

# Experiment No 1

## **Title:**

Consider telephone book database of N clients. Make use of a hash table implementation to quickly look up client's telephone number

## **Objectives:**

1. To understand concept of Hashing
2. To understand to find record quickly using hash function.
3. To understand concept & features of object oriented programming.

## **Learning Objectives**

- ✓ To understand concept of hashing.
- ✓ To understand operations like insert and search record in the database.

## **Learning Outcome**

- ✓ Learn object oriented Programming features
- ✓ Understand & implement concept of hash table .

## **Theory:**

Hash tables are an efficient implementation of a keyed array data structure, a structure sometimes known as an associative array or map. If you're working in C++, you can take advantage of the STL map container for keyed arrays implemented using binary trees, but this article will give you some of the theory behind how a hash table works.

### **Keyed Arrays vs. Indexed Arrays**

One of the biggest drawbacks to a language like C is that there are no keyed arrays. In a normal C array (also called an indexed array), the only way to access an element would be through its index number. To find element 50 of an array named "employees" you have to access it like this:

---

```
1employees[50];
```

In a keyed array, however, you would be able to associate each element with a "key," which can be anything from a name to a product model number. So, if you have a keyed array of employee records, you could access the record of employee "John Brown" like this:

```
1employees["Brown, John"];
```

One basic form of a keyed array is called the hash table. In a hash table, a key is used to find an element instead of an index number. Since the hash table has to be coded using an indexed array, there has to be some way of transforming a key to an index number. That way is called the hashing function.

## Hashing Functions

A hashing function can be just about anything. How the hashing function is actually coded depends on the situation, but generally the hashing function should return a value based on a key and the size of the array the hashing table is built on. Also, one important thing that is sometimes overlooked is that a hashing function has to return the same value every time it is given the same key.

Let's say you wanted to organize a list of about 200 addresses by people's last names. A hash table would be ideal for this sort of thing, so that you can access the records with the people's last names as the keys.

First, we have to determine the size of the array we're using. Let's use a 260 element array so that there can be an average of about 10 element spaces per letter of the alphabet.>

Now, we have to make a hashing function. First, let's create a relationship between letters and numbers:

A-->0

B --> 1

C --> 2

D --> 3

...

and so on until Z --> 25.

The easiest way to organize the hash table would be based on the first letter of the last name.

Since we have 260 elements, we can multiply the first letter of the last name by 10. So, when a key like "Smith" is given, the key would be transformed to the index 180 (S is the 19 letter of the alphabet, so S --> 18, and  $18 * 10 = 180$ ).

Since we use a simple function to generate an index number quickly, and we use the fact that the index number can be used to access an element directly, a hash table's access time is quite small. A linked list of keys and elements wouldn't be nearly as fast, since you would have to search through every single key-element pair.

## Basic Operations

Following are the basic primary operations of a hash table.

- **Search** – Searches an element in a hash table.
- **Insert** – inserts an element in a hash table.
- **delete** – Deletes an element from a hash table.

## DataItem

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

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

## Hash Method

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

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

## Search Operation

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

## Example

```
struct DataItem *search(int key)
{
    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty
    while(hashArray[hashIndex] != NULL) {
```

```

    if(hashArray[hashIndex]->key == key)
        return hashArray[hashIndex];

    //go to next cell
    ++hashIndex;

    //wrap around the table
    hashIndex %= SIZE;
}

return NULL;
}

```

### **Insert Operation**

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

#### **Example**

```

void insert(int key,int data)
{
    struct DataItem *item = (struct DataItem*) malloc(sizeof(struct DataItem));
    item->data = data;
    item->key = key;

    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty or deleted cell
    while(hashArray[hashIndex] != NULL &&
        hashArray[hashIndex]->key != -1) { //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    hashArray[hashIndex] = item;
}

```

### **Delete Operation**

Whenever an element is to be deleted, compute the hash code of the key passed and locate the index using that hash code as an index in the array. Use linear probing to get the element ahead if an

element is not found at the computed hash code. When found, store a dummy item there to keep the performance of the hash table intact.

### Example

```
struct DataItem* delete(struct DataItem* item) {
    int key = item->key;

    //get the hash
    int hashIndex = hashCode(key);

    //move in array until an empty
    while(hashArray[hashIndex] !=NULL) {

        if(hashArray[hashIndex]->key == key) {
            struct DataItem* temp = hashArray[hashIndex];
            //assign a dummy item at deleted position
            hashArray[hashIndex] = dummyItem;
            return temp;
        }

        //go to next cell
        ++hashIndex;

        //wrap around the table
        hashIndex %= SIZE;
    }

    return NULL;
}
```

### Expected Output

Menu

1.Create Telephone book

2.Display

3.Look up

Enter Choice1

how many entries2

enter Namea

enter number1234567890

enter Named

enter number3216549876

do u want to continue?(1 for continue)1

Menu

1.Create Telephone book

```
2.Display
3.Look up
Enter Choice2
```

d 3216549876

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

0

a 1234567890

0

0

do u want to continue?(1 for continue)1

Menu

## 1. Create Telephone book

## 2.Display

### 3.Look up

Enter Choice3

enter Name to searchd

found at 0

no of comparision1

```
do u want to continue?(1 for continue)0*/
```

**Conclusion:** In this way we have implemented Hash table for quick lookup using C++.

# Experiment No 2

## **Title:**

A book consists of chapters, chapters consist of sections and sections consist of subsections. Construct a tree and print the nodes. Find the time and space requirements of your method.

## **Objectives:**

1. To understand concept of tree data structure
2. To understand concept & features of object oriented programming.

## **Learning Objectives:**

- ✓ To understand concept of class
- ✓ To understand concept & features of object oriented programming.
- ✓ To understand concept of tree data structure.

## **Learning Outcome:**

- Define class for structures using Object Oriented features.
- Analyze tree data structure.

## **Theory:**

### **Introduction to Tree:**

### **Definition:**

A tree  $T$  is a set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following

- if  $T$  is not empty,  $T$  has a special tree called the root that has no parent
- each node  $v$  of  $T$  different than the root has a unique parent node  $w$ ; each node with parent  $w$  is a child of  $w$

### **Recursive definition**

- $T$  is either empty
- or consists of a node  $r$  (the root) and a possibly empty set of trees whose roots are the children of  $r$

Tree is a widely-used data structure that emulates a tree structure with a set of linked nodes. The tree graphically is represented most commonly as on *Picture 1*. The circles are the nodes and the edges are the links between them.

