

LECTURE NOTES  
ON  
DATA AND FILE STRUCTURE  
B. Tech. 3<sup>rd</sup> Semester  
Computer Science & Engineering  
and  
Information Technology

Prepared by

Dr. Rakesh Mohanty  
Dr. Manas Ranjan Kabat  
Mr. Sujaya Kumar Sathua



VEER SURENDRA SAI UNIVERSITY OF TECHNOLOGY, BURLA  
SAMBALPUR, ODISHA, INDIA – 768018

**3<sup>RD</sup> SEMESTER B.Tech.(CSE, IT)**  
**BCS-202 DATA AND FILE STRUCTURE – ( 3-0-0 )Cr.-3**  
**Proposed Lecture Plan**

**Lecture 1 :** Motivation, Objective of studying the subject, overview of Syllabus

**Lecture 2 : Module I :** Introduction to Data & file structures.

**Lecture 3 :** Linear data Structures – Linked list and applications

**Lecture 4 :** Stack and Queue

**Lecture 5 : Module II :** Introduction to Non- Linear data structures

**Lecture 6 :** General Trees , Binary Trees, Conversion of general tree to binary

**Lecture 7 :** Binary Search Tree

**Lecture 8 :** Red-Black trees

**Lecture 9 :** Multi linked structures

**Lecture 10 :** Heaps

**Lecture 11:** Spanning Trees, Application of trees

**Lecture 12 : Module III** Introduction to Sorting

**Lecture 13, 14 :** Growth of function , 'O' notation, Complexity of algorithms,

**Lecture 15 :** Internal sorting, Insertion sorting, Selection Sort

**Lecture 16 :** Bubble Sort, Quick sort, Heap sort

**Lecture 17 :** Radix sort, External sort, Multi way merge

**Lecture 18 : Module IV :** Introduction to Searching, Sequential Search, Binary Search

**Lecture 19 :** Search trees traversal

**Lecture 20 :** Threaded Binary search trees

**Lecture 21** : AVL Tree – concept and construction

**Lecture 22** : Balancing AVL trees - RR, LL, LR and RL Rotations

**Lecture 23** : **Module V** : Introduction to Hashing

**Lecture 24** : Hashing techniques, Hash function

**Lecture 25** : Address calculation techniques- common hashing functions

**Lecture 26** : Collision resolution

**Lecture 27** : Linear probing, quadratic probing

**Lecture 28** : Double hashing

**Lecture 29** : Bucket addressing

**Lecture 30** : **Module VI**- Introduction to file Structures

**Lecture 31** : External storage devices

**Lecture 32** : Records - Concepts and organization

**Lecture 33** : Sequential file – structures and processing

**Lecture 34** : Indexed sequential files – structures and processing

**Lecture 35** : Direct files

**Lecture 36** : Multi Key access

## INTRODUCTION

**DATA STRUCTURE:** -Structural representation of data items in primary memory to do storage & retrieval operations efficiently.

--FILE STRUCTURE: Representation of items in secondary memory.

While designing data structure following perspectives to be looked after.

- i. Application(user) level: Way of modeling real-life data in specific context.
- ii. Abstract(logical) level: Abstract collection of elements & operations.
- iii. Implementation level: Representation of structure in programming language.

Data structures are needed to solve real-world problems. But while choosing implementations for it, its necessary to recognize the efficiency in terms of TIME and SPACE.

### TYPES:

- i. Simple: built from primitive data types like int, char & Boolean.  
eg: Array & Structure
- ii. Compound: Combined in various ways to form complex structures.  
1:Linear: Elements share adjacency relationship& form a sequence.  
Eg: Stack, Queue , Linked List  
  
2: Non-Linear: Are multi-level data structure. eg: Tree, Graph.

### ABSTRACT DATA TYPE :

- Specifies the logical properties of data type or data structure.
- Refers to the mathematical concept that governs them.
- They are not concerned with the implementation details like space and time efficiency.
- They are defined by 3 components called Triple =(D,F,A)
  - D=Set of domain
  - F=Set of function
  - A=Set of axioms / rules

## **LINKED LIST:**

- A dynamic data structure.
- Linear collection of data items.
- Direction is associated with it.
- Logical link exists b/w items. Pointers acts as the logical link.
- Consists of nodes that has two fields.
  - Data field : info of the element.
  - Next field: next pointer containing the address of next node.

## **TYPES OF LINKED LIST:**

i. Singly or chain: Single link b/w items.



ii. Doubly: There are two links, forward and backward link.



iii. Circular: The last node is again linked to the first node. These can be singly circular & doubly circular list.

## **ADVANTAGES:**

- Linked list use dynamic memory allocation thus allocating memory when program is initialised. List can grow and shrink as needed. Arrays follow static memory allocation .Hence there is wastage of space when less elements are declared. There is possibility of overflow too bcoz of fixed amount of storage.
- Nodes are stored incontinuously thus insertion and deletion operations are easily implemented.
- Linear data structures like stack and queues are easily implemented using linked list.

## **DISADVANTAGES:**

- Wastage of memory as pointers require extra storage.
- Nodes are incontinuously stored thereby increasing time required to access individual elements. To access nth item arrays need a single operation while linked list need to pass through (n-1) items.
- Nodes must be read in order from beginning as they have inherent sequential access.

- Reverse traversing is difficult especially in singly linked list. Memory is wasted for allocating space for back pointers in doubly linked list.

### DEFINING LINKED LIST:

```
struct node {  
int info;  
  
    struct node *next; \\next field. An eg of self referencetial structure.(#)  
  
} *ptr;
```

(#)Self Referencetial structure: A structure that is referencing to another structure of same type. Here “next” is pointing to structure of type “node”.

-ptr is a pointer of type node. To access info n next the syntax is: ptr->info; ptr->next;

### OPERATIONS ON SINGLY LINKED LIST:

- i. Searching
- ii. Insertion
- iii. Deletion
- iv. Traversal
- v. Reversal
- vi. Splitting
- vii. Concatenation

Some operations:

a: Insertion :

```
void push(struct node** headref, int data)          -----(1)  
{  
  
    struct node* newnode = malloc(sizeof(struct node));  
  
    newnode->data= data;  
  
    newnode->next= *headref;  
  
    *headref = newnode;  
  
}
```

(1) : headref is a pointer to a pointer of type struct node. Such passing of pointer to pointer is called Reference pointer. Such declarations are similar to declarations of call by reference. When pointers are passed to functions ,the function works with the original copy of the variable.

i. Insertion at head:

```
struct node* head=NULL;
for(int i=1; i<6;i++)

{ push(&head,i); \\ push is called here. Data pushed is 1.'&' used coz references are
passed in function arguments.
```

```
    }
return(head);
}
# :o\p: 5 4 3 2 1

# :Items appear in reverse order.(demerit)
```

ii. Insertion at tail:

```
struct node* add1()
{ struct node* head=NULL;
struct node* tail;
push(&head,1);
tail = head;
for(int i=2 ;i<6; i++)
{ push(&(tail->next),i);
tail= tail->next;
}
return(head);
}
```

```
# : o\p: 1 2 3 4 5
```

**b. Traversal:**

```
int count( struct node* p)
```

```
{int count =0;
```

```
struct node* q;
```

```
current = q;
```

```

while(q->next != NULL)
{
    q=q->next;
    count++;
}
return(count);

```

c. Searching:

```

struct node* search( struct node* list, int x)
{
    struct node* p;
    for(p= list; p ->next != NULL; p= p->next )
    {
        if(p->data==x)
            return(p);
        return(NULL);
    }
}

```

IMPLEMENTATION OF LISTS:

i : Array implementation:

```

#define NUMNODES 100

structnodetype
{
    int info ,next ;
};

structnodetype node[NUMNODES];

```

# :100 nodes are declared as an array node. Pointer to a node is represented by an array index. Thus pointer is an integer b/w 0 & NUMNODES-1 . NULL pointer is represented by -1. node[p] is used to reference node(p) , info(p) is referenced by node[p].info & next by node[p].next.



ii : Dynamic Implementation :

This is the same as codes written under defining of linked lists. Using malloc() and free() there is the capability of dynamically allocating & freeing variable. It is identical to array implementation except that the next field is a pointer rather than an integer.

NOTE : Major demerit of dynamic implementation is that it may be more time consuming to call upon the system to allocate & free storage than to manipulate a programmer- managed list. Major advantage is that a set of nodes is not reserved in advance for use.

# **SORTING**

## **Introduction**

- Sorting is the process of arranging items in a certain sequence or in different sets .
- The main purpose of sorting information is to optimize its usefulness for a specific tasks.
- Sorting is one of the most extensively researched subject because of the need to speed up the operations on thousands or millions of records during a search operation.

## **Types of Sorting :**

- **Internal Sorting**

An **internal sort** is any data sorting process that takes place entirely within the main memory of a computer.

This is possible whenever the data to be sorted is small enough to all be held in the main memory.

For sorting larger datasets, it may be necessary to hold only a chunk of data in memory at a time, since it won't all fit.

The rest of the data is normally held on some larger, but slower medium, like a hard-disk.

Any reading or writing of data to and from this slower media can slow the sorting process considerably

- **External Sorting**

Many important sorting applications involve processing very large files, much too large to fit into the primary memory of any computer.

Methods appropriate for such applications are called external methods, since they involve a large amount of processing external to the central processing unit.

There are two major factors which make external algorithms quite different: f

First, the cost of accessing an item is orders of magnitude greater than any bookkeeping or calculating costs. f

Second, over and above with this higher cost, there are severe restrictions on access, depending on the external storage medium used: for example, items on a magnetic tape can be accessed only in a sequential manner

## **Well Known Sorting methods :**

->Insertion sort, Merge sort, Bubble sort, Selection sort, Heap sort, Quick sort

### **INSERTION SORT**

Insertion sort is the simple sorting algorithm which sorts the array by shifting elements one by one.

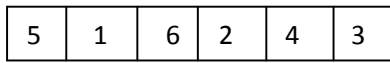
->OFFLINE sorting-This is the type of sorting in which whole input sequence is known. The number of inputs is fixed in offline sorting.

->ONLINE sorting-This is the type of sorting in which current input sequence is known and future input sequence is unknown i.e in online sort number inputs may increase.

#### ***INSERTION SORT ALGORITHM:***

```
int a[6]={5,1,6,2,4,3};
int i,j,key;
for(i=1;i<6;i++)
{
    key=a[i];
    j=i-1;
    while(j>=0 && key<a[j])
    {
        a[j+1]=a[j];
        j-- ;
    }
    a[j+1]=key;
}
```

**HOW INSERTION SORT WORKS:** Lets consider this array



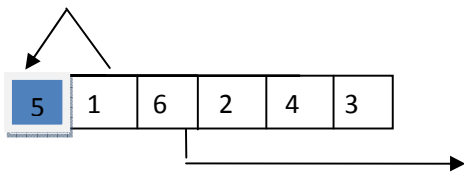
In insertion always start with 2nd element as key sort we

As we can see here, in insertion sort, we pick up a key,

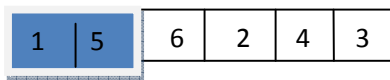
and

compares it with elements before of it.

