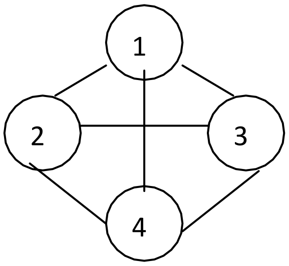
- Path between vertices u,v is a sequence of edges that connects u and v



- Path between 1 to 4 are {1 – 2 –4, 1–4, 1–3–4, 1–2–3–4, 1–3–2–4}

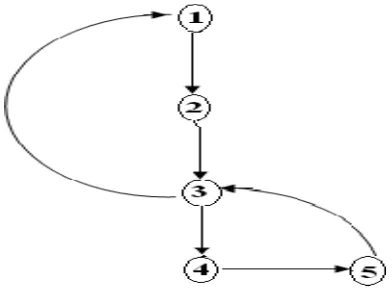
5. What is meant by a Connected graph?

- In Undirected graph, If there is a path from any vertex to every other vertex in a graph, then the graph is called connected graph.



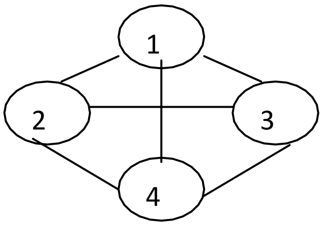
6. What is meant by Strongly Connected graph?

- In Directed graph, If there is a path from any vertex to every other vertex in a graph, then the graph is called strongly connected graph.



7. What is meant by a Complete Graph?

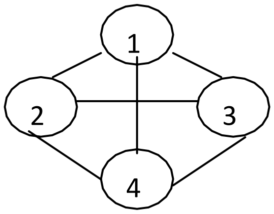
- A Graph is a complete, if there is an edge between every pair of vertices.
- It has exactly n(n-1)/2 number of edges.



8. Define Sub Graph.

A sub-graph of G is a graph G„ such that V(G‟) ⊆ V(G) and E(G „) ⊆E(G).

<u>Some of the Subgraphs of G</u>



(a)

Graph G

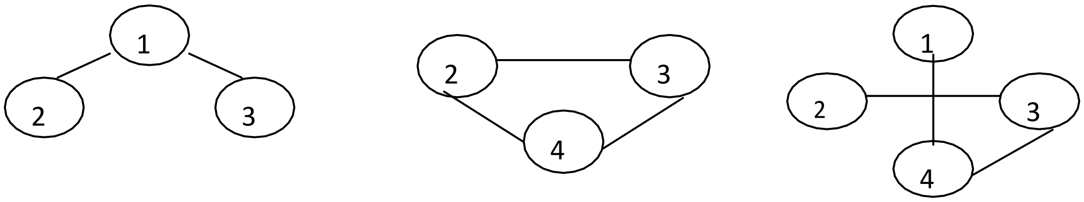
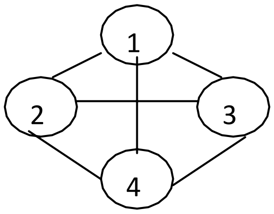


Fig. Sub graphs of G

9. Define Adjacent Vertex in graph

A vertex v1 is said to be a adjacent vertex of v2, if there exist an edge (v1,v2) or (v2,v1).



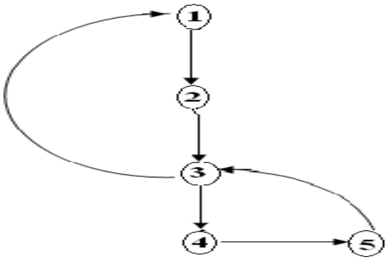
(a)

Graph G

Adjacent(1) = {2,3,4}, Adjacent(2) = {1,3,4}, Adjacent(3) = {1,2,4}, Adjacent(4) = {1,2,3}

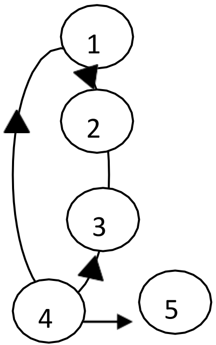
10. What is meant by Length of the graph?

The length of the graph is the number of edges in it.



11. What is meant by Weakly Connected Graph?

If there does not exist a directed path from one vertex to another vertex then it is said to be a weakly connected graph.



12. What is meant by Cycle in graph?

A cycle is a path in which the first and the last vertices are the same.

13. Define Degree of a graph.

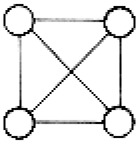
- The number of edges incident on a vertex determines its degree. There are two types of degrees for directed graph **In-degree** and **out-degree**.

➢ **In-Degree** of the vertex V is the number of incoming edges

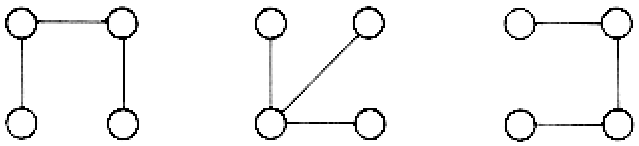
➢ **Out-Degree** of vertex V is the number of edges that leaves the vertex

14. What is a spanning tree?

Spanning tree is a subset of Graph G, which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected



Graph G



Spanning trees of G

15. Define minimum spanning tree.

Minimum spanning tree is a spanning tree that has minimum weight/cost than all other spanning trees of the same graph.

15. How does a graph differ from a tree?

Trees	**Graph**
Tree is a *minimally connected graph* and having only one path between any two vertices.	In graph there can be more than one path
Tree doesn‟t have loop, circuit or self loop	Graph can have loop, circuit or self loop
Tree is DAG(Directed Acyclic graph)	Graph may be cyclic or Acyclic
Parent child relationship exist	No such relationship exists
Tree can be traversed using inorder, preorder and postorder traversals	Graph can be traversed using Breadth First Search and Depth First Search

16. Define multistage graph.

- A multistage graph G=(V,E) is a directed graph in which vertices are partitioned into K>=2 disjoint set (set Vi) where [1<= i <= K]. In addition, if (u, v) is an edge E then u ∈vi, v ∈ vi+1. Each set vi is called stage in a graph.
- Let c(i, j) be the cost of edge. The cost of a path from (S to D) is the sum of costs of the edges on the path.
- The Multistage graph problem is to find the minimum cost path from “S” to “D”.

17. What are the two ways to maintain graph G in a memory of computer? (or) What are the different ways to represent a graph?

- Set representation
- Adjacency matrix representation
- Adjacency list (or) Linked list representation

18. Write the advantages of topological sorting.

- It is used in scheduling a sequence of jobs based on dependencies
- It is used in determining the order of compilation, data serialization, resolving dependencies in linkers.
- It is used to quickly compute shortest path with a weighted acyclic graph

19. Define topological sort.

- Topological sort of DAG G(V,E) is a linear ordering of all its vertices such that, if G contains an edge (u,v), then u always appears before v in the ordering.
- Topological sort of a graph is not unique.

20. State the applications of graph.

1. ***Facebook:*** Each user is represented as a vertex and two people are friends when there is an edge between two vertices. Similarly friend suggestion also uses graph theory concept.
2. ***Google Maps:*** Various locations are represented as vertices and the roads are represented as edges and graph theory is used to find shortest path between two nodes.
3. ***Recommendations on e-commerce websites:*** The "Recommendations for you" section on various e-commerce websites uses graph theory to recommend items of similar type to user"s choice.
4. ***Broadcasting in networks***

21. Define the term set and bag in set ADT

- A set is an unordered collection of distinct, homogeneous elements.
- A bag is an unordered collection of homogeneous elements and all the elements are not necessarily distinct

22. What do you know about graph traversal

- Two types of graph traversal that can be performed is Breadth First Search(BFS) and Depth First Search(DFS).
- BFS explores siblings before exploring its children
- DFS explores all the nodes reachable from X before exploring its siblings

23. Explain union by rank in a set

- *Union by rank* always attaches the shorter tree to the root of the taller tree. Thus, the resulting tree is no taller than the originals unless they were of equal height, in which case the resulting tree is taller by one node.
- To implement *union by rank*, each element is associated with a rank. Initially a set has one element and a rank of zero. If two sets are unioned and have the same rank, the resulting set's rank is one larger; otherwise, if two sets are unioned and have different ranks, the resulting set's rank is the larger of the two.

25. Define Adjacency matrix.

- Adjacency matrix is a representation used to represent a graph with zeros and ones.
- A graph containing n vertices can be represented by a matrix with n rows and n columns.
- The matrix is formed by storing 1 n its ith and jth column of the matrix, if there exists an edge between ith and jth vertex of the graph and 0 if there is no edge between ith and jth vertex of the graph .
- *Adjacency matrix is also referred as incidence matrix.*

26. Define weighted graph.

A graph is said to be a weighted graph if every edge in the graph is assigned some weight or value. The weight of an edge is a positive value that may be representing the distance between the vertices or the weights of the edges along the path.