Trees are usually used to store and represent data in some hierarchical order. The data are stored in the nodes, from which the tree is consisted of.

A node may contain a value or a condition or represent a separate data structure or a tree of its own. Each node in a tree has zero or more child nodes, which are one level lower in the tree hierarchy (by convention, trees grow down, not up as they do in nature). A node that has a child is called the child's parent node (or ancestor node, or superior). A node has at most one parent. A node that has no childs is called a leaf, and that node is of course at the bottommost level of the tree. The height of a node is the length of the longest path to a leaf from that node. The height of the root is the height of the tree. In other words, the "height" of tree is the "number of levels" in the tree. Or more formally, the height of a tree is defined as follows:

1. The height of a tree with no elements is 0
2. The height of a tree with 1 element is 1
3. The height of a tree with  $> 1$  element is equal to 1 + the height of its tallest subtree.

The depth of a node is the length of the path to its root (i.e., its root path). Every child node is always one level lower than his parent.

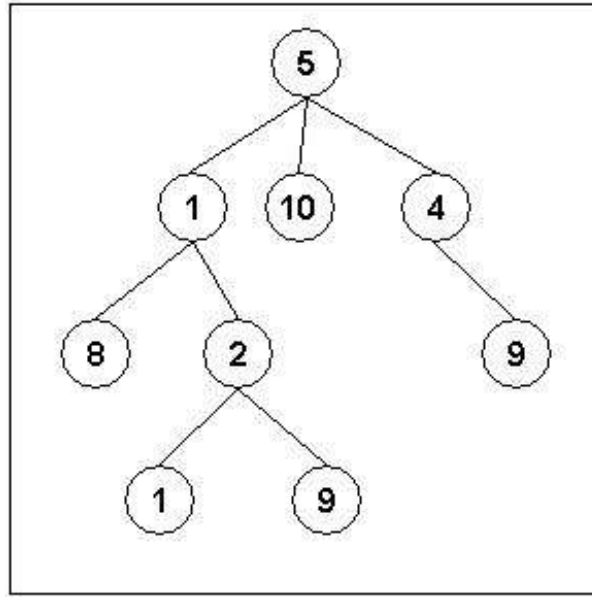
The topmost node in a tree is called the root node. Being the topmost node, the root node will not have parents. It is the node at which operations on the tree commonly begin (although some algorithms begin with the leaf nodes and work up ending at the root). All other nodes can be reached from it by following edges or links. (In the formal definition, a path from a root to a node, for each different node is always unique). In diagrams, it is typically drawn at the top.

In some trees, such as heaps, the root node has special properties.

A subtree is a portion of a tree data structure that can be viewed as a complete tree in itself. Any node in a tree  $T$ , together with all the nodes below his height, that are reachable from the node, comprise a subtree of  $T$ . The subtree corresponding to the root node is the entire tree; the subtree corresponding to any other node is called a proper subtree (in analogy to the term proper subset).

Every node in a tree can be seen as the root node of the subtree rooted at that node.





*Fig1. An example of a tree*

An internal node or inner node is any node of a tree that has child nodes and is thus not a leaf node.

There are two basic types of trees. In an unordered tree, a tree is a tree in a purely structural sense — that is to say, given a node, there is no order for the children of that node. A tree on which an order is imposed — for example, by assigning different natural numbers to each child of each node — is called an ordered tree, and data structures built on them are called ordered tree data structures. Ordered trees are by far the most common form of tree data structure. Binary search trees are one kind of ordered tree.

## Important Terms

Following are the important terms with respect to tree.

- **Path** – Path refers to the sequence of nodes along the edges of a tree.
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.
- **Parent** – Any node except the root node has one edge upward to a node called parent.
- **Child** – The node below a given node connected by its edge downward is called its child node.
- **Leaf** – The node which does not have any child node is called the leaf node.
- **Subtree** – Subtree represents the descendants of a node.
- **Visiting** – Visiting refers to checking the value of a node when control is on the node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.

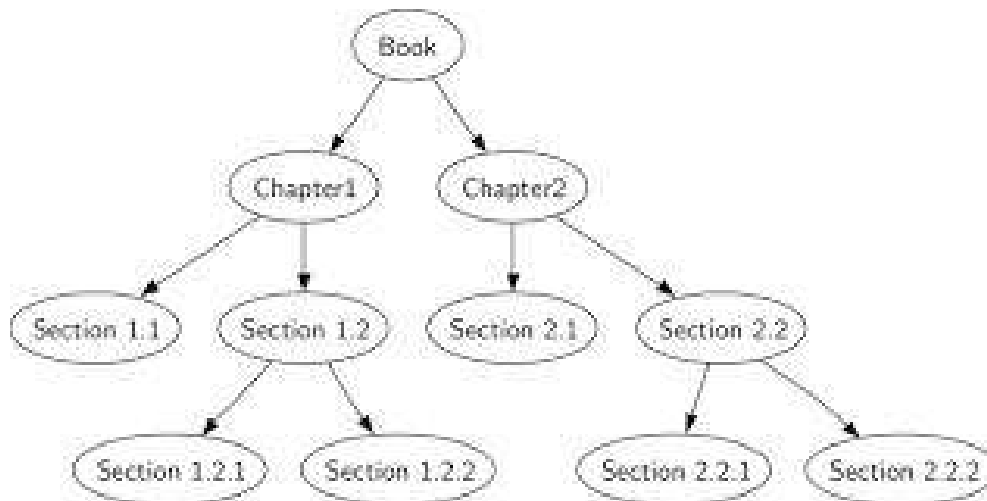
- **keys** – Key represents a value of a node based on which a search operation is to be carried out for a node.

## Advantages of trees

Trees are so useful and frequently used, because they have some very serious advantages:

- Trees reflect structural relationships in the data
- Trees are used to represent hierarchies
- Trees provide an efficient insertion and searching
- Trees are very flexible data, allowing to move subtrees around with minimum

effort For this assignment we are considering the tree as follows.