In first step, there is one comparison.



(partially sorted list)

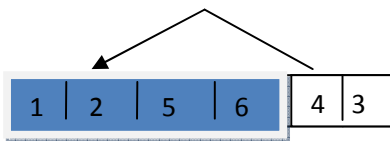


1 is compared with 5 and is inserted before 5.

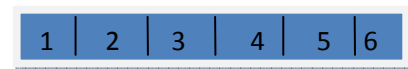
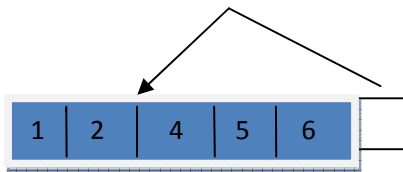
6 is greater than 1 and 5, so there is no swap.



2 is smaller than 5 and 6 but greater than 1, so it is inserted after 1.



And this goes on...



(complete sorted list)

In insertion sort there is a single pass, but there are many steps.

Number of steps utmost in Insertion sort is:

$1+2+3+\dots+n$ , where n is number of elements in the list.

$=((n-1)n)/2 = O(n^2)$  (TIME COMPLEXITY)

***Pseudo code for insertion sort:***

Input: n items

Output: sorted items in ascending order.

->compare item1 and item2

if (item1>item2 ),then SWAP

else , no swapping

A partially sorted list is generated

->Then scan next item and compare with item1 and item2 and then continue .

***ADVANTAGES OF INSERTION SORT:***

- \* Simple implementation.
- \* Efficient for small data sets.
- \* Stable i.e does not change the relative order of elements with same values.
- \* Online i.e can sort a list as it receives it.

***LIMITATIONS OF INSERTION SORT:***

- \* The insertion sort repeatedly scans the list of items, so it takes more time.
- \*With  $n^2$  steps required for every  $n$  elements to be sorted , the insertion sort does not deal well with a huge list. Therefore insertion sort is particularly useful when sorting a list of few items.

***REAL LIFE APPLICATIONS OF INSERTION SORT:***

- \* Insertion sort can used to sort phone numbers of the customers of a particular company.
- \* It can used to sort bank account numbers of the people visiting a particular bank.

## MERGE SORT

Merge sort is a recursive algorithm that continually splits a list. In merge sort parallel comparisons between the elements is done. It is based on "divide and conquer" paradigm.

Algorithm for merge sort:

```
void mergesort(int a[],int lower,int upper)
{
    int mid;
    if(upper >lower)
    {
        mid=(lower+upper)/2;
        mergesort(a,lower,mid);
        mergesort(a,mid+1,upper);
        merge(a,lower,mid,mid+1,upper);
    }
}

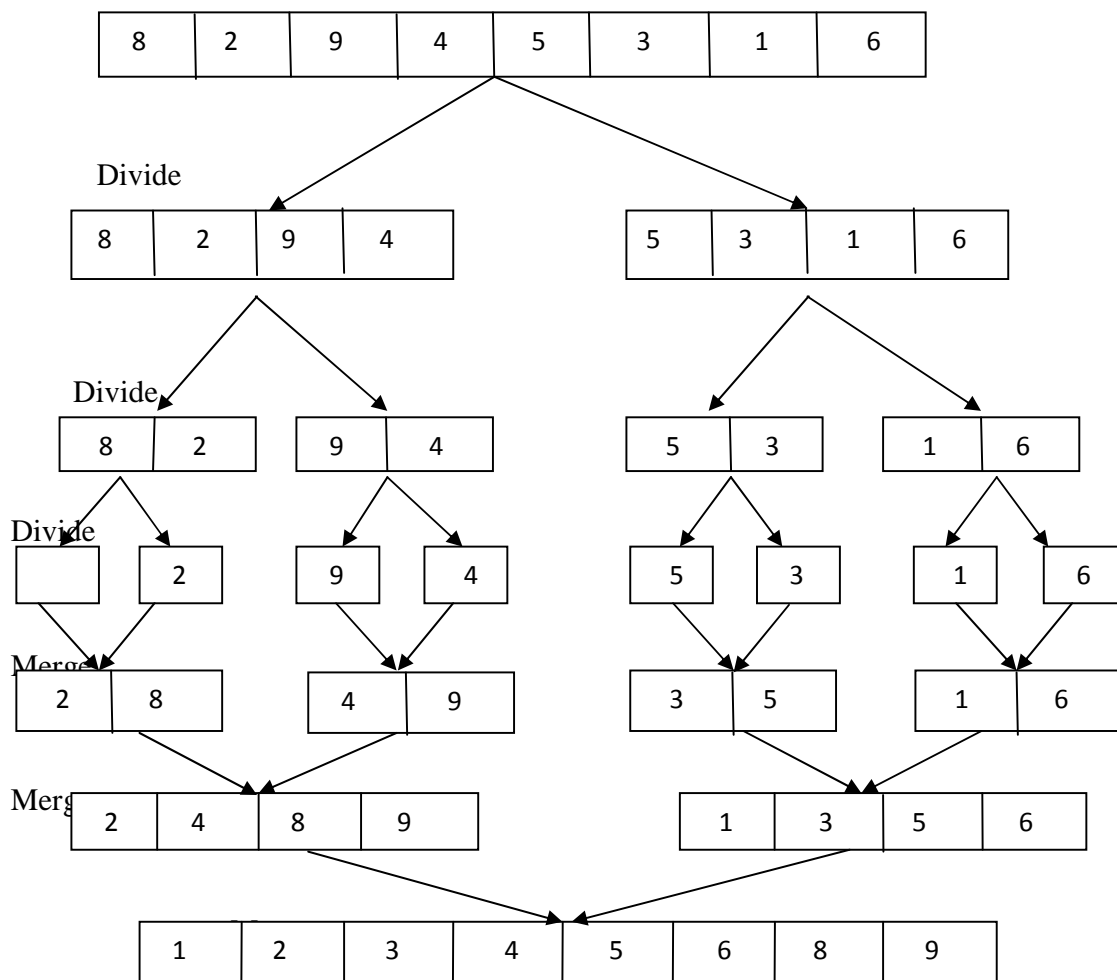
void merge(int a[],int lower1,int upper1,int lower2,int upper2)
{
    int p,q,j,n;
    int d[100];
    p=lower1; q=lower2; n=0;
    while((p<=upper1)&& (q<=upper2))
    {
        d[n++]=a[p]<a[q]?a[p++]:a[q++];
    }
}
```

```

while(p<=upper1)
    d[n++]=a[p++];
while(q<=upper2)
    d[n++]=a[q++];
for(q=lower1,n=0;q<=upper1;q++,n++)
    a[q]=d[n];
for(q=lower2,j=0;q<=upper2;q++,n++)
    a[q]=d[j];
}

```

Illustration



\*Divide it in two at the midpoint.



\*Conquer each side in turn by recursive sorting.

\*Merge two halves together.

#### TIME COMPLEXITY:

There are total  $\log_2 n$  passes in merge sort and in each pass there are  $n$  comparisons atmost.

Therefore total number of comparisons= $O(n * \log_2 n)$ .

In 3 way merge sorting time complexity is  $O(n \log_3 n)$ .

\* $k$  way merge is possible in merge sort where  $k$  is utmost  $n/2$  i.e  $k < n/2$ .

#### ADVANTAGES OF MERGE SORT:

\* It is always fast and less time consuming in sorting small data.

\* Even in worst case its runtime is  $O(n \log n)$

\*It is stable i.e it maintains order of elements having same value.

#### LIMITATIONS OF MERGE SORT:

\* It uses a lot of memory.

\* It uses extra space proportional to  $n$ .

\* It slows down when attempting to sort very large data.

#### APPLICATIONS OF MERGE SORT:

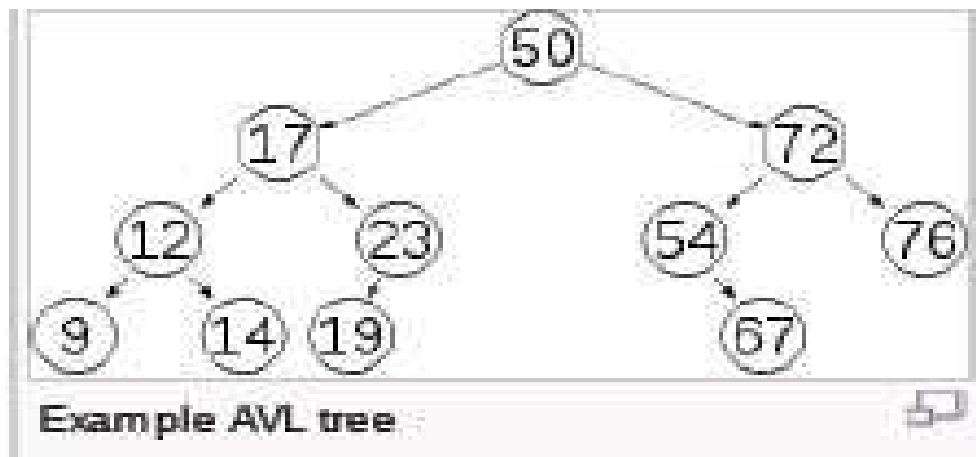
\* Merge sort operation is useful in online sorting where list to be sorted is received a piece at a time instead of all at the beginning, for example sorting of bank account numbers.



# Binary Search Trees and AVL Trees

## Introduction

- Binary search trees are an excellent data structure to implement associative arrays, maps, sets, and similar interfaces.
- The main difficulty, as discussed in last lecture, is that they are efficient only when they are balanced.
- Straightforward sequences of insertions can lead to highly unbalanced trees with poor asymptotic complexity and unacceptable practical efficiency.
- The solution is to dynamically rebalance the search tree during insert or search operations.
- We have to be careful not to destroy the ordering invariant of the tree while we rebalance.
- Because of the importance of binary search trees, researchers have developed many different algorithms for keeping trees in balance, such as AVL trees, red/black trees, splay trees, or randomized binary search trees.
- 
- In this lecture we discuss AVL trees, which is a simple and efficient data structure to maintain balance.
- It is named after its inventors, G.M. Adelson-Velskii and E.M. Landis, who described it in 1962.



## Height Invariant

### Ordering Invariant.

- At any node with key  $k$  in a binary search tree, all keys of the elements in the left subtree are strictly less than  $k$ , while all keys of the elements in the right subtree are strictly greater than  $k$ .