27. What are the two methods for finding shortest path?

- Single source shortest path
- All pair shortest path

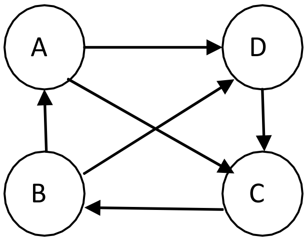
28. Differentiate BFS and DFS

DFS	BFS
Backtracking is possible	Backtracking is not possible
Stack can be used to implement DFS	Queue is used to implement BFS
It explores all the nodes reachable from X before exploring its siblings.	BFS explores siblings before exploring its children
Search is done in depth order	Search is done in same level order

29. Define shortest path problem. Give Example.

- The shortest path problem is the problem of finding a PATH between 2 nodes(source and destination), such that the sum of weight of its constituent edge is minimized.
- **Eg**: finding quickest way to go to one location to another.

30. Find the adjacency matrix of a given graph.



Adjacency matrix:

$$A= \begin{array}{c|cccc} & \mathbf{A} & \mathbf{B} & \mathbf{C} & \mathbf{D} \\ \hline \mathbf{A} & 0 & 0 & 1 & 1 \\ \mathbf{B} & 1 & 0 & 0 & 1 \\ \mathbf{C} & 0 & 1 & 0 & 0 \\ \mathbf{D} & 0 & 0 & 1 & 0 \end{array}$$

31. What are the different representations of set?

i) Linked list representation of sets ii) Hash table representation of sets
iii) Bit vector representation of sets iv) Tree representation of sets

32. What are the various operations on set ?

- Union, Intersection, Difference, Exclusion.

32. What are the various applications of set?

i) Hash table representation of set is used in spelling check

ii) Bit array representation of set is used in information system

iii) Tree representation of set is used in Client – Server environment

UNIT – 5

TABLES:

- Table is a data structure used to store the records.
- Access table stores the location of all records in a file. The method of accessing the record in a file using an access table is call table lookup, which is independent on number of records.
- It helps in efficient retrieval of records

Types of Tables:

i) Inverted tables:

Consider a telephone company maintains the records of all customers. These records can be used to serve several purposes. List of all customers based on alphabetical order of names, List of all customers based on lexicographical ordering of address, List of all customers based on ascending order of phone numbers

Storing these records in normal table has several drawbacks.

1. Requirement of extra storage: Three times the actual memory
2. Difficulty in modification of records: If the customer changes his address it need to beupdated in all the three tables.

To avoid maintaining multiple tables for different purpose, We use a single inverted table.

Original table

Index	Name	Address	Phone number
1	Raj	Nethaji street	250121
2	Anbu	Medical colony	230131
3	Akash	Officer‘s colony	260158
4	Jagan	Mgr street	254836

Table after performing ascending order of names, address and phone number

Name	Address	Phone number
Akash	Medical colony	230131
Anbu	Mgr street	250121
Jagan	Nethaji street	254836
Raj	Officer‘s colony	260158

The index of corresponding name, address and phone number in original table is written in the inverted table.

INVERTED TABLE:

Name	Address	Phone number
3	2	2
2	4	1
4	1	4
1	3	3

If we search the customer name or address or phone number, with the help of index, the corresponding record is retrieved from the original table.

ii) Jagged tables:

- In jagged table, Each row in a table has varying number of elements.
- Given a linear index value, the access array is searched to find the appropriate data in a table.

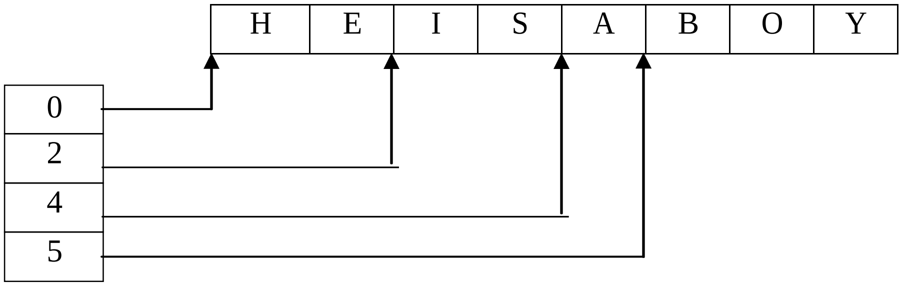
0
2
4
5

Access table

H	E	
I	S	
A		
B	O	Y

Jagged table

Storing the elements of jagged table as an array, we get



Access table

Accessing elements using access table

iii) Rectangular tables:

- Rectangular tables are similar to matrices
- Elements are stored in a form of rows and columns
- (i.e) m*n. where m represents the number of rows and n represents the number of columns

$$A= \left.\begin{matrix} a11 & a12 & a13\ldots\ldots\ldots a1n \\ a21 & a22 & a23\ldots\ldots\ldots a2n \\ a31 & a32 & a33\ldots\ldots\ldots a3n \\ \ldots\ldots\ldots\ldots\ldots\ldots \\ am1 & am2 & am3\ldots\ldots\ldots amn \end{matrix}\right)$$

iv) HASH TABLES:

- **Hashing** is an effective technique to calculate the direct location of a data record on the disk using hash function.

Several types of hash function are:

i) Mid-square method

In this method, A key is multiplied by itself and the address is obtained by choosing an appropriate number of bits or digits from the middle of the square. The selection of bits or digits based on the table size and also they should fit into one computer word of memory.

Example: Consider a key, 56789 and when it is squared we get 3224990521. If the three digit address is needed, then positions 5 to 7 may chosen, giving address 990.

ii) Division method

In this method, integer x is to divide by M and d then to use the remainder modulo M. The hash function is

H(x) = x mod M

Great care should be taken while choosing value for M and preferably it should be an even number. By making M a large prime number the keys are spread out evenly.

iii) Folding method

A key is partitioned into a number of parts, each of which has the same length as the required address. The parts are then added together, ignoring the final carry, to form an address. For eg., if the key 356942781 is to be transformed into a three-digit address.

Example:

356, 942 and 781 are added to yield 079.

iv) Digit analysis method

Digit analysis forms an addresses by selecting and shifting digits or bits of the original key. ***Example:*** A key 7546123 is transformed to the address 2164 by selecting digits in positions 3 to 6 and reversing their order.

Hash file organization:

- Data is stored in buckets. A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a hash function.

i) STATIC HASHING:

- In static hashing, number of buckets in the hash table is fixed.

Consider a hash function

h(x) = x , where h(x) is a hash function, x is the search key

- If we use the above hash function, the key element will be stored at the corresponding index.
- If we want to store 2, 4, 5, it will be stored at location 2,4, and 5 respectively in a hash table.
- **A hash table consists of several buckets.** Consider a hash table of size 6
- h(2) = 2, h(4) = 4, h(5) =5

0	
1	
2	2
3	
4	4
5	5

Hash table of size 6

- If we want to store, key 100, then 100 buckets need to be created in hash table.
- Hence the hash function, h(x) = x leads to wastage of memory.
- To improve it, hash function is modified as
    - **h(x) = x % size** , where size represents the size of hash table

Consider a hash table of size 10.

h(x) = x % 10

Insert the key values 3, 17, 18, 27

0	
1	
2	
3	3
4	
5	
6	
7	17
8	18
9	

Hash table of size 10

When 27 is to be inserted, h(27) = 27 % 10 => 7. When we try to insert 27 at position 7, collision occurs.

Collision resolution techniques:

To avoid the collision, when inserting elements into the Hash table, two techniques are used

i) Open hashing

a. Chaining

ii) Closed hashing

a. Linear Probing

b. Quadratic Probing

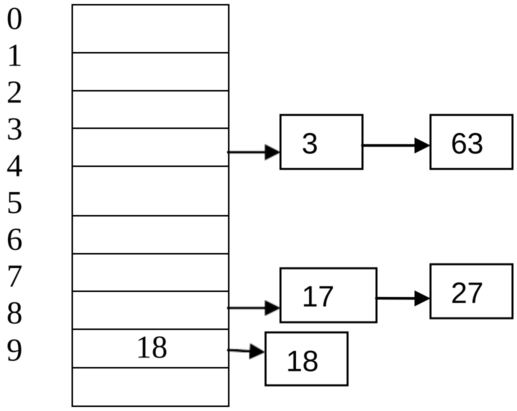
i) Open hashing:

a. Chaining:

- In chaining, each index in a hash table maintains a set of elements in a chain, if multiple elements corresponds to the same index.
- Consider a hash table of size 10.

h(x) = x % 10

Insert the key values 3, 17, 18, 27, 63



Hash table of size 10

ii) Closed hashing:

a. Linear Probing:

- In linear probing, if there is no space for storing the element in corresponding index, then Find the next free space and store the element in it.
- Consider a hash table of size 10.

h(x) = x % 10

Insert the key values 3, 17, 18, 27

h(3) = 3 % 10 => h(3) =3

h(17) = 17 % 10 => h(17) =7

h(18) = 18 % 10 => h(18) =8

h(27) = 27 % 10 => h(27) = 7. Since 17 and 27 corresponds to same index 7, 27 is stored in next free space

Hence, the Hash function is modified as

h'(x) = index position + i , where i ranges from 1 to n

h'(27) =7 + 1 => h'(27) = 8. Since index position 8 is already filled by 18, next position is checked

h'(27) = 7 +2 => h'(27) =9. Since index position 9 is free, 27 is stored in index 9.