**Software Required:** g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

**Input:** Book name & its number of sections and subsections along with name.

**Output:** Formation of tree structure for book and its sections.

**Conclusion:** This program gives us the knowledge tree data structure.

## **OUTCOME**

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features.

**ELO2:** Understand & implement tree data structure.

## **Questions asked in university exam.**

1. What is class, object and data structure?
2. What is tree data structure?
3. Explain different types of tree?

# Experiment No 3

## **Title:**

There are flight paths between cities. If there is a flight between city A and city B then there is an edge between the cities. The cost of the edge can be the time that flight take to reach city B from A, or the amount of fuel used for the journey. Represent this as a graph. The node can be represented by airport name or name of the city. Use adjacency list representation of the graph or use adjacency matrix representation of the graph. Justify the storage representation used.

## **Objectives:**

1. To understand concept of Graph data structure
2. To understand concept of representation of graph.

## **Learning Objectives:**

- ✓ To understand concept of Graph data structure
- ✓ To understand concept of representation of graph.

## **Learning Outcome:**

- Define class for graph using Object Oriented features.
- Analyze working of functions.

## **Theory:**

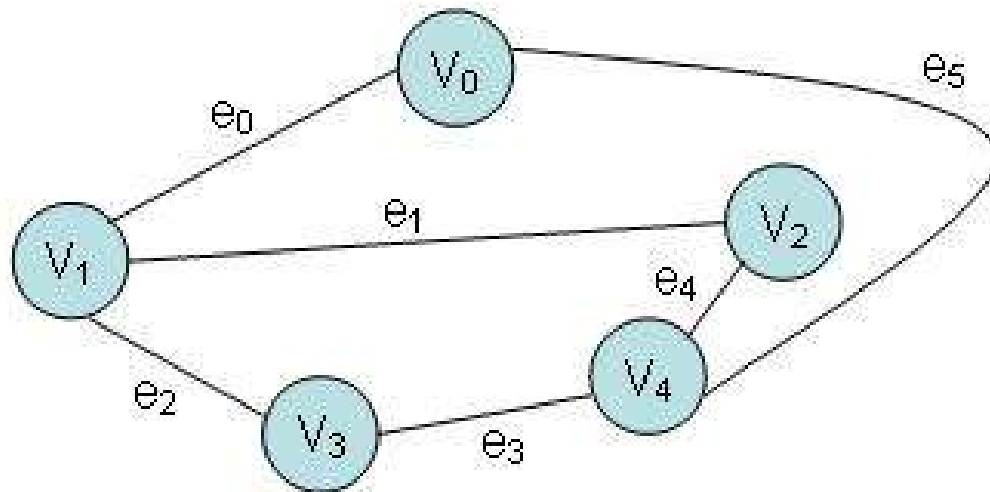
Graphs are the most general data structure. They are also commonly used data structures.

## **Graph definitions:**

- A non-linear data structure consisting of nodes and links between nodes.

## Undirected graph definition:

- An undirected graph is a set of nodes and a set of links between the nodes.
- Each node is called a **vertex**, each link is called an **edge**, and each edge connects two vertices.
- The order of the two connected vertices is unimportant.
- An undirected graph is a finite set of vertices together with a finite set of edges. Both sets might be empty, which is called the empty graph.



## **Graph Implementation:**

Different kinds of graphs require different kinds of implementations, but the fundamental concepts of all graph implementations are similar. We'll look at several representations for one particular kind of graph: directed graphs in which loops are allowed.

## Representing Graphs with an Adjacency Matrix

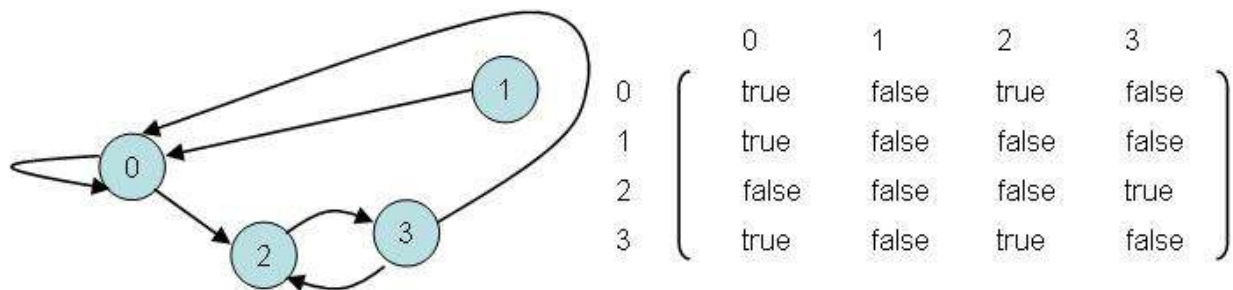


Fig: Graph and adjacency matrix

## **Definition:**

- An adjacency matrix is a square grid of true/false values that represent the edges of a graph.

- If the graph contains  $n$  vertices, then the grid contains  $n$  rows and  $n$  columns.
- For two vertex numbers  $i$  and  $j$ , the component at row  $i$  and column  $j$  is true if there is an edge from vertex  $i$  to vertex  $j$ ; otherwise, the component is false.

We can use a two-dimensional array to store an adjacency matrix:

```
boolean[][] adjacent = new boolean[4][4];
```

Once the adjacency matrix has been set, an application can examine locations of the matrix to determine which edges are present and which are missing.