- To describe AVL trees we need the concept of tree height, which we define as the maximal length of a path from the root to a leaf. So the empty tree has height 0, the tree with one node has height 1, a balanced tree with three nodes has height 2.
- If we add one more node to this last tree it will have height 3. Alternatively, we can define it recursively by saying that the empty tree has height 0, and the height of any node is one greater than the maximal height of its two children.
- 
- AVL trees maintain a height invariant (also sometimes called a balance invariant).
- 

### Height Invariant.

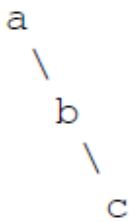
- At any node in the tree, the heights of the left and right subtrees differs by at most 1.

### Rotations: How they work

#### Left Rotation (LL)

Imagine we have this situation:

Figure 1-1



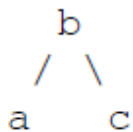
To fix this, we must perform a left rotation, rooted at A. This is done in the following steps:  
 b becomes the new root.

a takes ownership of b's left child as its right child, or in this case, null.

b takes ownership of a as its left child.

The tree now looks like this:

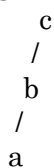
Figure 1-2



#### Right Rotation (RR)

A right rotation is a mirror of the left rotation operation described above. Imagine we have this situation:

Figure 1-3



To fix this, we will perform a single right rotation, rooted at C. This is done in the following steps:

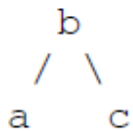
b becomes the new root.

c takes ownership of b's right child, as its left child. In this case, that value is null.

b takes ownership of c, as it's right child.

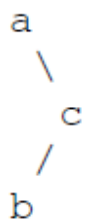
The resulting tree:

Figure 1-4



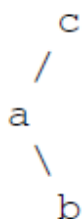
## Double Rotations (left-right)

Figure 1-6



Our initial reaction here is to do a single left rotation. Let's try that

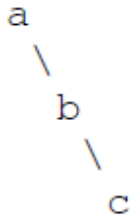
Figure 1-7



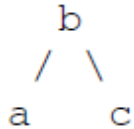
Our left rotation has completed, and we're stuck in the same situation. If we were to do a single right rotation in this situation, we would be right back where we started. What's causing this? The answer is that this is a result of the right subtree having a negative balance. In other words, because the right subtree was left heavy, our rotation was not sufficient. What can we do? The answer is to perform a right rotation on the right subtree. Read that again. We will perform a right rotation on the *right subtree*. We are not rotating on our current root. We are rotating on our right child. Think of our right subtree, isolated from our main tree, and perform a right rotation on it:

After performing a rotation on our right subtree, we have prepared our root to be rotated left.

Here is our tree now:

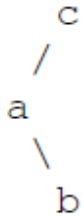


Looks like we're ready for a left rotation.

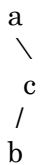


## Double Rotations (right-left)

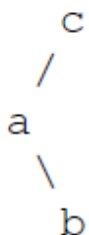
A double right rotation, or right-left rotation, or simply RL, is a rotation that must be performed when attempting to balance a tree which has a left subtree, that is right heavy. This is a mirror operation of what was illustrated in the section on Left-Right Rotations, or double left rotations. Let's look at an example of a situation where we need to perform a Right-Left rotation.



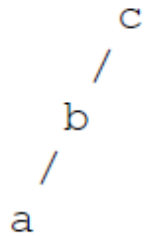
In this situation, we have a tree that is unbalanced. The left subtree has a height of 2, and the right subtree has a height of 0. This makes the balance factor of our root node, c, equal to -2. What do we do? Some kind of right rotation is clearly necessary, but a single right rotation will not solve our problem.



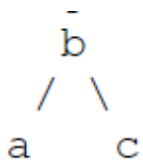
Looks like that didn't work. Now we have a tree that has a balance of 2. It would appear that we did not accomplish much. That is true. What do we do? Well, let's go back to the original tree, before we did our pointless right rotation:



The reason our right rotation did not work, is because the left subtree, or 'a', has a positive balance factor, and is thus right heavy. Performing a right rotation on a tree that has a left subtree that is right heavy will result in the problem we just witnessed. What do we do? The answer is to make our left subtree left-heavy. We do this by performing a left rotation on our left subtree. Doing so leaves us with this situation:



This is a tree which can now be balanced using a single right rotation. We can now perform our right rotation rooted at C.



Increasing the speed by minimizing the height difference is the main of AVL tree.

Operations like insertions, deletions etc can be done in time of  $O(\log n)$ , even in the worst case.

e.g. In a complete balanced tree, the left and right subtrees of any node would have same height.

Height difference is 0.

Suppose  $H_l$  is the height of left sub tree and  $H_r$  is the height of right sub tree, then following properties must be satisfied for the tree to be an AVL tree:

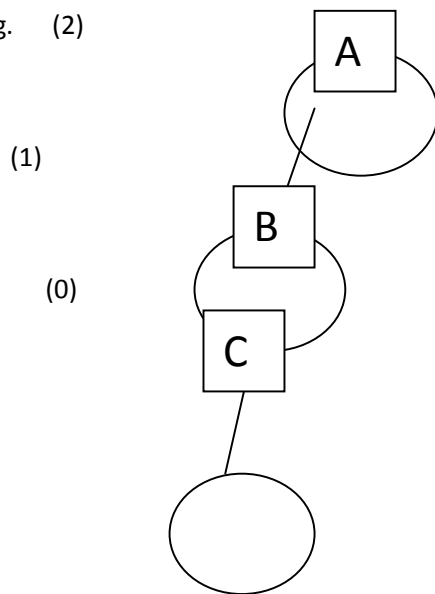
1. It must be a BST.

2. Height of left sub-tree - Height of right sub-tree  $\leq 1$ .

i.e.  $|H_l - H_r| \leq 1$

$$H_l - H_r = \{-1, 0, 1\}$$

e.g. (2)



Balancing factor =  $H_l - H_r$ .

In the above diagram, balancing factor for root node is 2, so, it is not an AVL tree. In such cases, the tree can be balanced by rotations.

There are 4 type of rotations:

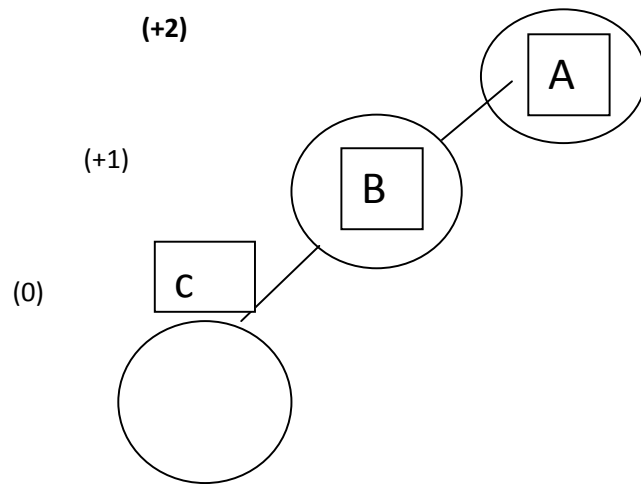
1. LL Rotation (left-left)

2. RR Rotation (right-right)

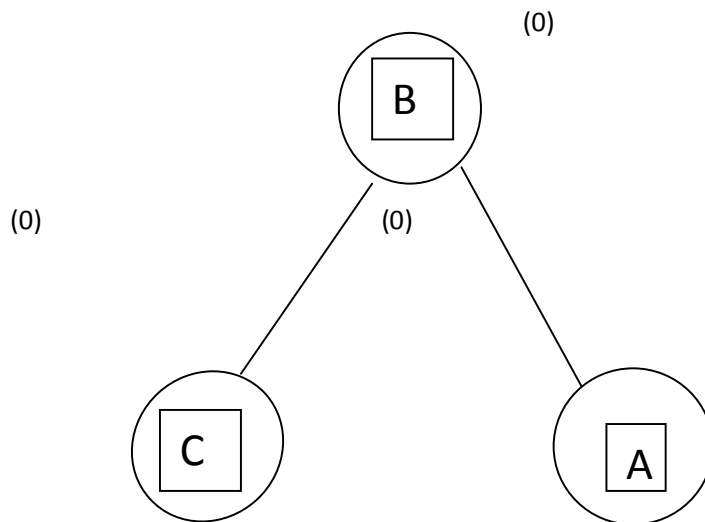
3. LR Rotation (left-right)

4. RL Rotation (right-left)

**LL Rotation:**

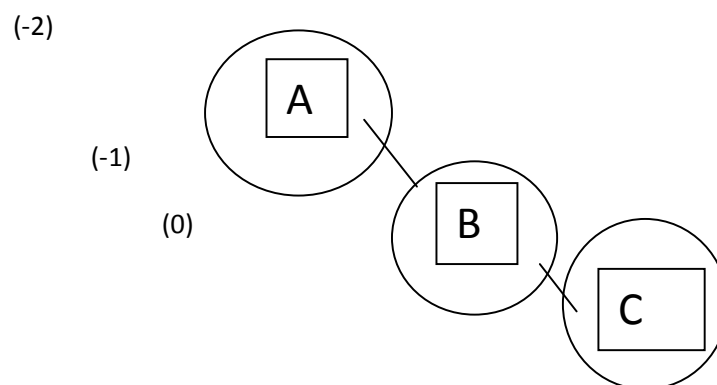


Balance factor of A is +2. Applying LL rotation, we get,

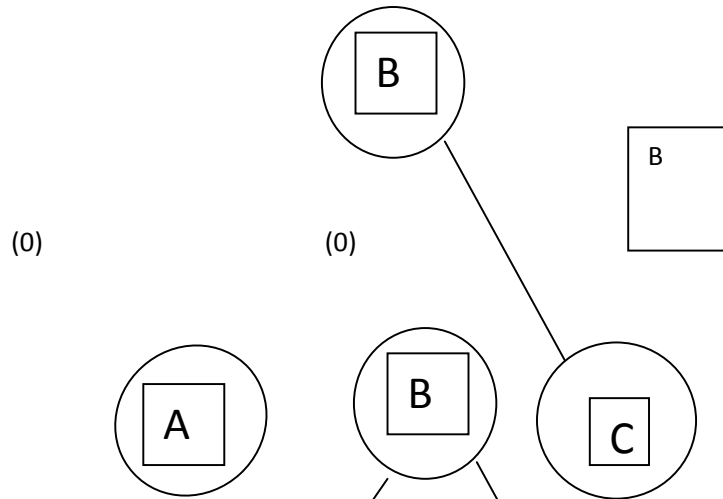


Now, this is a height balanced tree.