Insert the key values 3, 17, 18, 27

0	
1	
2	
3	3
4	
5	
6	
7	17
8	18
9	27

Hash table of size 10

- Drawback of this approach is that, elements are clustered in a single place

<u>b. Quadratic probing:</u>

- To avoid the clustering of elements, Quadratic probing is used
- Hash function
  - **h(x) = x % 10**
  - **h’(x) = index position + x^2**

Insert the key values 3,22, 31

h(3) = 3 % 10 => h(3) =3

h(22) = 22 % 10 => h(22) =2

h(32) = 32 % 10 => h(32) =2

since index position 2 is already filled by 22,

h‘(32) = 2 + 1^2 = h‘(32) = 3. Position 3 is also already filled

h‘(32) = 2 + 2^2 = h‘(32) = 6. Since position 6 is free, element 32 is filled at position 6. Hence clustering of elements is avoided.

0	
1	
2	22
3	3
4	
5	
6	32
7	
8	
9	

Hash table of size 10

ii) DYNAMIC HASHING (or) EXTENDABLE HASHING:

- In dynamic hashing, ***Buckets are allocated on-demand dynamically.***
- It specifies a binary value for each bucket.
- If the number of bit is 1, then possible number of buckets are 0 and 1
- If the number of bits is 2, then possible number of buckets are 00, 01, 10 and 11
- If the number of bits is 3, then possible number of buckets are 000, 001, 010, 011, 100, 101, 110, 111.

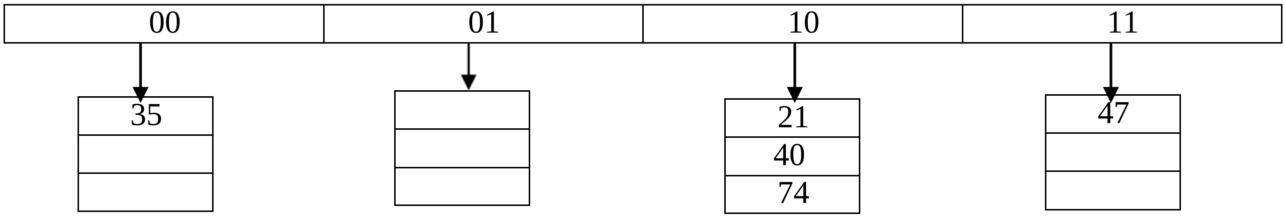
Consider a hash function

h(x) = x % 32

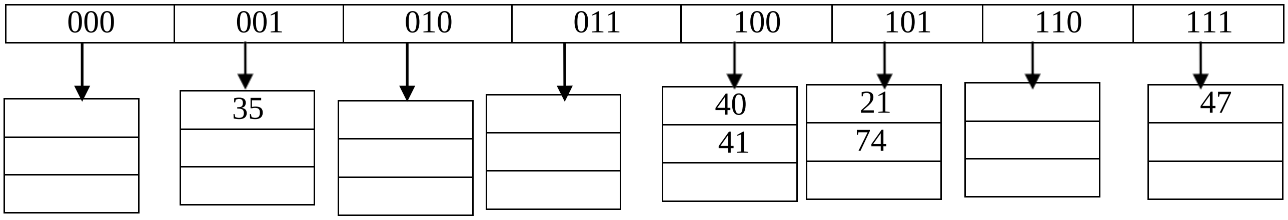
Consider, we want to insert 35, 21, 47, 40, 74

Hash function, h(x)	**x % 32**	**Remainder value**	**Binary value of the remainder**
h(35)	35%32	3	0011
h(21)	21%32	21	10101
h(47)	47%32	15	1111
h(40)	40%32	8	1000
h(74)	74%32	10	1010

- In a binary value of the remainder, consider 1st two bits of each number and store the corresponding number in that bucket. Consider each bucket can store 3 values



- *When we try to insert 41, h(41) => 41 % 32 => 9 => 1001, 41 should be inserted in bucket 10. But it leads to overflow in bucket 10.*
- Hence the ***number of bucks is increased*** by considering the *1st three binary digits* of the remainder



Advantage:

- It is good for database that grows and shrinks in size

Disadvantage:

- As the data size increases, the number of buckets also increased

SYMBOL TABLE:

Symbol table is a Data Structure which contains an information about the identifier.

Symbol table contains a set of locations where a record of each identifier is present.

Symbol table has two fields

- Identifier name
- Memory location

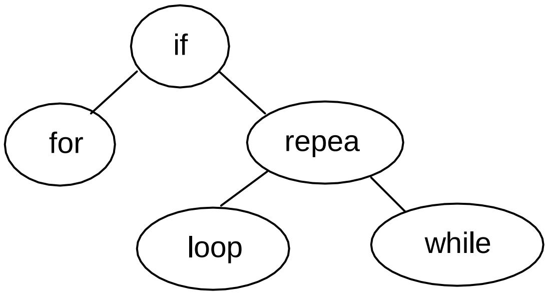
Identifier name	Memory location
X	1000
Y	2000
Z	3000

- Here X, Y, Z are the variables used in the program
- Column 1 => Contains the entry for the variables
- Column 2 => Contains the address where the value of the variable is stored

Two ways to implement symbol table:

i) STATIC TREE TABLE:

- In Static tree table, identifiers are known in advance. No new insertion and deletion of identifiers are allowed.
- Example: Reserved keywords
- Sorting the names and store them sequentially using Binary Search Tree(BST)
- **Example:** Perform insertion and deletion of keywords in BST
- **Insert keywords** if, for, repeat, loop, while in BST



ii) DYNAMIC TREE TABLE:

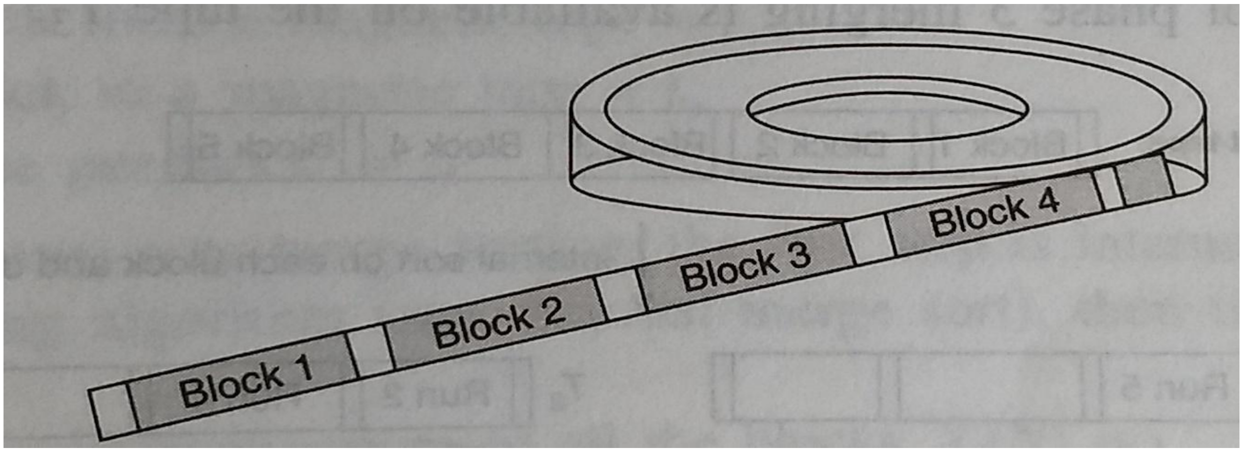
- In Dynamic tree table, identifiers are not known in advance. Insertion and deletion of identifiers are allowed dynamically.
- **Example:** Perform Insertion in BST, AVL tree with any example.

EXTERNAL SORTING

- When sorting is performed on data which is stored on external storage devices, it is called external sorting.
- Application such as Geographical Information System(GIS), Bioinformatics, Statistical analysis, soft computing need to store large amount of data.
- When the data need to be stored is very large and it cannot be accommodated in main memory, then those data is stored in external storage devices like magnetic tapes, hard disk etc.
- Some of the external sorting algorithms are: Balanced two-way merge, Multi-way merging, Poly-phase merge sort, Oscillating sorting, External radix sort.

SORTING ON TAPES

- Data in magnetic tape is divided into n-blocks of equal size.