### Representing Graphs with Edge Lists

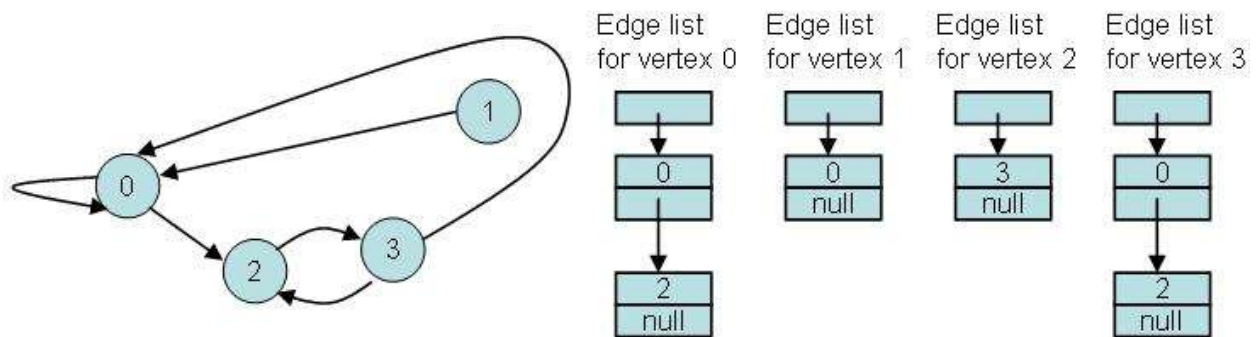


Fig: Graph and adjacency list for each node

#### **Definition:**

- A directed graph with  $n$  vertices can be represented by  $n$  different linked lists.
- List number  $i$  provides the connections for vertex  $i$ .
- For each entry  $j$  in list number  $i$ , there is an edge from  $i$  to  $j$ .

Loops and multiple edges could be allowed.

### Representing Graphs with Edge Sets

To represent a graph with  $n$  vertices, we can declare an array of  $n$  sets of integers. For example:

```
IntSet[] connections = new IntSet[10]; // 10 vertices
```

A set such as `connections[i]` contains the vertex numbers of all the vertices to which vertex  $i$  is connected.

**Software Required:** g++ / gcc compiler- / 64 bit Fedora, eclipse IDE


**Input:** 1.Number of cities.  
2.Time required to travel from one city to another.


**Output:** Create Adjacency matrix to represent path between various cities.

**Conclusion:** This program gives us the knowledge of adjacency matrix graph.

## **OUTCOME**

**Upon completion Students will be able to:**

**ELO1:** Learn concept of graph data structure. 

**ELO2:** Understand & implement adjacency matrix for graph. 

## **Questions asked in university exam.**

1. What are different ways to represent the graph? Give suitable example.
2. What is time complexity of function to create adjacency matrix?

## **Expected Output**

Menu

1.Create Graph using adjacency List

2.Display Graph

Enter Choice1

No of Cities ?4

No of Flights?3

Please enter source city and destination city starting from A upto number of cities like A,B,C,D

Edge no -> 1

Source city->A

Destination city-> B

cost->23

Edge no -> 2

Source city->B

Destination city-> C

cost->35

Edge no -> 3

Source city->C

Destination city-> D

cost->56

do u want to continue?(1 for continue)1

Menu

1.Create Graph using adjacency List

2.Display Graph

Enter Choice2

A--> B & cost23

|

B--> C & cost35

|

C--> D & cost56

|

D

do u want to continue?(1 for continue)0

\*/



# Experiment No 4

**Title:** A Dictionary stores keywords & its meanings. Provide facility for adding new keywords, deleting keywords, updating values of any entry. Provide facility to display whole data sorted in ascending/ Descending order. Also find how many maximum comparisons may require for finding any keyword. Use Height balance tree and find the complexity for finding a keyword.

## **Objectives:**

1. To understand concept of height balanced tree data structure.
2. To understand procedure to create height balanced tree.

## **Learning Objectives:**

- ✓ To understand concept of height balanced tree data structure.
- ✓ To understand procedure to create height balanced tree.

## **Learning Outcome:**

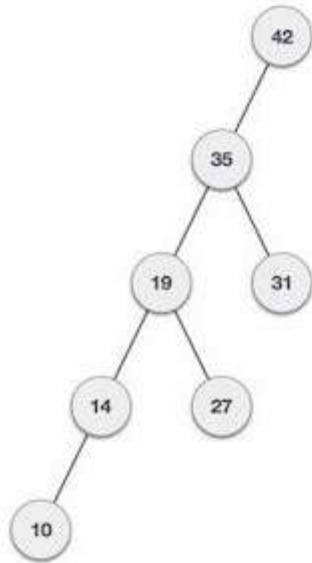
- Define class for AVL using Object Oriented features.
- Analyze working of various operations on AVL Tree .

## **Theory:**

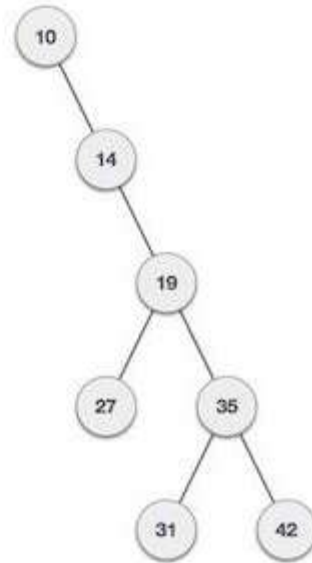
An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR as its left and right sub trees. The T is height balance if and only if Its balance factor is 0, 1, -1.

**AVL (Adelson- Velskii and Landis) Tree:** A balance binary search tree. The best search time, that is  $O(\log N)$  search times. An AVL tree is defined to be a well-balanced binary search tree in which each of its nodes has the AVL property. The AVL property is that the heights of the left and right sub-trees of a node are either equal or if they differ only by 1.

What if the input to binary search tree comes in a sorted (ascending or descending) manner? It will then look like this –



If input 'appears' non-increasing manner

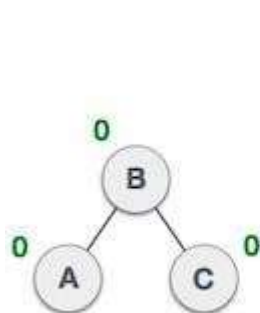


If input 'appears' in non-decreasing manner

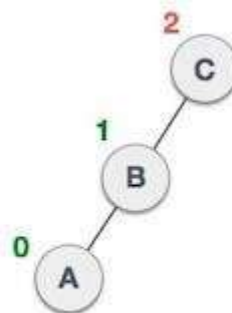
It is observed that BST's worst-case performance is closest to linear search algorithms, that is  $O(n)$ . In real-time data, we cannot predict data pattern and their frequencies. So, a need arises to balance out the existing BST.

Named after their inventor **Adelson, Velski & Landis**, **AVL trees** are height balancing binary search tree. AVL tree checks the height of the left and the right sub-trees and assures that the difference is not more than 1. This difference is called the **Balance Factor**.