**RR Rotation:**

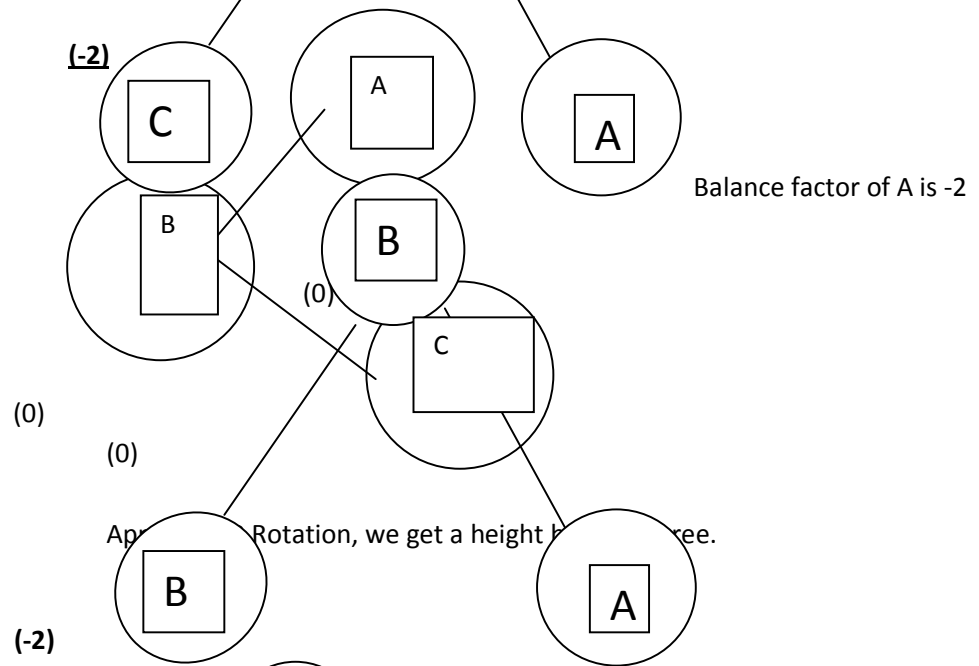


Balance factor of A is -2. Applying RR rotation, we get,

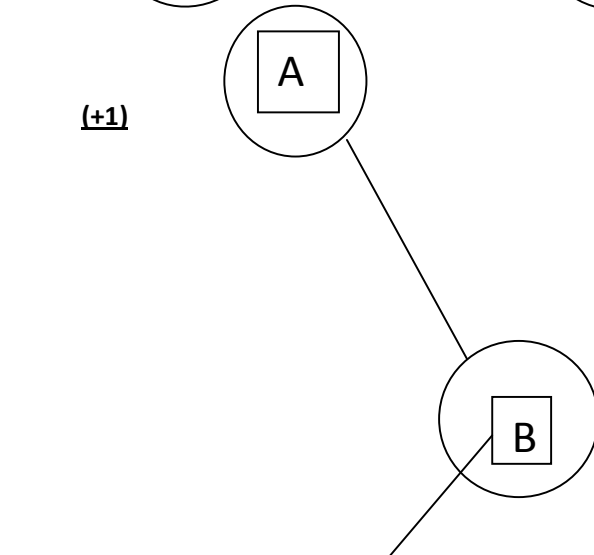


Now, this is a height balanced tree.

**LR Rotation:**



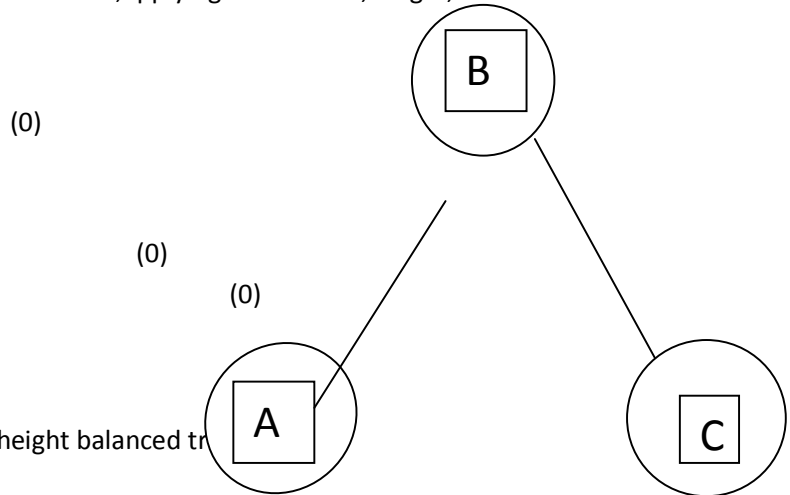
**RL Rotation:**





(0)

Balance factor of A is -2; applying RL Rotation, we get,

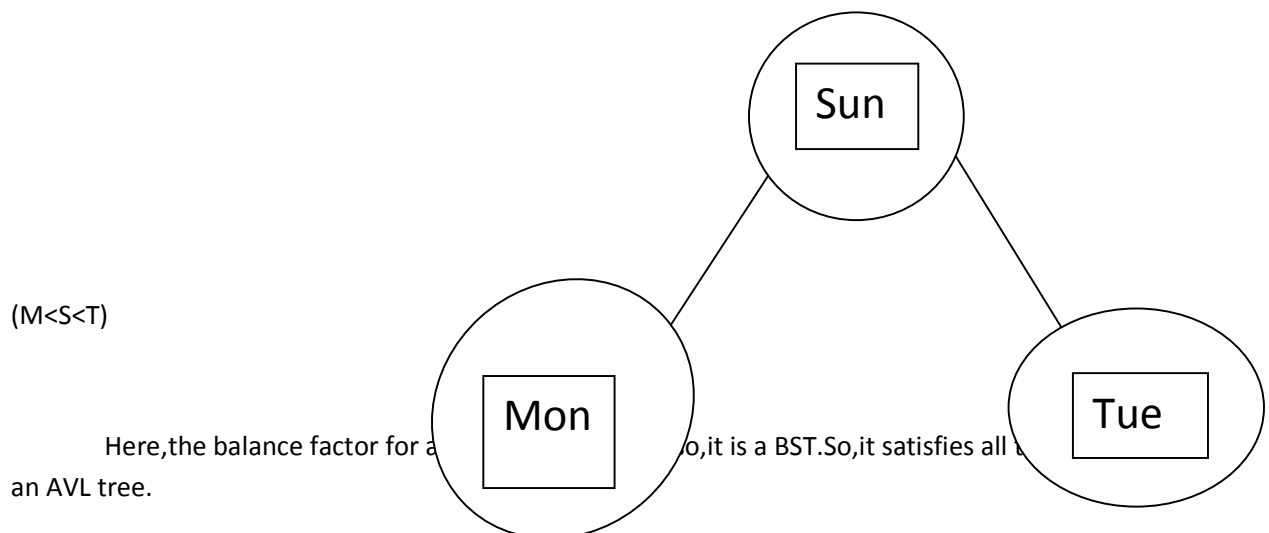


Now, this is a height balanced tree

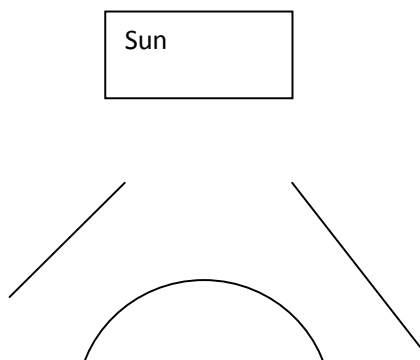
Q. Arrange days of week:-

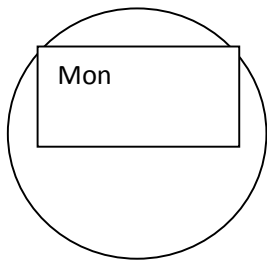
Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, in an AVL tree.

**1<sup>st</sup> step:-** We have to arrange these days alphabetically, and the constructed tree should satisfy the conditions of an AVL Tree. Starting with Sunday (Sun):



**2<sup>nd</sup> step:-** Now, Wednesday is to be inserted, As (W) > (T), so, it will be placed at right of Tuesday for satisfying BST conditions:





Now, balance factor of Sunday=-1

Balance factor of Monday=-1

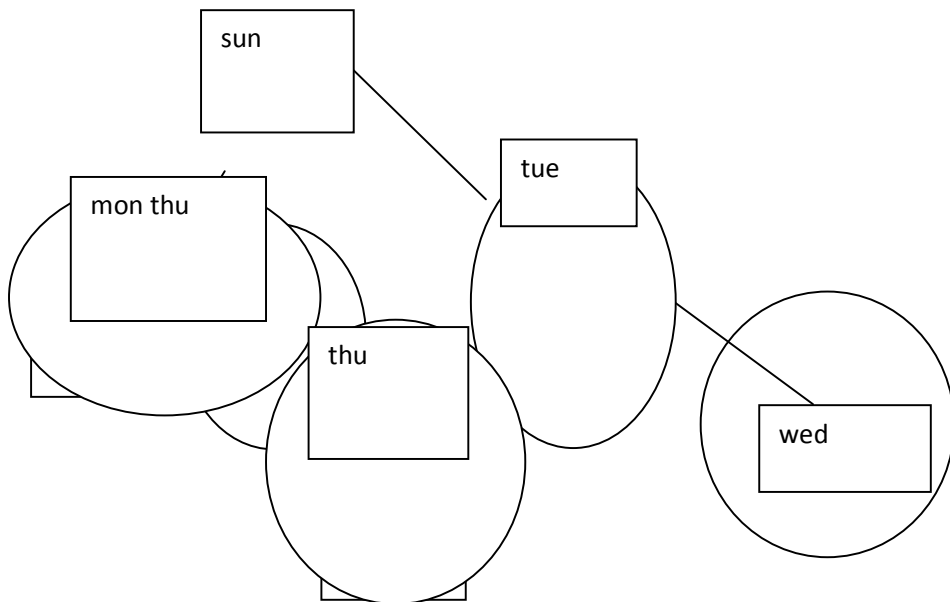
Balance factor of Tuesday=0

Balance factor of Wednesday=-1

Hence, it is an AVL tree.

**3<sup>rd</sup> step:-**

Now, Thursday has to be inserted. As alphabetically, Thursday < Tuesday, so it will be at the left of Tuesday.



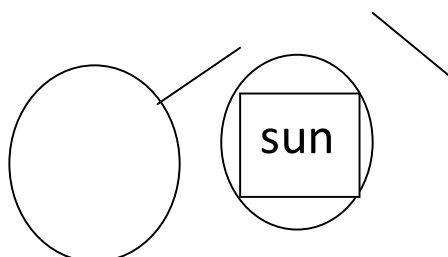
Here, balance factor of sun=-1

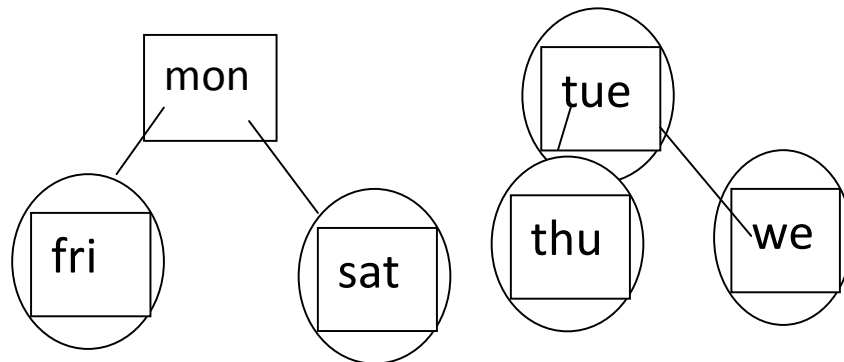
Balance factor of tue, wed, thu=0

So, it is an AVL tree.