Data organization on a magnetic tape

i) Balanced 2 – way merge sort

- Balanced 2- way merge sort requires 4 magnetic tapes to sort the data.
- 2 – way merge sort combines two ascending runs into single ascending run.(A sorted segment of a file is called ascending run)
- Let T1, T2, T3, T4 be the four tapes. Size of each tape need to store entire data under sort

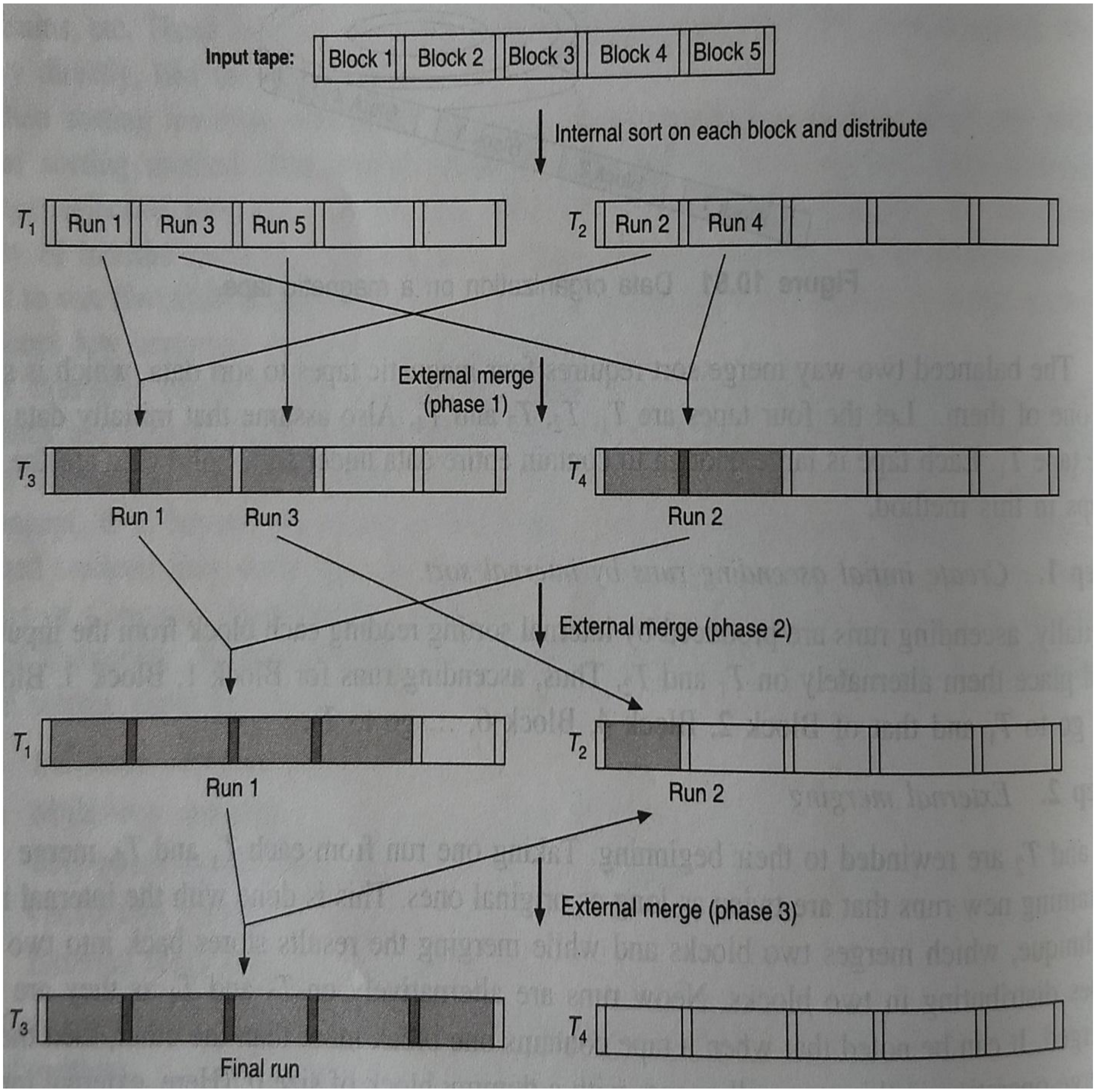
Steps in performing sorting:

- **Create initial ascending runs by internal sort**

  i) Read each block from Input tape and place that alternatively on tape T1 and T2

  ii) Thus, ascending runs for Block1, Block3, Block5…etc are stored in T1 and Block2, Block4, Block6…etc are stored in T2
- **Perform external merge**

  i) While performing external merging, runs are merged and stored on alternate tapes, until the final run is obtained.



Analysis of Balanced two – way merge sort:

- Total time for performing internal sorting of all blocks

  T(N) = O(N), where N is a number of data blocks under sort.

- Time to retrieve data from tape and storing data into tape, (i.e) external device time is

  Te(N) = (tr + tw) * N

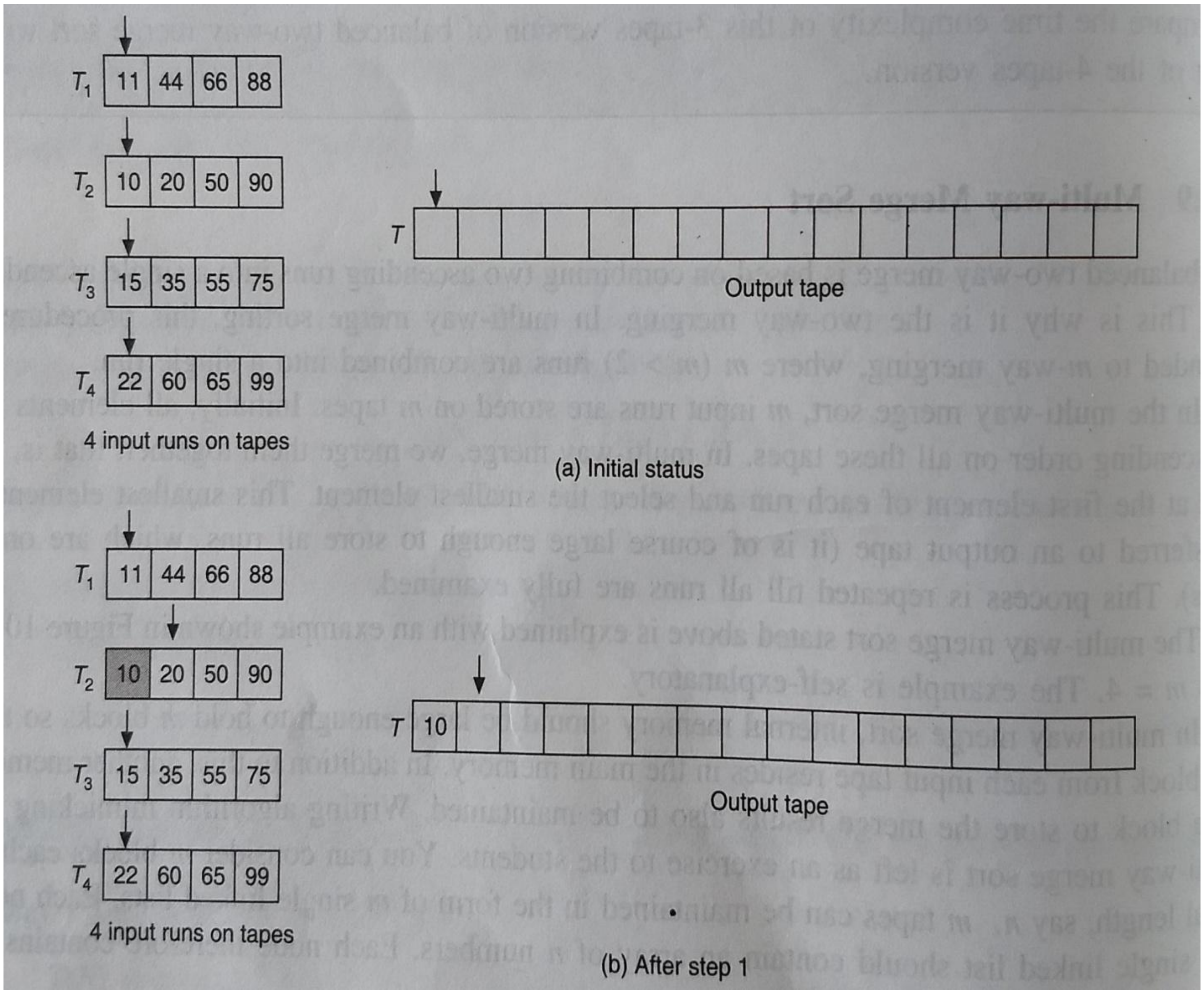
  Where, tr is time to read data from tape

  tw is time to write data into tape

- Time complexity of performing 2 – way merge sort is O(N log N)

