Experiment List

Exp. No.	**Title of Experiments**
6	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. Check whether the graph is connected or not. Justify the storage representation used.
7	You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.
8	Given sequence k = k1 <k2 < … <kn of n sorted keys, with a search probability pi for each key ki . Build the Binary search tree that has the least search cost given the access probability for each key?
9	A Dictionary stores keywords and 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
10	Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in that subject. Use heap data structure. Analyze the algorithm.
11	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.
12	Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data.
13	Mini Project

EXPERIMENT NO: 7 (C)

Problem Statement:

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 nodecan 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 representationused.

Objectives:

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

Outcomes:

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

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

- Definition of directed and undirected graph.
- Represent adjacency matrix with one example of directed graph and one example of undirected graph.
- Difference between DFS and BFS
- Definition of graph connectivity with e.g.

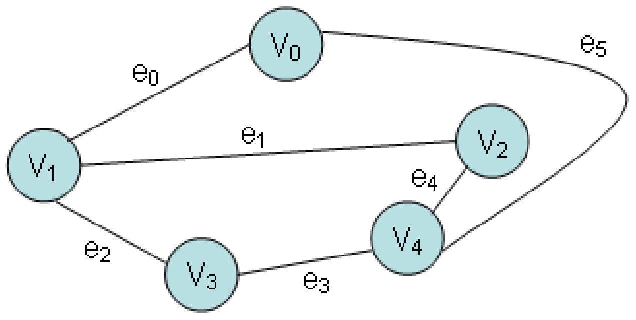
Graphs are the most general data structure. They are also commonly used data structures.

Graph definitions:

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

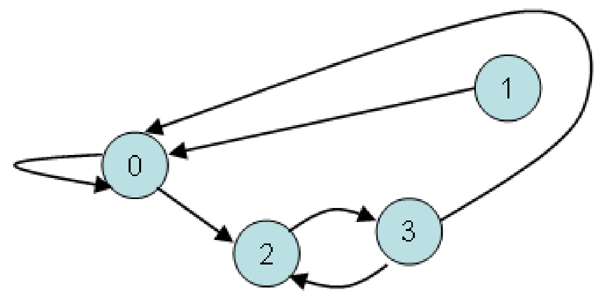
Undirected graph definition:

 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. Bothsets 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.

<u>Representing Graphs with an Adjacency Matrix</u>



$$\begin{array}{c|cccc} & 0 & 1 & 2 & 3 \\ \hline 0 & \text{true} & \text{false} & \text{true} & \text{false} \\ 1 & \text{true} & \text{false} & \text{false} & \text{false} \\ 2 & \text{false} & \text{false} & \text{false} & \text{true} \\ 3 & \text{true} & \text{false} & \text{true} & \text{false} \end{array}$$

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 anedge 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 todetermine which edges are present and which are missing.

<u>Representing Graphs with Edge Lists</u>

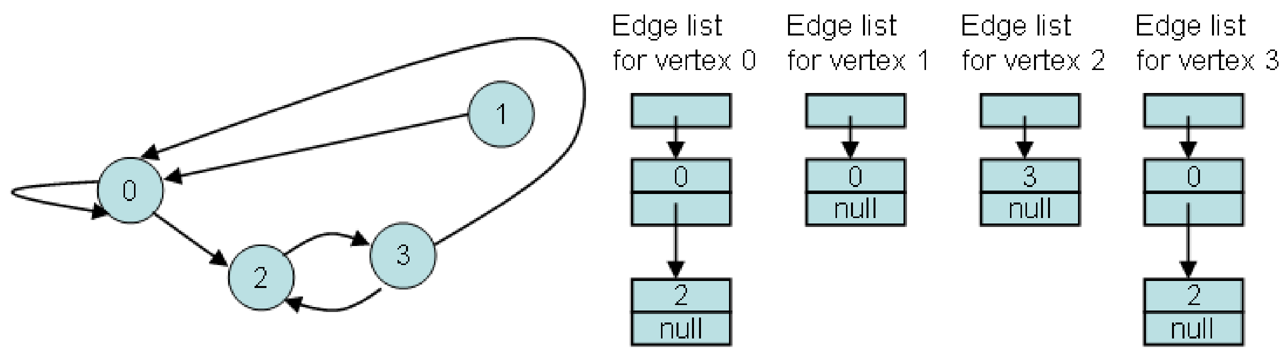


Fig: Graph and adjacency list for each node

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 isconnected.

Conclusion: Many graph applications need to visit the vertices of a graph in some specific order based on the graph's topology. Thus implemented flight path program using different graph primitives.

Oral questions

1. An undirected graph having n edges, then find out no. of vertices that graph have?
2. Define data structure to represent graph.
3. What are the methods to display graph.
4. Where you apply directed and undirected graph?

EXPERIMENT NO: 8 (C)

Problem Statement:

You have a business with several offices; you want to lease phone lines to connect them up with each other; and the phone company charges different amounts of money to connect different pairs of cities. You want a set of lines that connects all your offices with a minimum total cost. Solve the problem by suggesting appropriate data structures.