**4<sup>th</sup> step:**

Now, Friday and Saturday are to be inserted,





Here, balance factor of all the days=0.

So, it is an AVL tree.

#### Use :

1. Search is  $O(\log N)$  since AVL trees are always balanced.
2. Insertion and deletions are also  $O(\log n)$
3. The height balancing adds no more than a constant factor to the speed of insertion.
4. AVL Tree height is less than BST.
5. Stores data in balanced way in a disk.

#### Limitations

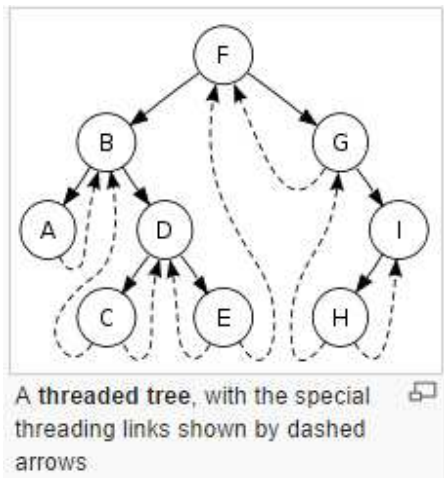
1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have  $O(N)$  for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

## Threaded BST

A **threaded binary tree** defined as follows:

"A binary tree is *threaded* by making all right child pointers that would normally be null point to the inorder successor of the node (if it exists), and all left child pointers that would normally be null point to the inorder predecessor of the node."

A threaded binary tree makes it possible to traverse the values in the binary tree via a linear traversal that is more rapid than a recursive in-order traversal. It is also possible to discover the parent of a node from a threaded binary tree, without explicit use of parent pointers or a stack, albeit slowly. This can be useful where stack space is limited, or where a stack of parent pointers is unavailable (for finding the parent pointer via DFS).



## 7.Types of threaded binary trees

---

1. Single Threaded: each node is threaded towards **either(right)' the in-order predecessor or'** successor.
2. Double threaded: each node is threaded towards **both(left & right)' the in-order predecessor and'** successor.

## SPANNING TREE:-

->A tree is a connected undirected graph with no cycles.It is a spanning tree of a graph G if it spans G (that is, it

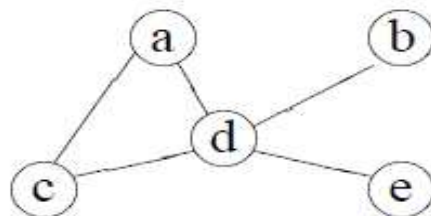
includes every vertex of G) and is a subgraph of G (every edge in the tree belongs to G).

->A spanning tree of a connected graph G can also be defined as a maximal set of edges of G that contains no cycle,

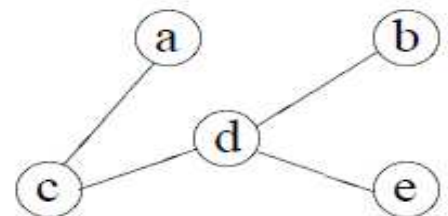
or as a minimal set of edges that connect all vertices.

-> So key to spanning tree is number of edges will be 1 less than the number of nodes.

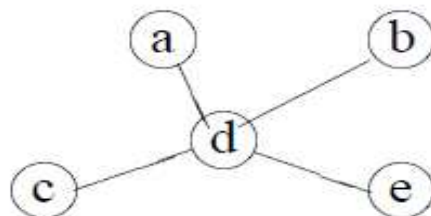
- **no. of edges=no. of nodes-1**  
**example:-**



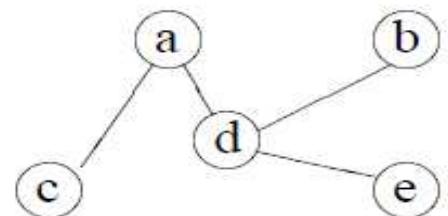
Graph



spanning tree 1



spanning tree 2



spanning tree 3

**Weighted Graphs:-** A weighted graph is a graph, in which each edge has a weight (some real number).

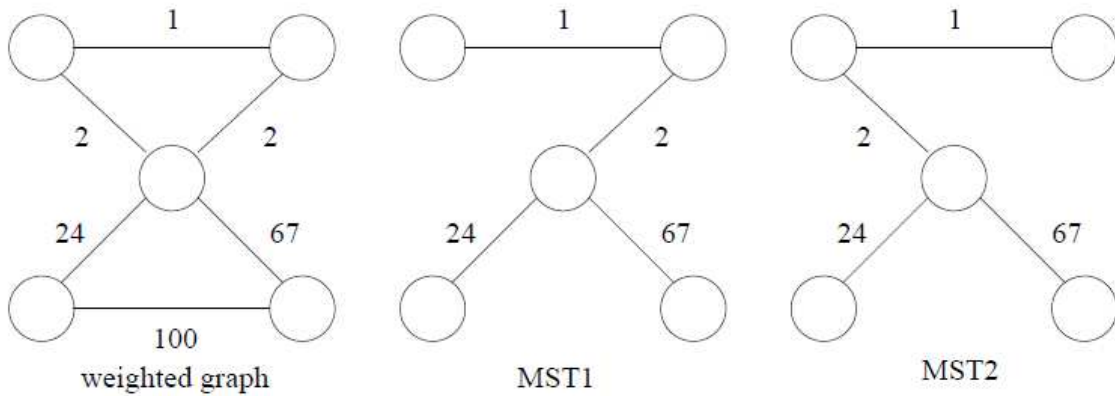
**Weight of a Graph:-** The sum of the weights of all edges.

## Minimum SPANNING TREE:-

Minimum spanning tree is the spanning tree in which the sum of weights on edges is minimum.

**NOTE:-** The minimum spanning tree may not be unique. However, if the weights of all the edges are pairwise distinct, it is indeed unique.

**example:-**



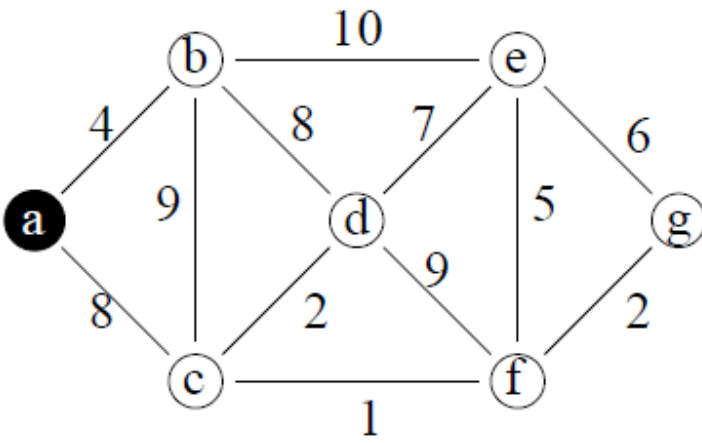
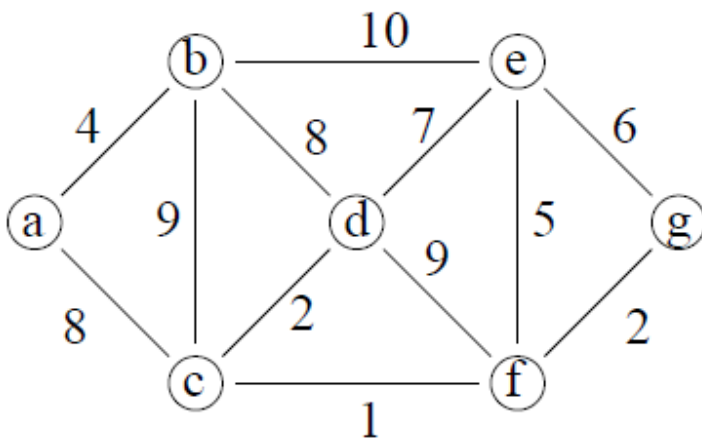
There are a number of ways to find the minimum spanning tree but out of them most popular methods are prim's algorithm and kruskal algorithm.

### **PRIM'S ALGORITHM:-**

**Prim's algorithm** is a greedy algorithm that finds a minimum spanning tree for a connected weighted graph. This means it finds a subset of the edges that forms a tree that includes every vertex, where the total weight of all the edges in the tree is minimized.

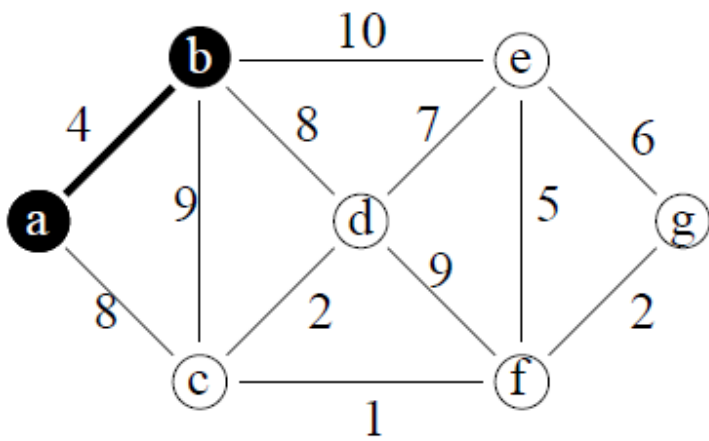
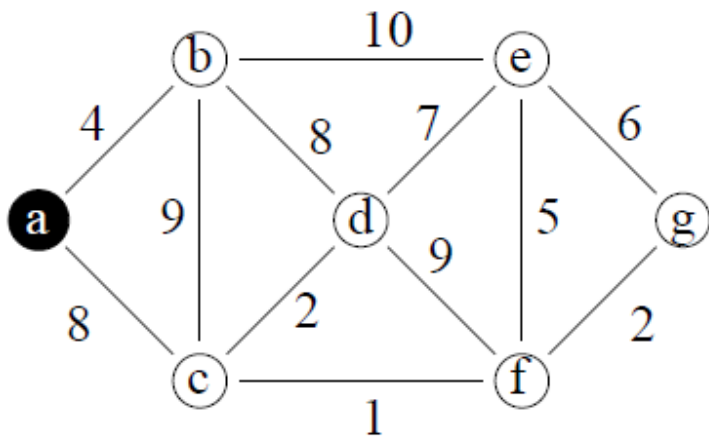
**steps:-**

1. Initialize a tree with a single vertex, chosen arbitrarily from the graph.
2. Grow the tree by one edge: of the edges that connect the tree to vertices not yet in the tree, find the minimum-weight edge, and transfer it to the tree.
3. Repeat step 2 (until all vertices are in the tree).