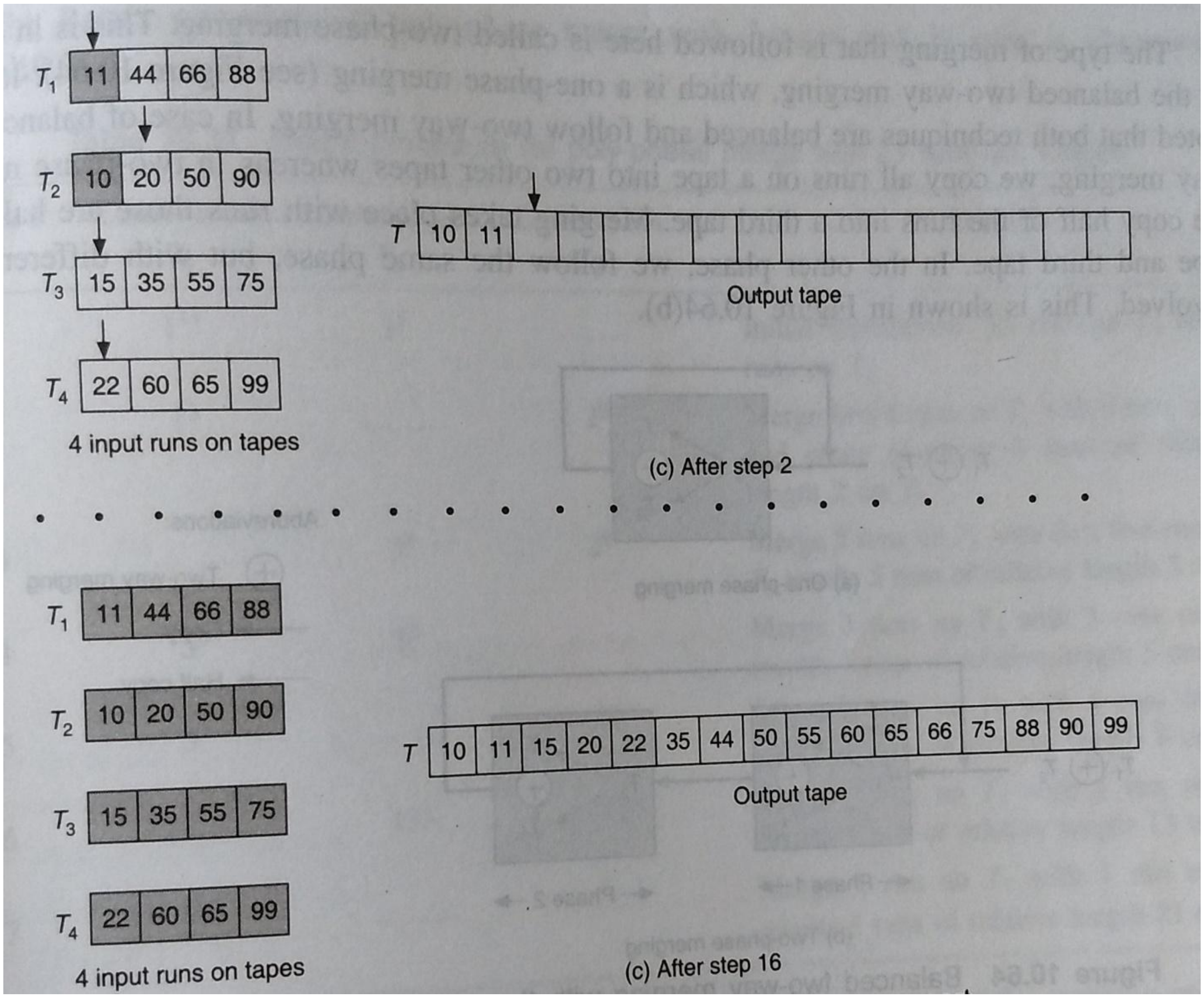
ii) Multi-way merge sort:

- It is an m- way merging where m-runs are combined into a single run. Here m > 2
- Consider m =3. Hence 3 tapes need to be maintained in a form of three linked list
- Consider initial input runs of each tape is already sorted.



(a) Initial status

(b) After step 1



(c) After step 2

(c) After step 16

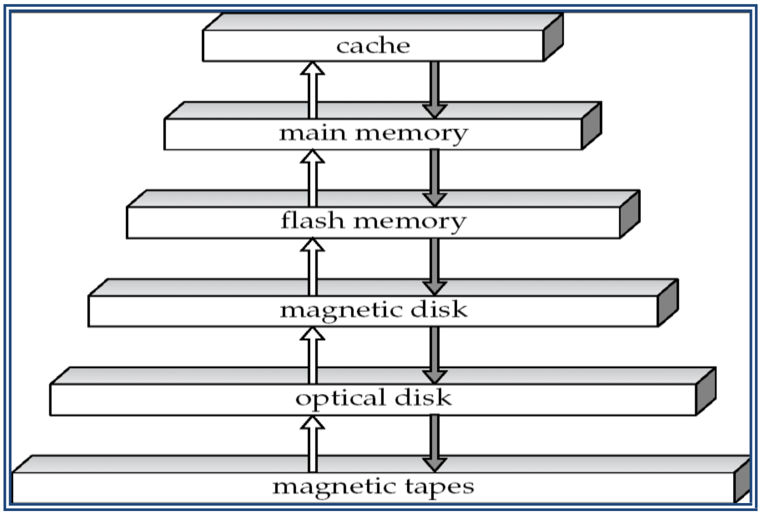
iii) **Poly- phase merge sort:**

Poly-phase merge sort combines both the 2-way merge sort and multi-way merge sort

EXTERNAL STORAGE DEVICES

Overview of physical storage media:

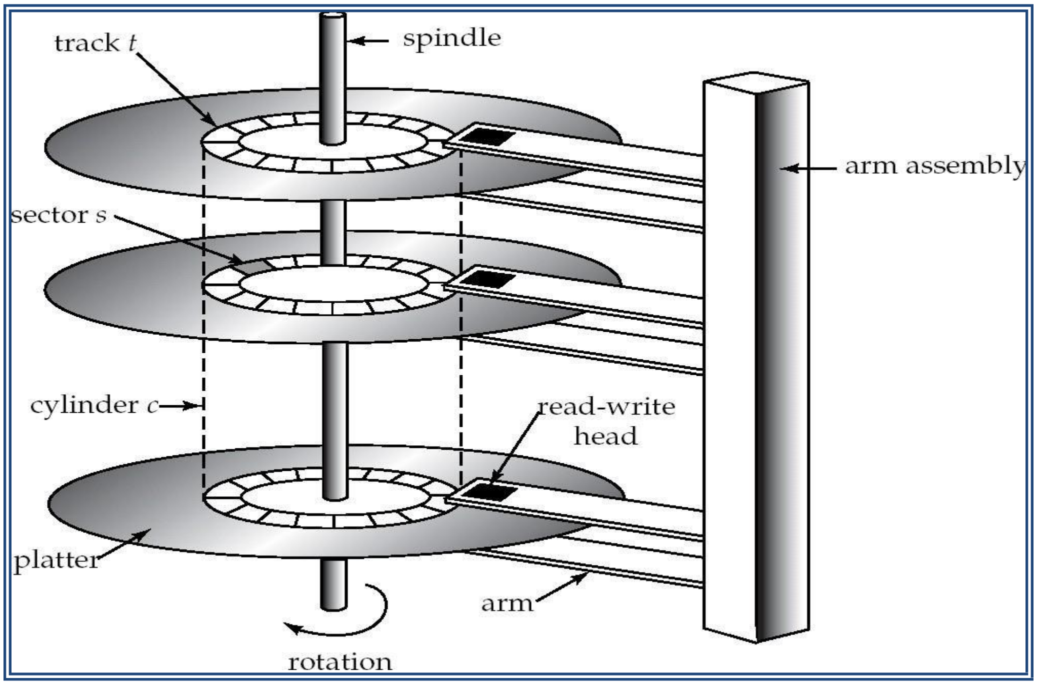
- Physical storage media is classified based on Speed with which data can be accessed, Cost per unit of data, Reliability



- **Primary storage:** Fastest media but volatile (cache, main memory).
- **Secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **Tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage
- **Cache** – fastest and most costly form of storage; It is a volatile(erasable) storage;
- **Main memory**:
  - **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.
  - fast access and size is too small
- **Flash memory**
  - Data survives power failure(Non-volatile)
  - Data can be written at a location only once.
  - Reads are as faster as main memory, but writes and erase is slower
- **Magnetic-disk**
  - Survives power failures(non-volatile) and but disk failure leads to failure of data
  - Data is stored on spinning disk, and read/written magnetically
  - It is a long-term storage and stores entire database.
  - **direct-access** – possible to read data on disk in any order, unlike magnetic tape

- **Optical storage**
  - It is non-volatile, data is read optically from a spinning disk using a laser
  - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
  - Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
  - Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
  - **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data
- **Tape storage**
  - It is non-volatile, used primarily for backup (to recover from disk failure), and for archival data
  - **sequential-access** – much slower than disk
  - very high capacity (40 to 300 GB tapes available)