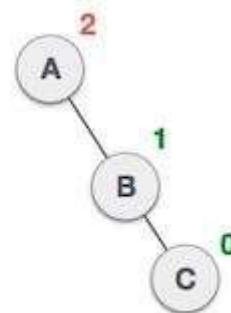
Here we see that the first tree is balanced and the next two trees are not balanced –



Balanced



Not balanced



Not balanced

In the second tree, the left subtree of C has height 2 and the right subtree has height 0, so the difference is 2. In the third tree, the right subtree of A has height 2 and the left is missing, so it is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

$$\text{BalanceFactor} = \text{height}(\text{left-sutree}) - \text{height}(\text{right-sutree})$$

If the difference in the height of left and right sub-trees is more than 1, the tree is balanced using some rotation techniques.

## AVL Rotations

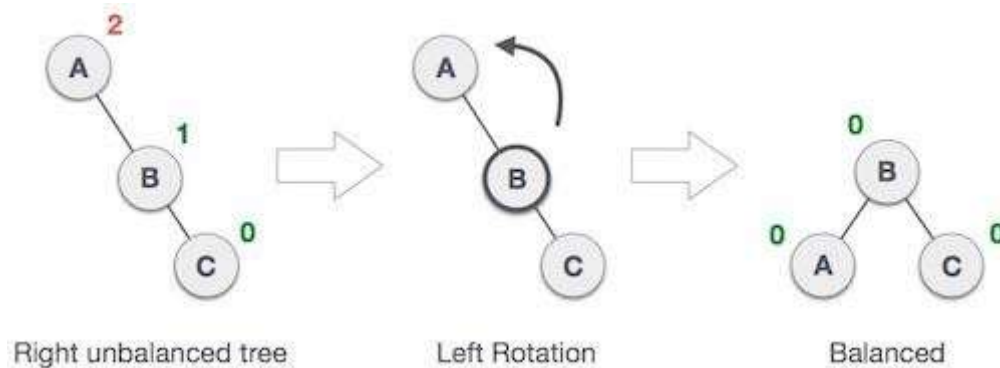
To balance itself, an AVL tree may perform the following four kinds of rotations –

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are single rotations and the next two rotations are double rotations. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

### Left Rotation

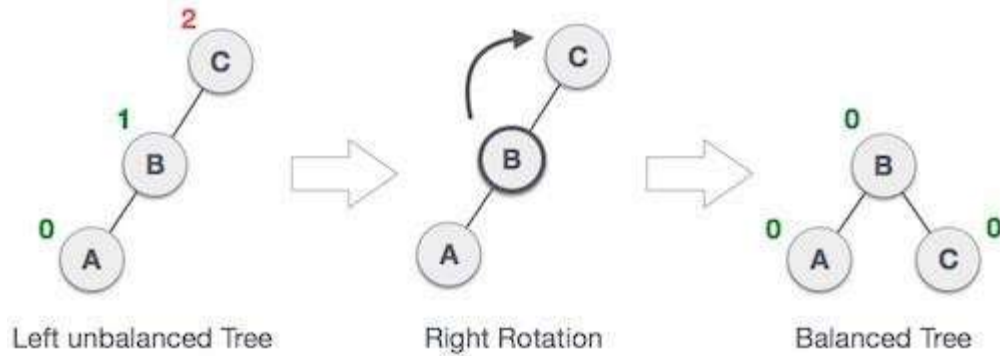
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation –



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

### Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

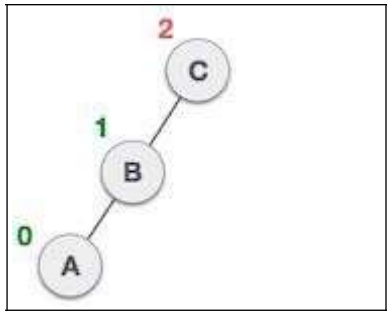
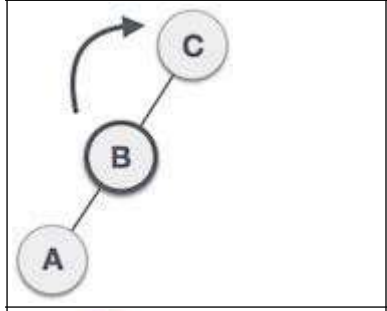
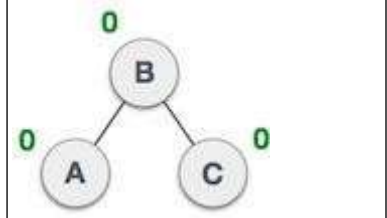


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

### Left-Right Rotation

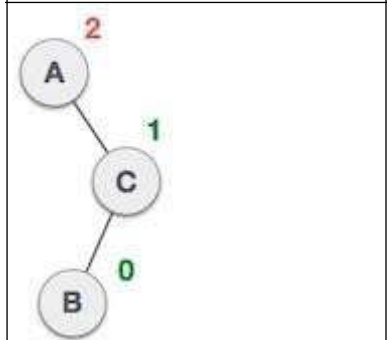
Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

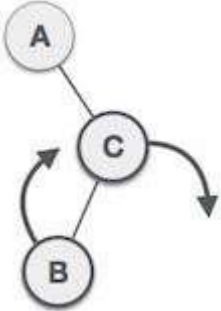
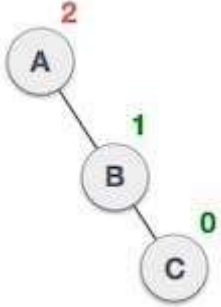
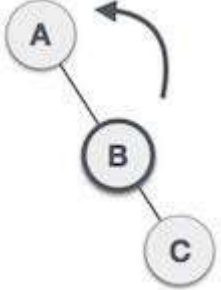
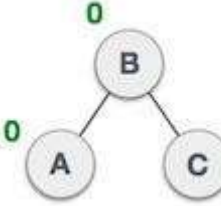
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes C an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.
	We first perform the left rotation on the left subtree of C. This makes A, the left subtree of B.

	<p>Node <b>C</b> is still unbalanced, however now, it is because of the left-subtree of the left-subtree.</p>
	<p>We shall now right-rotate the tree, making <b>B</b> the new root node of this subtree. <b>C</b> now becomes the right subtree of its own left subtree.</p>
	<p>The tree is now balanced.</p>

### Right-Left Rotation

The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation.