step-0

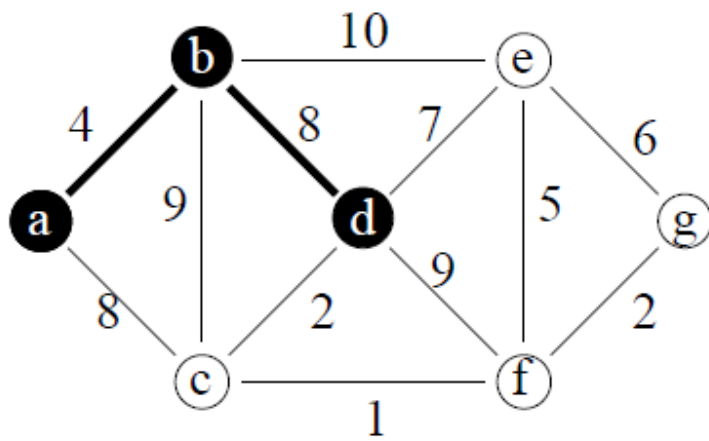
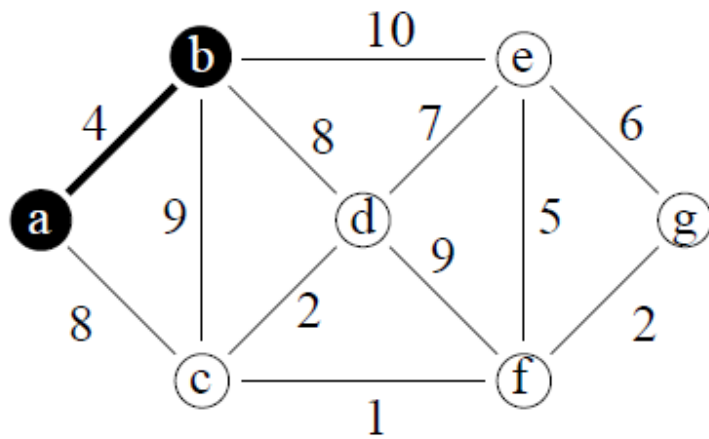
choose a.



step-1

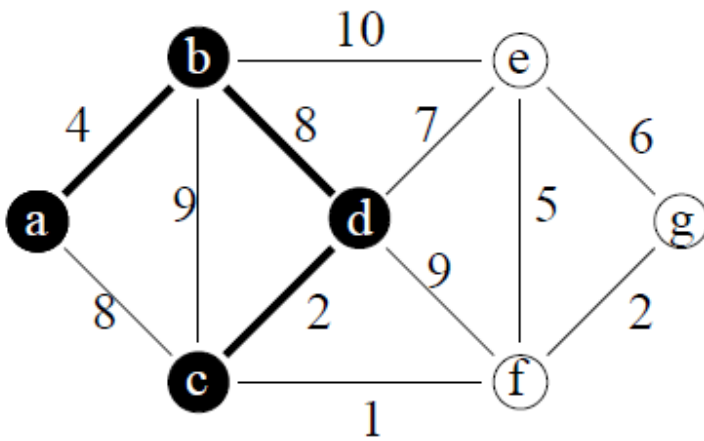
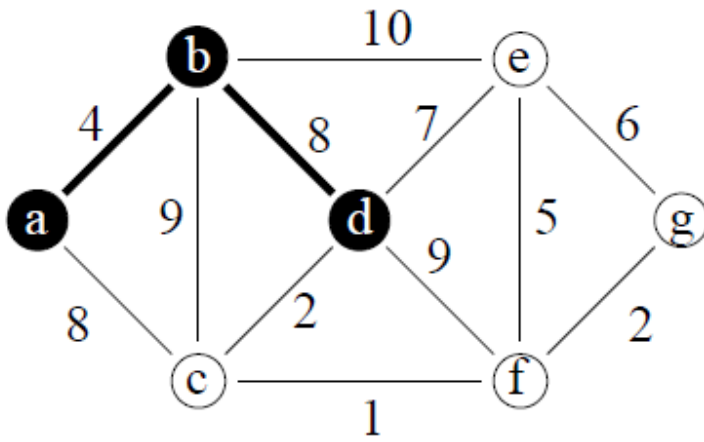
since (a,b) is minimum.





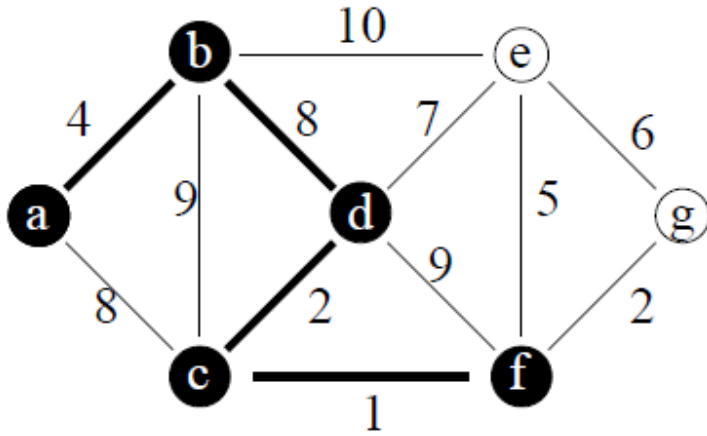
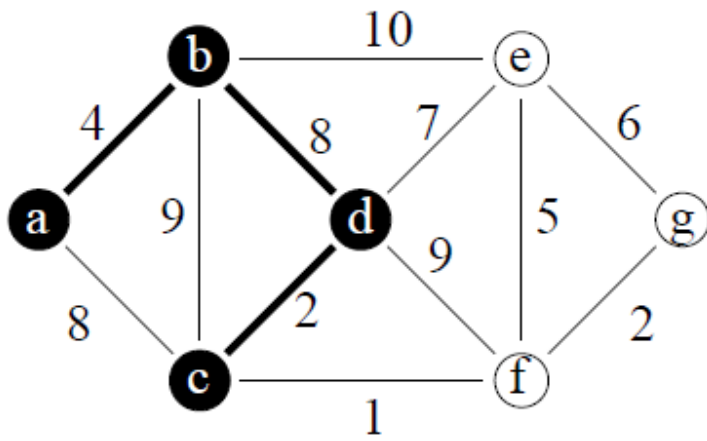
step-2

since (b,d) is minimum.



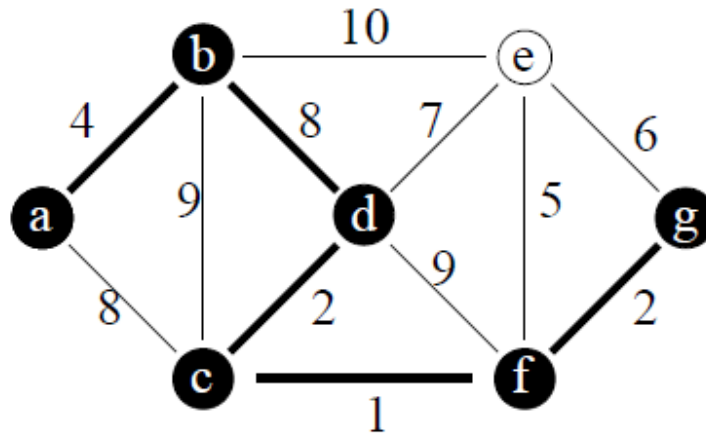
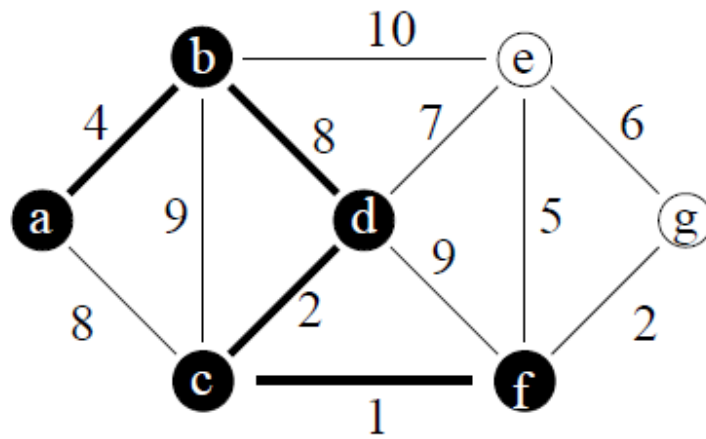
step-3

since weight of (d,c) is minimum.



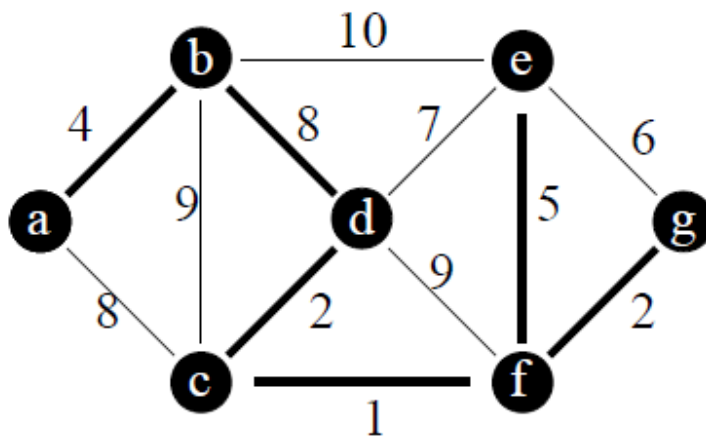
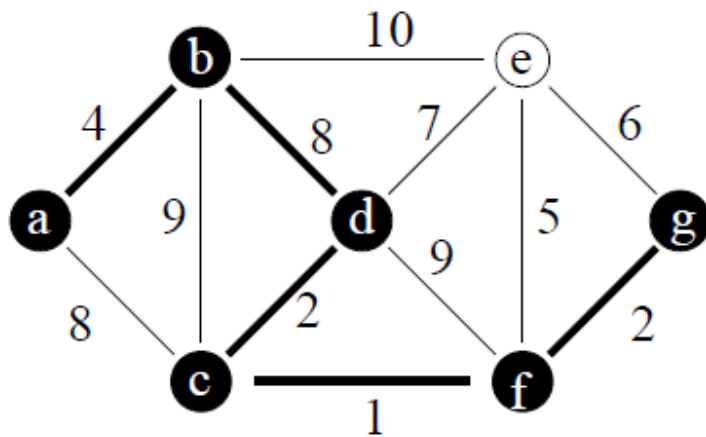
step-4

since (c,f) is minimum.



step-5

since weight of (f,g) is less than (f,e).



step-6

weight of (g,e) is greater than weightof(f,e).

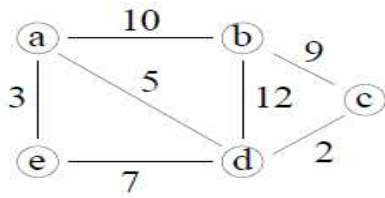
**kruskal algorithm:-**

The Kruskal's Algorithm is based directly on the generic algorithm. Unlike Prim's algorithm, we make a different choices of cut.