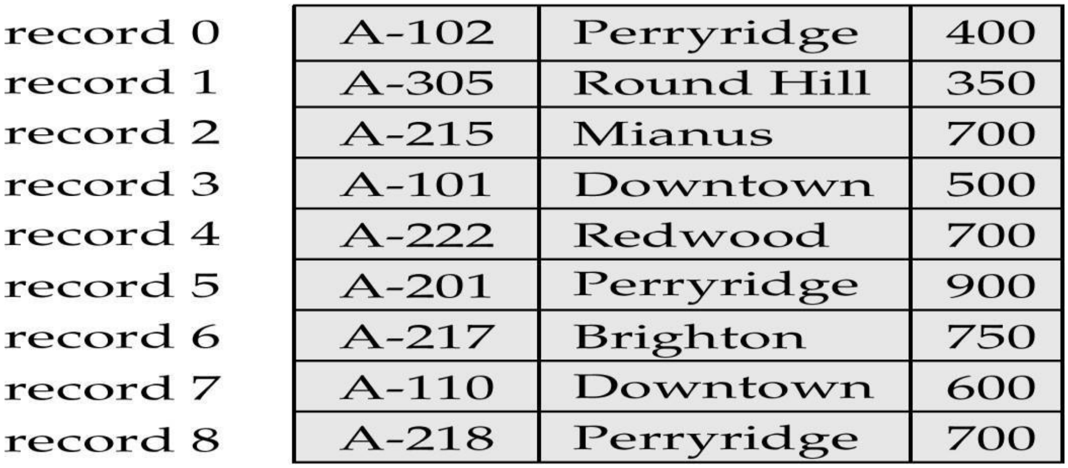
<u>Magnetic disks:</u>



- Platter is divided into circular tracks
  - Over 50K-1lakh tracks are present in one platter
- Each track is divided into sectors.
  - There are 500 to 1000 sectors per track
- To read/write a sector
  - Disk arm swings platter from center to end
  - Read-write head read/write the data in the sector
  - Cylinder *i* consists of *i*th track of all the platters

FILE ORGANIZATION

- The database is stored as a collection of *files*. Each file is a sequence of *records.* A record is a sequence of fields.
- Fields: Account_Number, Brach_Name and Balance.
- Collection of Records in a file is described by the following diagram



record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

FILE QUERIES (OR) QUERIES ON FILES:

- Files are like tables.
- We use a file system database as prefix in queries when we refer to object across the database
- Example: select * from hdfs . ‗ path of the file‘
- Here the file system database used is Hadoop Distributed File System(HDFS).

i) Plain Text Files:

- Comma – Separated Values(.csv)
- Tab – Separated Values(.tsv)
- Pipe – Separated Values(.psv)

Comma – Separated Values:

- In this, each column in a table is stored in a file, separated by comma.
- The file is stored with an extension .csv
- To display the entire columns of the csv file,
  - Select * from dfs. ‗ path of the csv file‘.
- To display only 1st and 2nd columns of csv file,
  - Select columns[0], columns[1] from dfs. ‗ path of csv file‘

Expr $0	Expr $1
1	Raj
2	Bala
3	Ravi

- To display the column name as meaningful, we can rename the column name by using following queries
    - Select columns[0] as rollno, columns[1] as name from dfs. _ path of csv file‘

Rollno	Name
11	Raj
5	Bala
15	Ravi

- To display the data in columns based on condition,
    - Select columns[0] as rollno, columns[1] as name from dfs. _ path of csv file‘ where columns[0] > 10

Tab-separated values and Pipe-separated values:

- Similar type of queries which are used in comma-separated values can be used for Tab-separated values and Pipe-separated values files

ii) Structured data file:

JSON file:

- JSON file is a file that stores simple data structures and objects.
- It contains data in a standard data interchange format which is text-based and human-readable. JSON files were originally based on a subset of JavaScript, but is considered a language-independent format, being supported by many different programming APIs.
- Google+, which uses JSON files for saving Profile data.
- JSON is commonly used in Ajax Web application programming. It is becoming increasingly popular as an alternative to XML.
- Mozilla Firefox saves bookmark backups using JSON files. The files are saved to the Firefox user profile directory within a folder called bookmarkbackups.

Queries on JSON files:

Select * from cp . _ path of json file‘ limit 5; // cp represents the class path

It displays the first five rows of the json file.

iii) Querying sequence files:

- Sequence files are flat files which stores binary key value pairs.
- In this, the values are stored in binary form.
- Select * from dfs. _ path of the sequence file‘ Limit 1

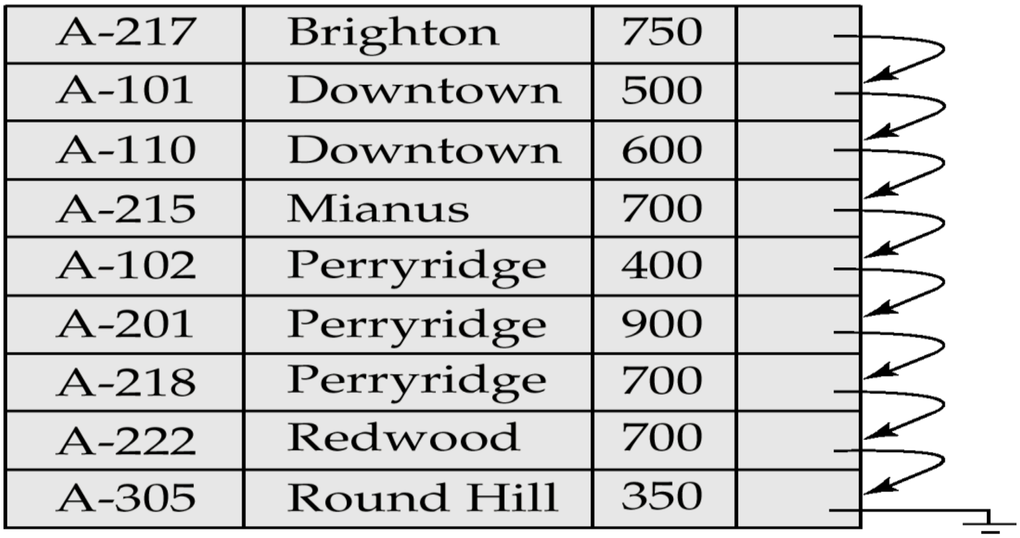
Binary key	Binary value
ff011	1001100

- The binary form of key value pair can be converted to the corresponding string
    - Select convert_from(Binary key , _UTF8‘), convert_from(Binary value, _UTF8‘) from dfs.‘path of sequence file‘ limit 1; // UTF represents Unicode Text Format

Expr $0	Expr $1
Key0	Value0

SEQUENTIAL FILE ORGANIZATION

- Sequential files have data records stored in a specific order.
- Sequential file can be accessed in serially.
- Suitable for applications that require sequential processing of the entire file
- The records in the file are ordered by a search-key.
- A key is defined to the data records to uniquely identify each data.
- For e.g., in a Bank application the customer is uniquely identified by bank account number.
- The following figure shows how the records are organized by a **links or pointer** chains sequentially.



A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

OPERATIONS PERFORMED IN SEQUENTIAL FILE ORGANIZATION

1. INSERTION OF RECORDS

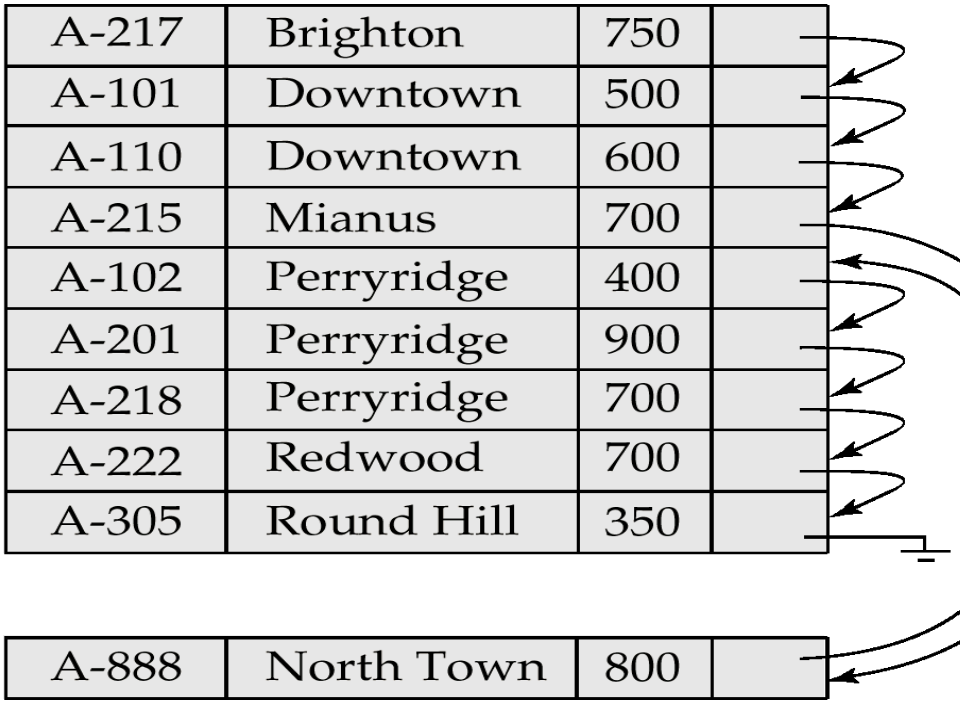
2. UPDATION OF RECORDS

3. SEARCHING OF RECORDS

4. DELETION OF RECORDS

INSERTION AND DELETION IN SEQUENTIAL FILE ORGANIZATION

- Insertion and Deletion – use pointer chains
- Insertion –locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order.