State	Action
	<p>A node has been inserted into the left subtree of the right subtree. This makes <b>A</b>, an unbalanced node with balance factor 2.</p>

	<p>First, we perform the right rotation along <b>C</b> node, making <b>C</b> the right subtree of its own left subtree <b>B</b>. Now, <b>B</b> becomes the right subtree of <b>A</b>.</p>
	<p>Node <b>A</b> is still unbalanced because of the right subtree of its right subtree and requires a left rotation.</p>
	<p>A left rotation is performed by making <b>B</b> the new root node of the subtree. <b>A</b> becomes the left subtree of its right subtree <b>B</b>.</p>
	<p>The tree is now balanced.</p>

### Algorithm AVL TREE:

#### Insert:-

1. If P is NULL, then
  - I. P = new node
  - II. P -> element = x
  - III. P -> left = NULL
  - IV. P -> right = NULL
  - V. P -> height = 0
2. else if  $x > P \rightarrow \text{element}$ 
  - a.) insert(x, P -> right)

```

b.) if height of P->left -height of P ->right =2
    1. insert(x, P ->left)
    2. if height(P ->left) -height(P ->right) =2
        if x<P ->left ->element
            P =singlesrotateleft(P)
        else    P =doublerotateleft(P)

3. else
    if x<P ->element
        a.) insert(x, P -> right)
        b.) if height (P -> right) -height (P ->left) =2
            if(x<P ->right) ->element
                P =singlesrotateright(P)
            else    P =doublerotateright(P)

4. else
Print already exists
5. int m, n, d.
6. m = AVL height (P->left)
7. n = AVL height (P->right)
8. d = max(m, n)
9. P->height = d+1
10. Stop

```

#### **RotateWithLeftChild( AvlNode k2 )**

- AvlNode k1 = k2.left;
- k2.left = k1.right;
- k1.right = k2;
- k2.height = max( height( k2.left ), height( k2.right ) ) + 1;
- k1.height = max( height( k1.left ), k2.height ) + 1;
- return k1;

#### **RotateWithRightChild( AvlNode k1 )**

- AvlNode k2 = k1.right;
- k1.right = k2.left;
- k2.left = k1;
- k1.height = max( height( k1.left ), height( k1.right ) ) + 1;
- k2.height = max( height( k2.right ), k1.height ) + 1;
- return k2;

#### **doubleWithLeftChild( AvlNode k3)**

- k3.left = rotateWithRightChild( k3.left );
- return rotateWithLeftChild( k3 );

**doubleWithRightChild( AvlNode k1 )**

- k1.right = rotateWithLeftChild( k1.right );
- return rotateWithRightChild( k1 );

**Software Required:** g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

**Input:** Dictionary word and its meaning.

**Output:** Allow Add, delete operations on dictionary and also display data in sorted order.

**Conclusion:** This program gives us the knowledge height balanced binary tree.

### **OUTCOME**

**Upon completion Students will be able to:**

**ELO1:** Learn height balanced binary tree in data structure.



**ELO2:** Understand & implement rotations required to balance the tree.



### **Questions asked in university exam.**

1. What is AVL tree?
2. In an AVL tree, at what condition the balancing is to be done
3. When would one want to use a balance binary search tree (AVL) rather than an array data structure

### **CODE:**

**Function to get max element:**

```
int getMax(int h1,int h2)
{
    if(h1>h2)
        return h1;
    return h2;
}
```



```

avlnode * dict::rrotate(avlnode *y)
{
cout<<endl<<"rotating right - "<< y->key;
avlnode *x=y->left;
avlnode *t=x->right;
x->right= y;
y->left=t;
y->ht=getmax(getht(y->left),getht(y->right))+1;
x->ht=getmax(getht(x->left),getht(x->right))+1;
return x;
}
avlnode * dict::lrotate(avlnode *x)
{
cout<<endl<<"rotating left - "<< x->key;

avlnode *y=x->right;
avlnode *t=y->left;
x->right= t;
y->left=x;
y->ht=getmax(getht(y->left),getht(y->right))+1;
x->ht=getmax(getht(x->left),getht(x->right))+1;
return y;
}

```

OUTPUT:

```
[@localhost ~]$ ./a.out
```

Menu

```

1.Insert node
2.Inorder Display tree
Enter Choice1

```

```

enter key and meaning(single char)a
a

```

```

enter key and meaning(single char)b
b

```

```

height - 2
node bal - -1 current key is b

```

enter key and  
meaning(single char)cc

height - 2  
node bal - -1  
current key is c  
height - 3  
node bal - -2  
current key is c  
rotating left - a  
enter key and  
meaning(single char)zz

height - 2  
node bal - -1  
current key is z  
height - 3  
node bal - -1 current key is z  
do u want to continue?(1 for continue)1

Menu

1.

In

se

rt

no

de

2.Inorder

Display

treeEnter

Choice2

a a b b c c z z

do u want to continue?(1 for  
continue)0[@localhost ~]\$

\*/

# Experiment No 5

## Title:

Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in a that subject. Use heap data structure. Analyze the algorithm.

## Objectives:

1. To understand concept of heap
2. To understand concept & features like max heap, min heap.

## Learning Objectives:

- ✓ To understand concept of heap
- ✓ To understand concept & features like max heap, min heap.

## Learning Outcome:

- Define class for heap using Object Oriented features.
- Analyze working of functions.