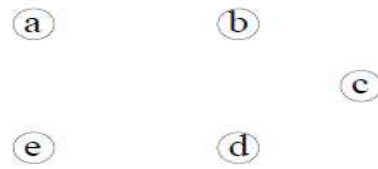
- Initially, trees of the forest are the vertices (no edges).
- In each step add the cheapest edge that does not create a cycle.
- Observe that unlike Prim's algorithm, which only grows one tree, Kruskal's algorithm grows a collection of trees(a forest).
- Continue until the forest 'merge to' a single tree.

This is a *minimum* spanning tree.

example:-



original graph



forest  $\longrightarrow$  MST

	edge	weight
E	{d, c}	2
	{a, e}	3
	{a, d}	5
	{e, d}	7
	{b, c}	9
	{a, b}	10
	{b, d}	12

Forest (V, A)

A = { } }

# HASHING

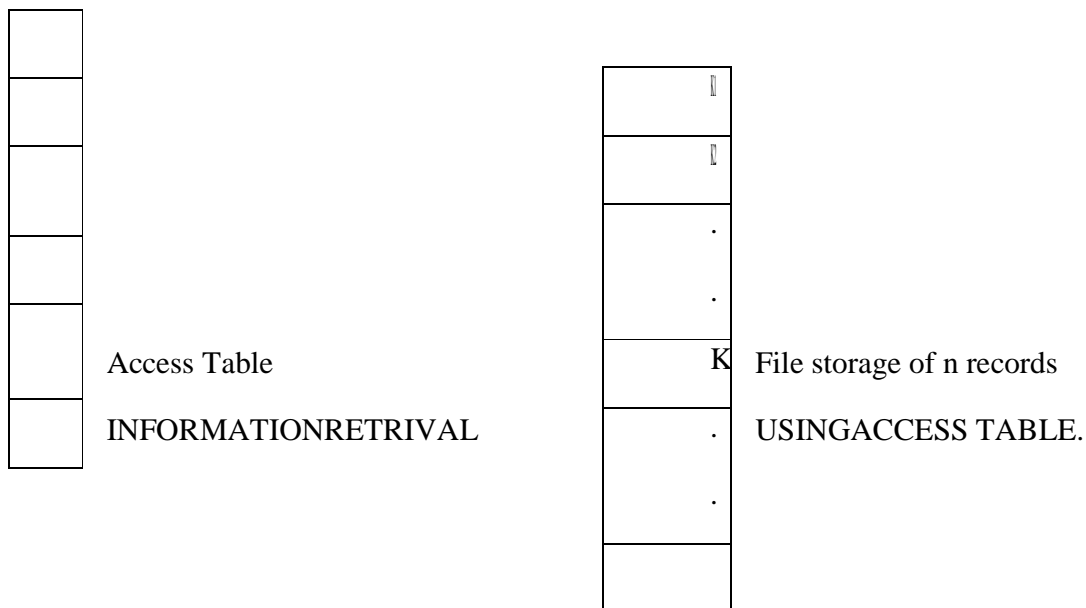
## Introduction:

Hashing involves less key comparison and searching can be performed in constant time.

Suppose we have keys which are in the range 0 to  $n-1$  and all of them are unique.

We can take an array of size  $n$  and store the records in that array based on the condition that key and array index are same.

The searching time required is directly proportional to the number of records in the file. We assume a function  $f$  and if this function is applied on key  $K$  it returns  $i$ , an index so that  $i=f(K)$ . then entry in the access table gives the location of record with key value  $K$ .



## HASH FUNCTIONS

The main idea behind any hash function is to find a one to one correspondence between a key value and index in the hash table where the key value can be placed. There are two principal criteria deciding a hash function  $H:K \rightarrow I$  are as follows:

- i. The function  $H$  should be very easy and quick to compute.

- ii. The function H should achieve an even distribution of keys that actually occur across the range of indices.

Some of the commonly used hash functions applied in various applications are:

#### **DIVISION:**

It is obtained by using the modulo operator. First convert the key to an integer then divide it by the size of the index range and take the remainder as the result.

$H(k)=k\%i$       if indices start from 0

$H(k)=(k\%i)+1$     if indices start from 1

#### **MID –SQUARE:**

The hash function H is defined by  $H(k)=x$  where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. it is criticized as it is time consuming.

#### **FOLDING:**

The key is partitioned into a number of parts and then the parts are added together. There are many variations in this method one is *fold shifting method* where the even number parts are each reversed before the addition. Another is the *fold boundary method* here the two boundary parts are reversed and then are added with all other parts.

### **3. COLLISION RESOLUTION TECHNIQUES**

If for a given set of key values the hash functions does not distribute them uniformly over the hash table, some entries are there which are empty and in some entries more than one key value are to be stored. Allotment of more than one key values in one location in the hash table is called *collision*.




HASH TABLE

Collision in hashing cannot be the size of the hash tables. There techniques to resolve collisions. methods are:

- i. Closed hashing(linear probing)
- ii. Open hashing(chaining)

**CLOSED HASHING:**

19
10
49
9, 31, 77
33
43
55, 62

H:K->I

ignored whatever are several Two important

The simplest method to resolve a collision is closed hashing. Here the hash table is considered as circular so that when the last location is reached the search proceeds to the first location of the table. That is why this is called closed hashing.

The search will continue until any one case occurs:

- 1. The key value is found.
- 2. Unoccupied location is encountered.
- 3. It reaches to the location where the search was started.

**DRAWBACK OF CLOSED HASHING:**

As the half of the hash table is filled there is a tendency towards clustering. The key values are clustered in large groups and as a result sequential search becomes slower and slower.

Some solutions to avoid this are:

- a. Random probing
- b. Double hashing or rehashing
- c. Quadratic probing

### RANDOM PROBING:

This method uses a pseudo random number generator to generate a random sequence of locations, rather than an ordered sequence as was the case in linear probing method. The random sequence generated by the pseudo random number generator contains all positions between 1 and  $h$ , the highest location of the hash table.

$$I=(i+m)\%h+1$$

$i$  is the number in the sequence

$m$  and  $h$  are integers that are relatively prime to each other.

### DOUBLE HASHING:

When two hash functions are used to avoid secondary clustering then it is called double hashing. The second function should be selected in such a way that hash address generated by two hash functions are distinct and the second function generates a value  $m$  for the key  $k$  so that  $m$  and  $h$  are relatively prime.

$$(k)=(k\%h)+1$$

$$(k)=(k\%(h-4))+1$$

### QUADRATIC PROBING:

It is a collision resolution method that eliminates the primary clustering problem of linear probing. For quadratic probing the next location after  $i$  will be  $i+,i++..... etc.$

$$H(k)+\%h \text{ for } i=1,2,3.....$$

### OPEN HASHING

In closed hashing two situations occurs 1. If there is a table overflow situation 2. If the key values are haphazardly intermixed. To solve this problem another hashing is used *open chaining*.

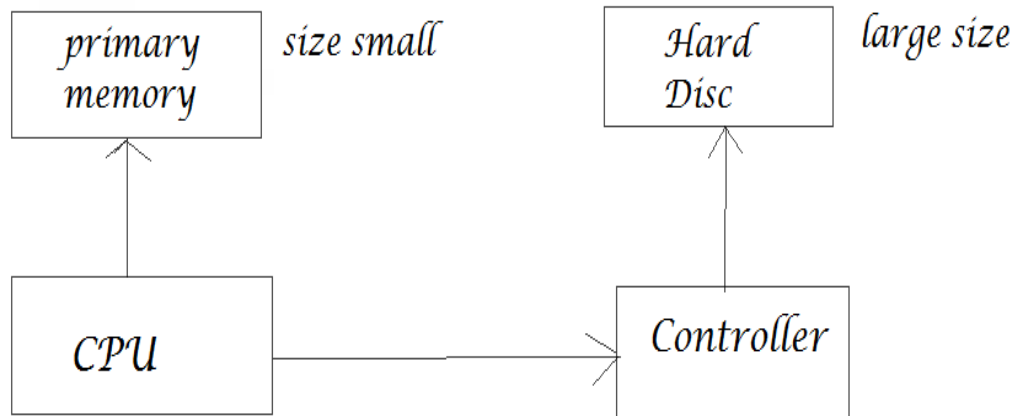
### 4.ADVANTAGES AND DISADVANTAGES OF CHAINING

1. Overflow situation never arises. Hash table maintains lists which can contain any number of key values.
2. Collision resolution can be achieved very efficiently if the lists maintain an ordering of keys so that keys can be searched quickly.
3. Insertion and deletion become quick and easy task in open hashing. Deletion proceeds in exactly the same way as deletion of a node in single linked list.

4. Open hashing is best suitable in applications where number of key values varies drastically as it uses dynamic storage management policy.
5. The chaining has one disadvantage of maintaining linked lists and extra storage space for link fields.

## FILE STRUCTURE

File Organisation:-



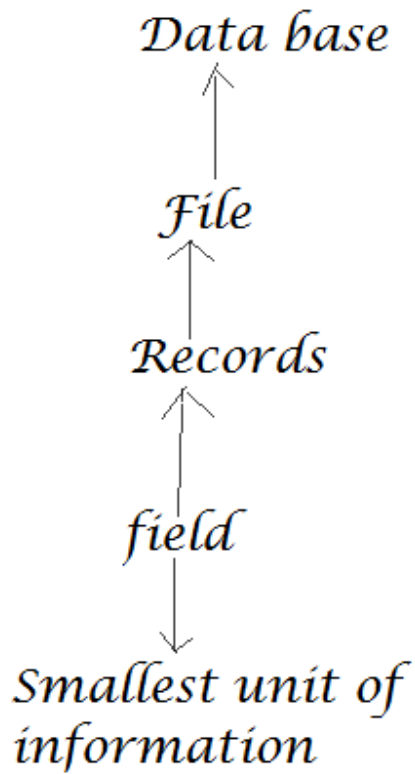
It is the organisation of records in a file.

Record:-

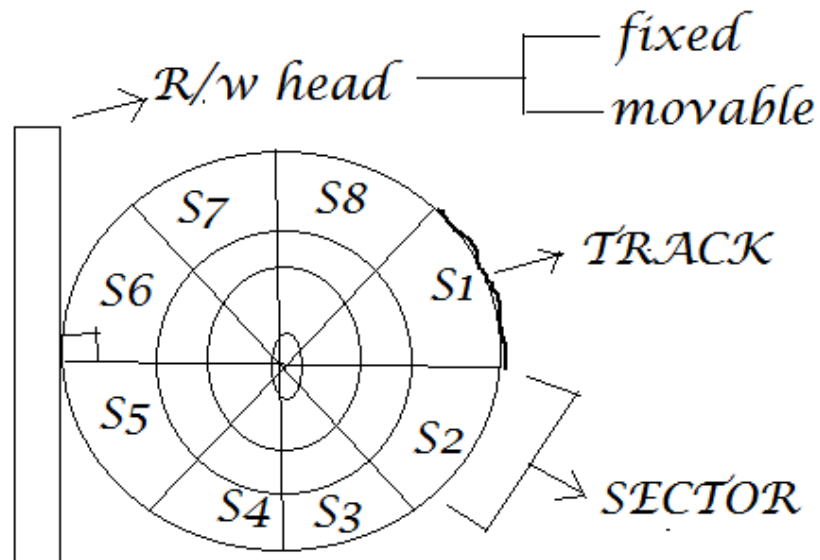
It is the collection of related fields.