A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

A-888	North Town	800	

Fig. The role of Overflow block in sequential file organization

SEARCHING A RECORD IN A FILE

- It involves looking or finding a record sequentially (one by one) in file until finding the required record.

INDEXED SEQUENTIAL FILE

- Each record of a file has a key field which uniquely identifies that record.
- An index consists of keys and address (physical disc locations)
- An index sequential file is a sequential file (i.e. sorted into order of a key field) which has an index.
- A full index to a file is one in which there is an entry for every record.
- Indexed sequential file are important for applications where data needs to be accessed sequentially and randomly using the index.
- An indexed sequential file allows fast access to a specific record.
- E.g.:- A company may store details about its employees as an indexed sequential file. Sometimes the file is accessed, Sequentially, for e.g.:- When the whole file is processed to produce pay slips at the end of the month.
- Randomly, may be an employee changes address, or a female employee gets married and changes her surname.

Disadvantage of Sequential Files - The retrieval of a record from a sequential file, on average, requires access to half the records in the file, making such enquiries not only inefficient but very time consuming for large files. To improve the query responses time of a sequential file, a type of indexing technique can be added.

INDEXING

- An index is used to quickly locate and access the data in a database table.
    - E.g., author catalog in library
- Index files are typically much smaller than the original file
- Data is stored in the form of records. Every record has a key field, by which a record can be uniquely identified.
- The general format of an index file is

Search-key value	Pointer to actual record on disk

- ***Search Key***–It is an attribute to set of attributes which is used to look up records in a file.

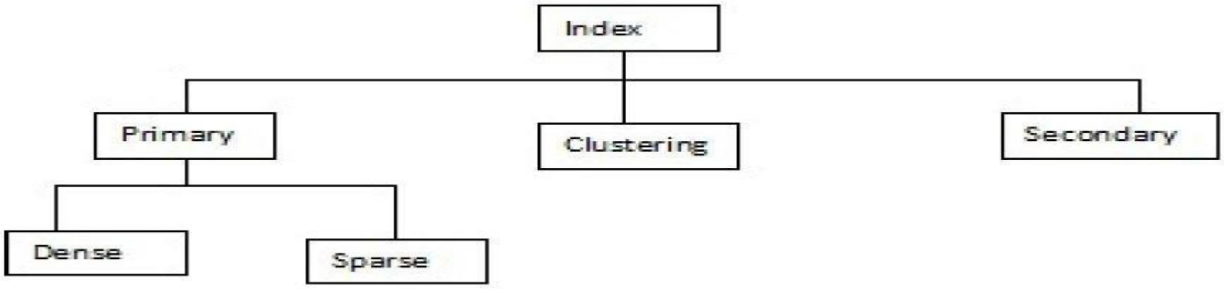
Example:

- Consider, there are 100 records each of 10 bytes long. If we need to access 60th record, ***we need to access*** (59 records*10bytes)=> ***590 bytes*** before accessing 60th record, ***If indexing is not used***
- Index file contain only ***id and pointer*** whose size is assumed to be **2 bytes** for each record
- Since, index file is much smaller than the actual file,
- If we need to access 60th record, ***We need to access*** (59 records*2bytes)=> ***118 bytes*** before accessing 60th record, ***If indexing is used.***
- Hence the time of searching is reduced.

Two basic types of indices:

- **Ordered indices:** Search keys are stored in sorted order
- **Hash indices:** Search keys are distributed uniformly across –buckets‖ using a –hash function‖.

TYPES OF ORDERED INDICES:

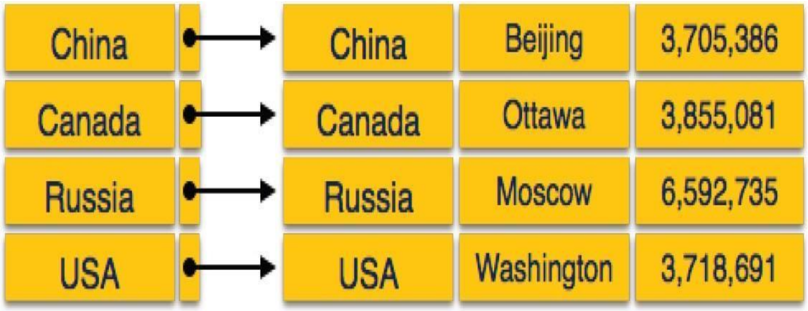


i) PRIMARY INDEX

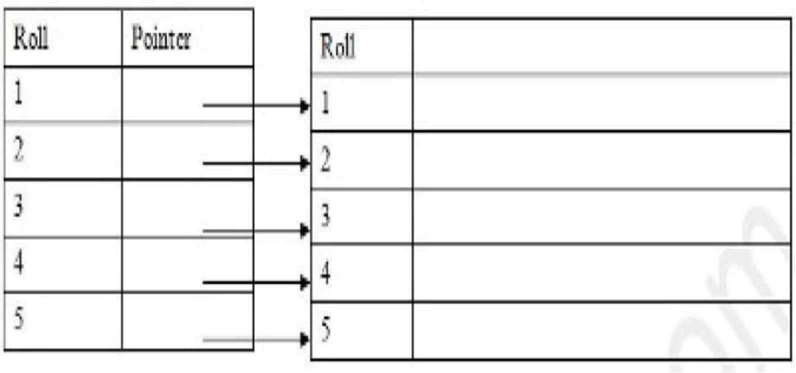
- Entries in the index table have a one-to-one relationship with the main table.
- **Primary index** can be of two types:
    - **Dense index:** each and every record in the main table has an entry in the index table.
    - **Sparse index:** Only some of the record in main table will have an entry in index table

a) DENSE INDEX:

index table File with set of records

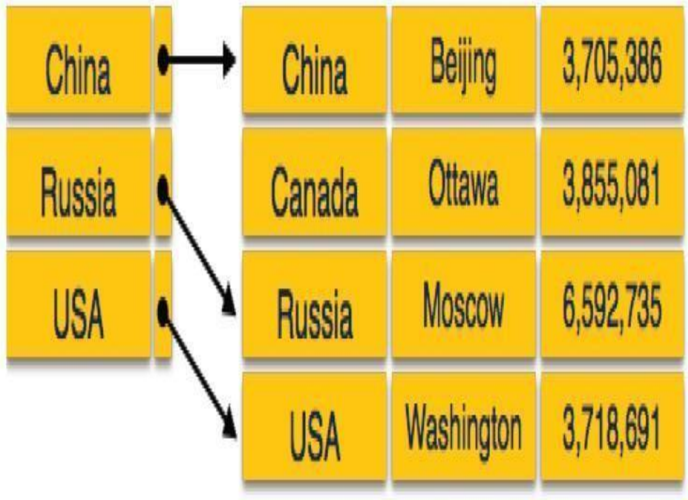


index table File with set of records

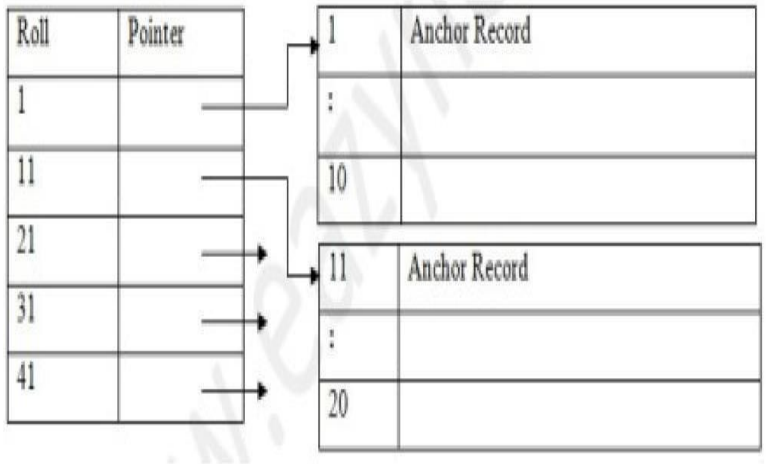


b) SPARSE INDEX:

index table File with set of records

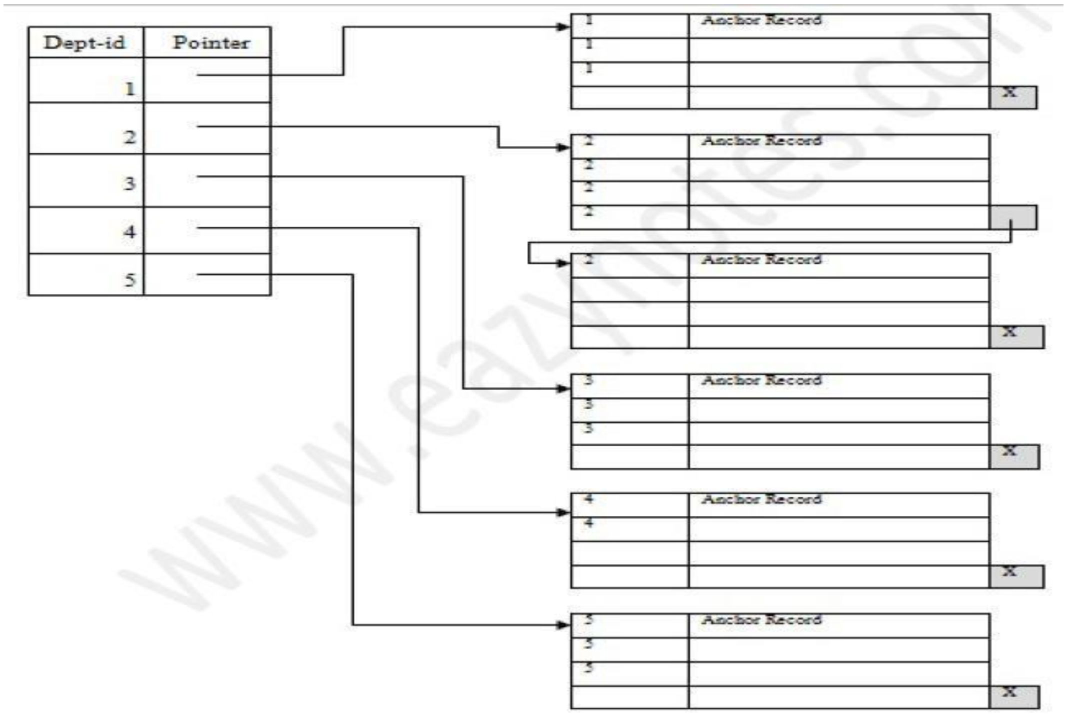


index table **File with set of records**



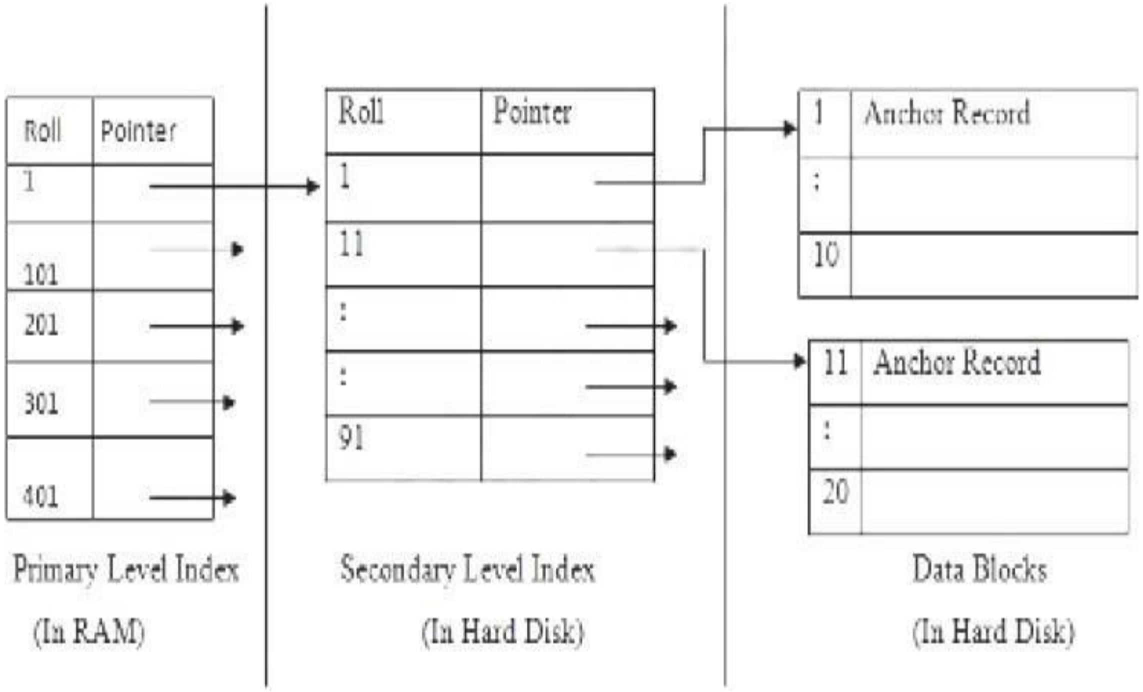
ii) CLUSTERING INDEX:

- Grouping of related (or) same type of record is performed in clustering.
- Several Employees of same department can be grouped into cluster
- Several Students of same department can be grouped into cluster
- Based on department – id, the particular records in the cluster can be accessed



iii) SECONDARY INDICES:

- Secondary level index are used, if the data blocks doesn‘t fit to main memory
- Two level of index is used to access the records.
- The data blocks are stored in hard disk.



UNIT – 5 TWO MARKS

1. What are Jagged tables? (or) Comment on Jagged tables.

- In jagged table, Each row in a table has varying number of elements.
- Given a linear index value, the access array is searched to find the appropriate data in a table.

2. What are the components of hash table?

- **Bucket array =>** It is an array where elements are stored
- **Hash Function =>** It is a function/algorithm which is used for storing elements in a hash table.

3. Compare sequential and index organization.

Sequential organization	Index organization
Data is entered in entry sequential order	Data is entered in key sequential order
Data is not in sorted order	Data is in sorted order based on key
Access of data is slower	Access of data is fast
Duplicate data is allowed	Duplicate data is not allowed
Key is not available	Key is available
It is also called as QSAM(Queue Sequential Access Method)	It is also called as VSAM(Virtual Storage Access Method)

4. Write the difference between internal and external sorting.

- When data which is to be sorted fits in main memory, then that sorting is called as internal sorting. Example: Sorting of small set of values.
- When sorting is performed on massive amount of data, which is stored on external storage devices, it is called external sorting.Example: Sorting of Geographical Information System(GIS), Biometrics data.

5. Classify external sorting.

- Some of the external sorting algorithms are: Balanced two-way merge, Multi-way merging, Poly-phase merge sort, Oscillating sorting, External radix sort.

6. What is hashing?

- **Hashing** is an effective technique to calculate the direct location of a data record on the disk using hash function.
- Hash function is denoted by H(x)

7. What is hash table? (Or) what is Hash map ? (or) Why hash table have been used?

- Hash table is a Data structure where the data elements are stored, searched, deleted based on the keys generated for each element, which is obtained from a hashing function.
- Hash table has an array of buckets which stores data.

8. What is index technique?

- An index is used to quickly locate and access the data in a database table.
    - E.g., author catalog in library
- Index files are typically much smaller than the original file
- The general format of an index file is

Search-key value	Pointer to actual record on disk

9. Define symbol table. (or) what is the use of symbol table? (or) List the properties of symbol table.

- Symbol table is a data structure used by compilers in order to store information about the occurrence of identifiers, function names etc. (i.e) It stores the memory address where the value of identifier present
- It is used to verify that the used identifiers are declared.
- It is used to verify that the expression is semantically correct – Perform typechecking.

10. What do you know about collision in hashing? (or) Explain hash collision.

- When two or more data elements in the data set *U*, maps to the same location in the hash table, then it is called as hash collision.

11. What is a need of inverted table?

- Inverted table is used to avoid maintaining multiple tables for different purpose. Instead, a single inverted table is used.
- To create inverted table, each of the column in original table is ordered. Then, the corresponding index of each value in original table is written in the inverted table.

12. Name some of the external storage devices.

- Magnetic disk, Magnetic tape, Hard disk, Flash drive, CD/DVD, Solid state memory card

13. What are the indexing techniques available for files?

- Indexed Sequential file organization, Hash indices, B+ tree index, Trie tree index, Denseindex, Sparse index, Clustered index, Secondary index, multilevel index.

14. What is sequential file access?

- In sequential file, Records are stored in sequential order. The records in the file are ordered by a search-key.
- Each record can be accessed only in a sequential manner.
- It is suitable for applications that require sequential processing of the entire file

15. Define dense and sparse index.

- **Dense index:** In dense index, each and every record in the main table has an entry in the index table.
- **Sparse index:** In sparse index, Only some of the record in main table will have an entry in index table

16. What are the methods available for accessing a symbol table?

- Insert( ) operation is used to add information into the symbol table
- Lookup( ) operation is used to search a name in symbol table, to determine whether the symbol exists in a table
- Scope( ) operation is used to identify whether the symbol is in local or global scope

17. Write the main advantages of indexed sequential file. (or) Write about ISAM(Indexed Sequential Access Method).

- Indexed sequential file are important for applications where data needs to be accessed sequentially and randomly using the index.
- Example: Company stores employee details in indexed sequential file. Here, the entire file is accessed while crediting salary for all employees. File is accessed randomly when a particular employee changes his address.

18. What are the four types of queries?

- Querying plain text files
- Querying sequential files
- Querying JSON files
- Querying directories

19. What is meant by a table?

- Table is a data structure used to store the records.
- It helps in efficient retrieval of records.
- Access table stores the location of all records in a file. The method of accessing the record in a file using an access table is call table lookup, which is independent on number of records.

20. What are the various collision resolution techniques in hashing?

i) Open hashing

a. Chaining

ii) Closed hashing

a. Linear Probing

b. Quadratic Probing

21. What is meant by dynamic hashing?

- In dynamic hashing, ***Buckets are allocated on-demand dynamically.***
- It specifies a binary value for each bucket.