## Theory:

### **Theory:**

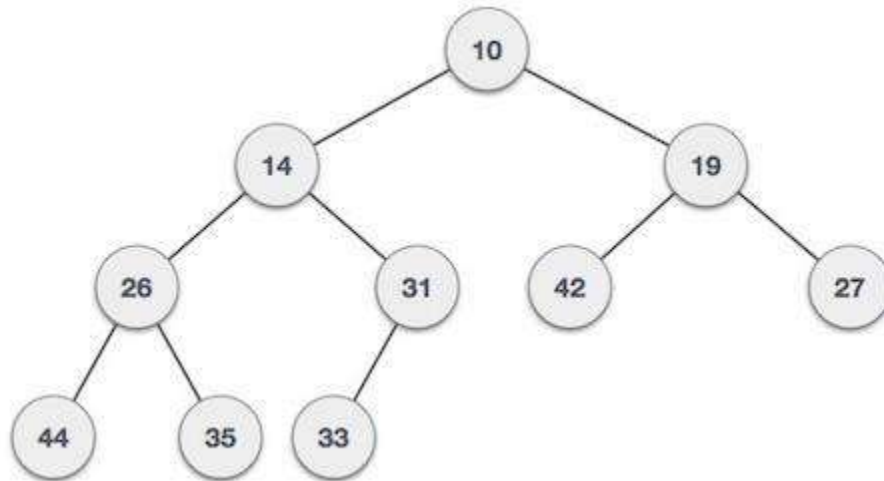
Heap is a special case of balanced binary tree data structure where the root-node key is compared with its children and arranged accordingly. If  $\alpha$  has child node  $\beta$  then –

$$\text{key}(\alpha) \geq \text{key}(\beta)$$

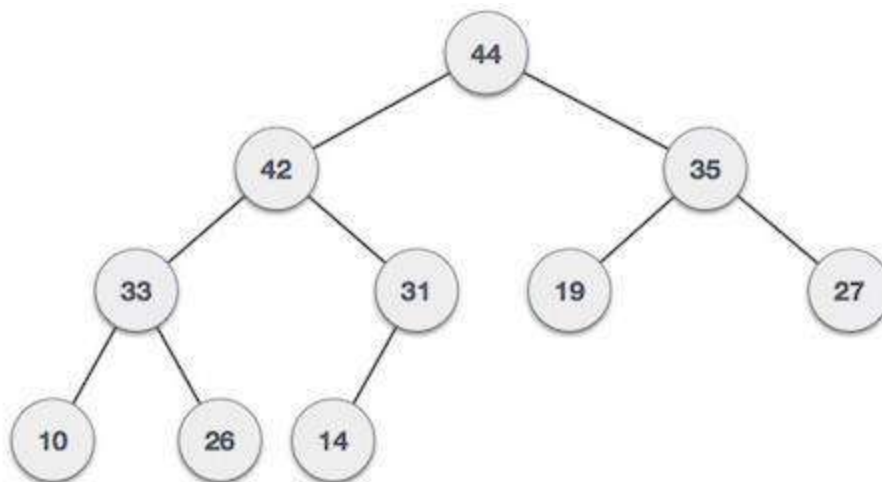
As the value of parent is greater than that of child, this property generates **Max Heap**. Based on this criteria, a heap can be of two types –

For Input  $\rightarrow$  35 33 42 10 14 19 27 44 26 31

**Min-Heap** – Where the value of the root node is less than or equal to either of its children.



**Max-Heap** – Where the value of the root node is greater than or equal to either of its children.



Both trees are constructed using the same input and order of arrival.

### Max Heap Construction Algorithm

We shall use the same example to demonstrate how a Max Heap is created. The procedure to create Min Heap is similar but we go for min values instead of max values.

We are going to derive an algorithm for max heap by inserting one element at a time. At any point of time, heap must maintain its property. While insertion, we also assume that we are inserting a node in an already heapified tree.

**Step 1** – Create a new node at the end of heap.

**Step 2** – Assign new value to the node.

**Step 3** – Compare the value of this child node with its parent.

**Step 4** – If value of parent is less than child, then swap them.

**Step 5** – Repeat step 3 & 4 until Heap property holds.

**Note** – In Min Heap construction algorithm, we expect the value of the parent node to be less than that of the child node.

Let's understand Max Heap construction by an animated illustration. We consider the same input sample that we used earlier.

INPUT:35,33,42,10,14,19,27,44,16,31

### **Max Heap Deletion Algorithm**

Let us derive an algorithm to delete from max heap. Deletion in Max (or Min) Heap always happens at the root to remove the Maximum (or minimum) value.

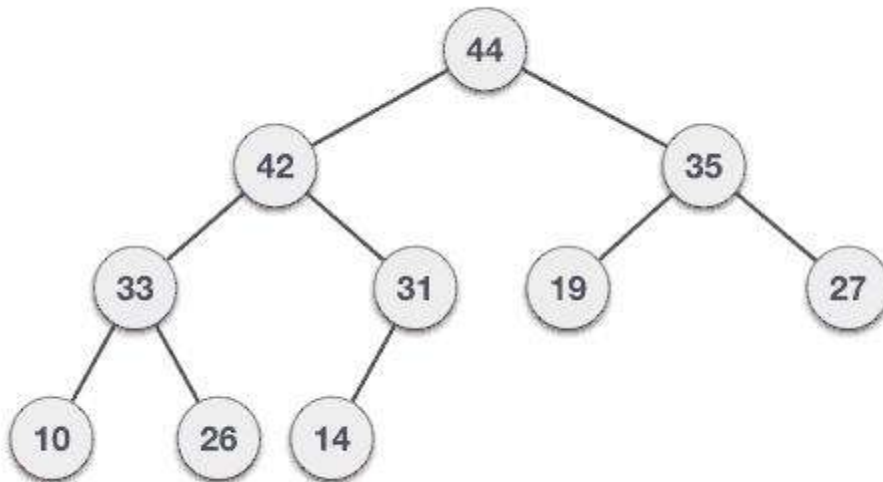
**Step 1** – Remove root node.

**Step 2** – Move the last element of last level to root.

**Step 3** – Compare the value of this child node with its parent.

**Step 4** – If value of parent is less than child, then swap them.

**Step 5** – Repeat step 3 & 4 until Heap property holds.



**Software Required:** g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

**Input:** Marks obtained by student..

**Output:** Find min and max marks obtained.

**Conclusion:** This program gives us the knowledge of heap and its types.

## **OUTCOME**

**Upon completion Students will be able to:**

**ELO1:** Learn object oriented Programming features. 

**ELO2:** Understand & implement Heap data structure.

# Experiment No 6

## **Title:**

Department maintains a student information. The file contains roll number, name, division and address. Allow user to add, delete information of student. Display information of particular employee. If record of student does not exist an appropriate message is displayed. If it is, then the system displays the student details. Use sequential file to main the data.