Field:-

It is the smallest logically meaningful unit of information in a file.



Secondary memory structure:-



Seek time:-

It is the track access time by R/w haed.

Latency time:-

It is the time taken for the sector movement.

“ $L < S$ ”

Data access time:-It is the time taken for the file movement.Student file:-

KEY:-

Key is the field which uniquely identify the records.

	<i>NAME</i>	<i>ROLL NO.</i>	<i>DOB</i>	<i>DEPT.</i>	<i>FIELD</i>
<i>S1</i>	<i>KAPIL</i>	<i>144</i>	<i>01/04/95</i>	<i>CSE</i>	<i>RECORDS</i>
<i>S2</i>	<i>RAJESH</i>	<i>254</i>	<i>02/06/95</i>	<i>EE</i>	

- (1) Key must have distinct value.  
(no duplicate)
- (2) Key must be in proper order.

File:-

Records binary relation:-

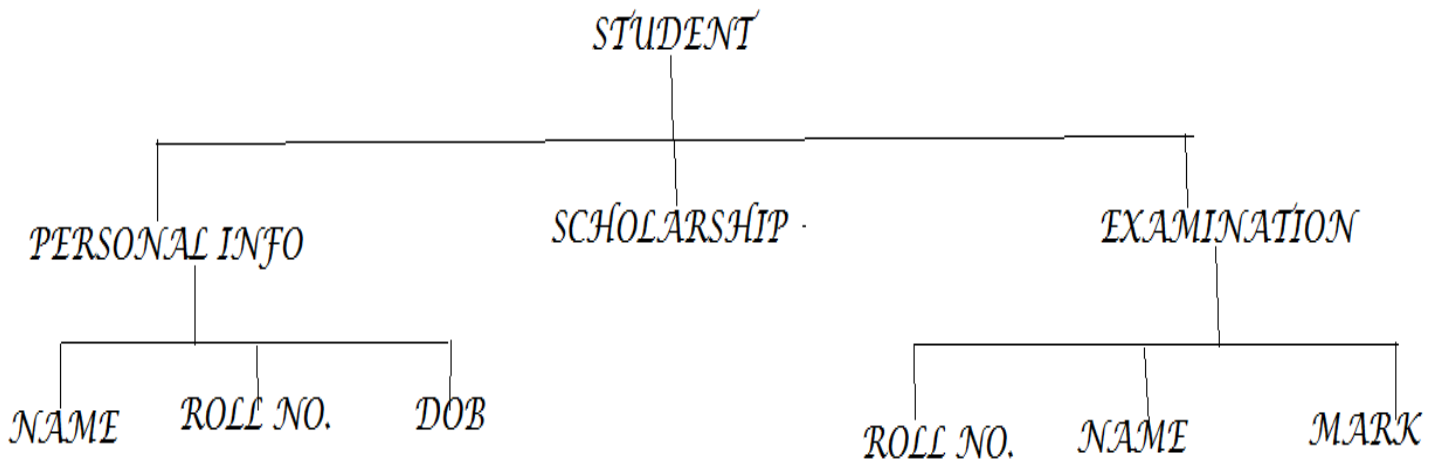
	<i>A</i>	<i>B</i>	<i>C</i>	<i>D</i>	<i>E</i>
<i>r1</i>					
<i>r2</i>					
<i>r3</i>					
<i>r4</i>					

$$r1 \in A*B*C*D*E$$

No. of records(rows) is called cardinality.

No. of fields(columns) is called degree.

APPLICATION:-



Types of files –

1. Serial Files
2. Sequential Files
3. Direct or Random Files
4. Index Sequential Files

**Serial Files –**

- This type of file was used in 1950-1960.
- The records are arranged one after another.
- New record is added at the end of the file.
- These files were stored in magnetic tapes.
- When the size of the file increases, the time required to access data becomes more. This is because it can only apply linear search.
- Logical ordering = Physical ordering
- These are used in secondary files.
- Examples – A material without page no. Here searching will be difficult due to lack of page number.

**Sequential File –**

- The records are ordered.
- . There is no key value.
- Gaps are left so that new records can be added there to maintain ordering.
- New record is added in gaps left between records according to ordering.
- These files were stored in magnetic tapes.
- When the size of the file increases, the time required to access data becomes more. This is because there are no key.
- Logical ordering may not be equal to Physical ordering.
- These are used in master files.
- Examples – Arranging cards in sequential order (A 2 3 4 5 6 7 8 9 10 J Q K)



### **Direct or Random Files-**

- Each record is associated with a direct function.
- There are key values.
- The records have mapping.
- Disk device like CD are used.
- Searching is quit faster.
- Hashing is its extension.
- Due to mapping more space is needed.
- It is more complex than the previous 2 files.

### **Index Sequential File –**

- This was invented by IBM.
- These uses indices.
- Access is sequential.
- Indexing and sorting is random.
- These have keys.
- A group of keys are given one index.
- Disk device is used for storage.
- Searching is fast as the index is searched and then the key is searched.
- This is used in banking.
- Example – Contents of a book. The topics are the keys. They have indices like page number. That topic can be found in that page no. When new information needs to be added, a pointer is taken to point to a new location like in appendix of a book. This saves the time and errors that occur due to shifting the later data after insertion.

## **Key-**

A vital, crucial element notched and grooved, usually metal implement that is turned to open or close a lock. The keys have the characteristics of having unique attribute.

In database management systems, a key is a field that you use to sort data.

It can also be called a key field , sort key, index, or key word.

For example, if you sort records by age, then the age field is a key.

Most database management systems allow you to have more than one key so that you can sort records in different ways.

One of the keys is designated the primary key, and must hold a unique value for each record.

A key field that identifies records in a different table is called a foreign key.

**LOCK:** In general a device operated by a key, combination, or keycard and used for holding, closing or securing the data.

The key lock principle states that once lock can be opened or fastened by specific one type of keys only. In digital electronics latch is used for temporary security while lock is used for permanent security.

## **FILE ORGANISATION TECHNIQUES:**

The structure of a file (especially a data file), defined in terms of its components and how they are mapped onto backing store.

Any given file organization supports one or more file access methods.

Organization is thus closely related to but conceptually distinct from access methods.

The distinction is similar to that between data structure and the procedures and functions that operate on them (indeed a file organization is a large-scale data structure), or to that between a logical schema of a database and the facilities in a data manipulation language.

There is no very useful or commonly accepted taxonomy of methods of file organization: most attempts confuse organization with access methods.

Choosing a file organization is a design decision, hence it must be done having in mind the achievement of good performance with respect to the most likely usage of the file.

The criteria usually considered important are:

1. Fast access to single record or collection of related records.
2. Easy record adding/update/removal, without disrupting
3. Storage efficiency.
4. Redundancy as a warranty against data corruption.

To read a specific record from an indexed sequential file, you would include the KEY= parameter in the READ (or associated input) statement.

The "key" in this case would be a specific record number (e.g., the number 35 would represent the 35th record in the file).

The direct access to a record moves the record pointer, so that subsequent sequential access would take place from the new record pointer location, rather than the beginning of the file.

Now question arises how to access these files. We need KEYS to access the file.

#### **TYPES OF KEYS:**

**PRIMARY KEYS:** The primary key of a relational table uniquely identifies each record in the table.

It can either be a normal attribute that is guaranteed to be unique (such as Social Security Number in a table with no more than one record per person) or it can be generated by the DBMS (such as a globally unique identifier, or GUID, in Microsoft SQL Server).

Primary keys may consist of a single attribute or multiple attributes in combination.

Examples: Imagine we have a STUDENTS table that contains a record for each student at a university.

The student's unique student ID number would be a good choice for a primary key in the STUDENTS table.

The student's first and last name would not be a good choice, as there is always the chance that more than one student might have the same name.

**ALTERNATE KEY:** The keys other than the primary keys are known as alternate key.

**CANDIDATE KEY:** The Candidate Keys are super keys for which no proper subset is a super key. In other words candidate keys are minimal super keys.

**SUPER KEY:** Super key stands for superset of a key. A Super Key is a set of one or more attributes that are taken collectively and can identify all other attributes uniquely.

**SECONDARY KEY:** Any field in a record may be a secondary key.

The problem with secondary keys is that they are not unique and are therefore likely to return more than one record for a particular value of the key.

Some fields have a large enough range of values that a search for a specific value will produce only a few records; other fields have a very limited range of values and a search for a specific value will return a large proportion of the file.

An example of the latter would be a search in student records for students classified as freshmen.

**FOREIGN KEY :** Foreign Key (Referential integrity) is a property of data which, when satisfied, requires every value of one attribute of a relation to exist as a value of another attribute in a different relation.

For referential integrity to hold in a relational database, any field in a table that is declared a foreign key can contain either a null value, or only values from a parent table's primary key or a candidate key.

In other words, when a foreign key value is used it must reference a valid, existing primary key in the parent table.

For instance, deleting a record that contains a value referred to by a foreign key in another table would break referential integrity.

Some relational database management systems can enforce referential integrity, normally either by deleting the foreign key rows as well to maintain integrity, or by returning an error and not performing the delete.

Which method is used may be determined by a referential integrity constraint defined in a data dictionary. "Referential" the adjective describes the action that a foreign key performs, 'referring' to a link field in another table.

In simple terms, 'referential integrity' is a guarantee that the target it 'refers' to will be found.

A lack of referential integrity in a database can lead relational databases to return incomplete data, usually with no indication of an error.

A common problem occurs with relational database tables linked with an 'inner join' which requires non-NULL values in both tables, a requirement that can only be met through careful design and referential integrity.

#### **ENTITY INTEGRITY:**

In the relational data model, entity integrity is one of the three inherent integrity rules.

Entity integrity is an integrity rule which states that every table must have a primary key and that the column or columns chosen to be the primary key should be unique and not NULL.

Within relational databases using SQL, entity integrity is enforced by adding a primary key clause to a schema definition.

The system enforces Entity Integrity by not allowing operation (INSERT, UPDATE) to produce an invalid primary key.

Any operation that is likely to create a duplicate primary key or one containing nulls is rejected.

The Entity Integrity ensures that the data that you store remains in the proper format as well as comprehensible. SU