## **Objectives:**

1. To understand concept of file organization in data structure.
2. To understand concept & features of sequential file organization.

## **Learning Objectives:**

- ✓ To understand concept of file organization in data structure.
- ✓ To understand concept & features of sequential file organization.

## **Learning Outcome:**

- Define class for sequential file using Object Oriented features.
- Analyze working of various operations on sequential file .

## **Theory:**

File organization refers to the relationship of the key of the record to the physical location of that record in the computer file. File organization may be either physical file or a logical file. A physical file is a physical unit, such as magnetic tape or a disk. A logical file on the other hand is a complete set of records for a specific application or purpose. A logical file may occupy a part of physical file or may extend over more than one physical file.

There are various methods of file organizations. These methods may be efficient for certain types of access/selection meanwhile it will turn inefficient for other selections. Hence it is up to the programmer to decide the best suited file organization method depending on his requirement.

Some of the file organizations are

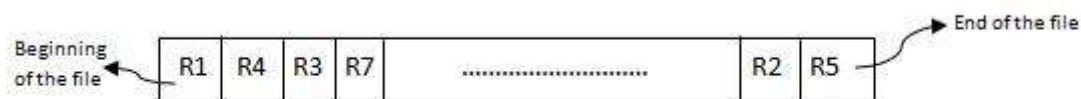
1. Sequential File Organization

2. Heap File Organization
3. Hash/Direct File Organization
4. Indexed Sequential Access Method
5. B+ Tree File Organization
6. Cluster File Organization

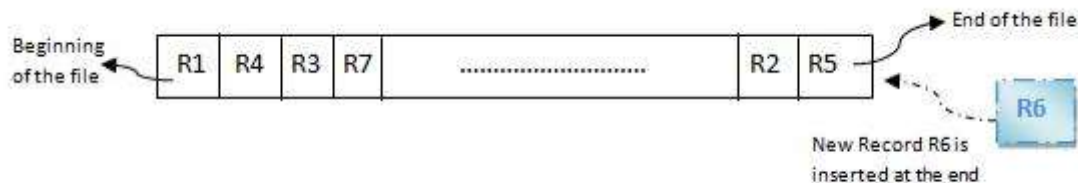
## Sequential File Organization:

It is one of the simple methods of file organization. Here each file/records are stored one after the other in a sequential manner. This can be achieved in two ways:

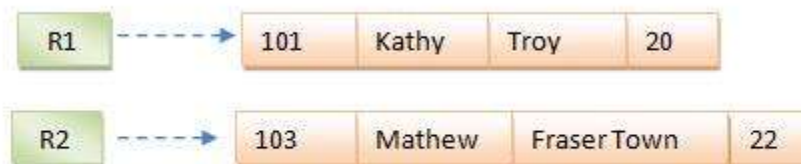
- Records are stored one after the other as they are inserted into the tables. This method is called pile file method. When a new record is inserted, it is placed at the end of the file. In the case of any modification or deletion of record, the record will be searched in the memory blocks. Once it is found, it will be marked for deleting and new block of record is entered.



### Inserting a new record:

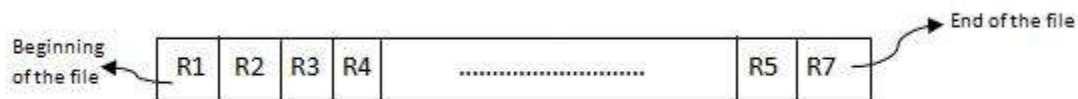


In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2, R3 etc can be considered as one full set of attributes.

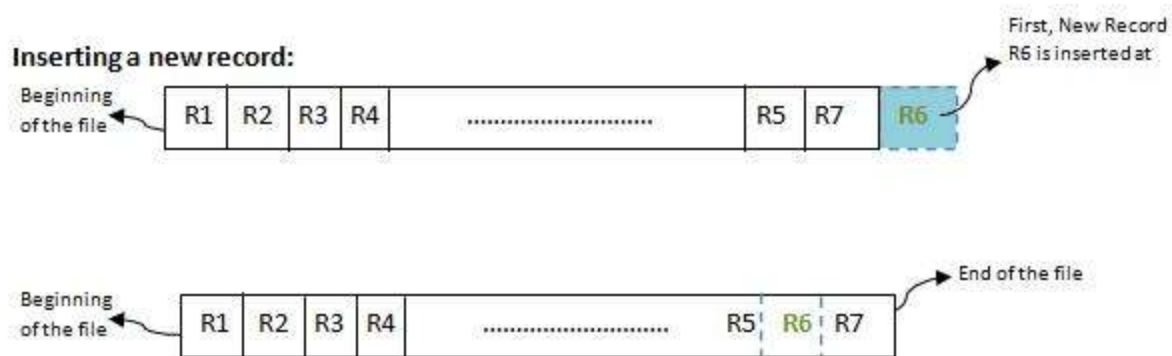




In the second method, records are sorted (either ascending or descending) each time they are inserted into the system. This method is called **sorted file method**. Sorting of records may be based on the primary key or on any other columns. Whenever a new record is inserted, it will be inserted at the end of the file and then it will sort – ascending or descending based on key value and placed at the correct position. In the case of update, it will update the record and then sort the file to place the updated record in the right place. Same is the case with delete.



### Inserting a new record:



### Advantages:

- Simple to understand.
- Easy to maintain and organize
- Loading a record requires only the record key.
- Relatively inexpensive I/O media and devices can be used.
- Easy to reconstruct the files.
- The proportion of file records to be processed is high.

### Disadvantages:

- Entire file must be processed, to get specific information.
- Very low activity rate stored.
- Transactions must be stored and placed in sequence prior to processing.
- Data redundancy is high, as same data can be stored at different places with different keys.

- Impossible to handle random enquiries.

**Software Required:** g++ / gcc compiler- / 64 bit Fedora, eclipse IDE

**Input:** Details of student like roll no, name, address division etc.

**Output:** If record of student does not exist an appropriate message is displayed otherwise the student details are displayed.

**Conclusion:** This program gives us the knowledge sequential file organization..

### **OUTCOME**

**Upon completion Students will be able to:**

**ELO1:** Learn File organization in data structure. 

**ELO2:** Understand & implement sequential file and operation on it. 