Objectives:

1. To understand minimum spanning tree of a Graph
2. To understand how Prim‘s algorithm works

Outcomes:

Stduents will be able to understand concept of minimum spanning tree with Prims algorithm.

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

Data structures to be used:

Array: Two dimensional array (adjacency matrix) to store the adjacent vertices & the weights associated edges. One dimensional array to store an indicator for each vertex whether visited or not.

```
#define max 20
int adj_ver[max][max]; int edge_wt[max][max];int ind[max];
```

Concepts to be used:

- Arrays
- Function to construct head list & adjacency matrix for a graph.
- Function to display adjacency matrix of a graph.
- Function to generate minimum spanning tree for a graph using Prim‘s algorithm.

Spanning Tree:

A Spanning Tree of a graph G = (V, E) is a sub graph of G having all vertices of G and no cycles in it.

Minimal Spanning Tree: The cost of a graph is the sum of the costs of the edges in the weighted graph. A spanning tree of a graph G= (V, E) is called minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.

- When a graph G is connected, depth first or breadth first search starting at any vertex visits all the vertices in G.

- The edges of G are partitioned into two sets i.e. T for the tree edges & B for back edges. T is the set of tree edges and B for back edges. T is the set of edges used or traversed during the search & B is the set of remaining edges.

- The edges of G in T form a tree which includes all the vertices of graph G and this tree is called as spanning tree.

  Definition: Any tree, which consists solely of edges in graph G and includes all the vertices in G, is called as spanning tree. Thus for a given connected graph there are multiple spanning trees possible. For maximal connected graph having _n' vertices the number of different possible spanning trees is equal to n.
  Cycle: If any edge from set B of graph G is introduced into the corresponding spanning tree T of graph G then cycle is formed. This cycle consists of edge (v, w) from the set B and all edges on the path from w to v in T.

- **Prim's algorithm:**
  Prim's algorithm is an algorithm in graph theory 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. The algorithm was discovered in 1930 by mathematician Vojtech Jarník and later independently by computer scientist Robert C. Prim in 1957 and rediscovered by Edsger Dijkstra in 1959. Therefore it is sometimes called the DJP algorithm, the Jarník algorithm, or the Prim-Jarník algorithm.

  An arbitrary node is chosen initially as the tree root (any node can be considered as root node for a graph). The nodes of the graph are then appended to the tree one at a time until all nodes of the graph are included. The node of the graph added to the tree at each pointis that node adjacent to a node of the tree by an arc of minimum weight. The arc of minimum weight becomes a tree arc containing the new node to tree. When all the nodesof the graph have been added to the tree, a minimum spanning tree has been constructed for the graph.

Algorithm / Pseudo code:

- **Prim's Algorithm:**
  All vertices of a connected graph are included in the minimum spanning tree. Prim's algorithm starts from one vertex and grows the rest of tree by adding one vertex at a time by adding associated edge in T. This algorithm iteratively adds edges until all vertices are visited.

```
void prims (vertex i)
1. Start
2. Initialize visited [ ] to 0for (i=0;i<n; i++)
   visited [ i ] = 0;
3. Find minimum edge from ifor (j=0;j<n; j++)
   {
   if (min > a [i] [j])
   {
   min = a[i] [j]x = i;
   y = j;
   }
   }
4. Print the edge between i and j with weight.
```

5. Make visit [i++] = x

visit [j++] = y

6. Find next minimum edge starting from nodes of visit array.
7. Repeat step 6 until all the nodes are visited.
8. End.

Conclusion: Thus we have studied and implemented minimum spanning tree concept with the help of Prims Algorithm.

EXPERIMENT NO: 9 (D)

Problem Statement:

Given sequence k = k1 <k2 < … <kn of n sorted keys, with a search probability pi for each key ki . Build the Binary search tree that has the least search cost given the access probability for each key?

Objectives:

1. To understand concept of OBST.
2. To understand concept & features like extended binary search tree.

Outcomes:

1. Define class for Extended binary search tree using Object Oriented features.
2. Analyze working of functions.

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

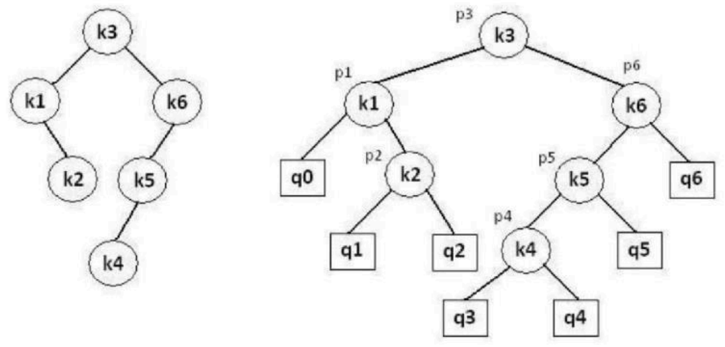
Theory:-

- Explain Dynamic Programming
- Need of weight balanced tree
- Explain what is OBST with example

An optimal binary search tree is a binary search tree for which the nodes are arranged on levels such that the tree cost is minimum.

For the purpose of a better presentation of optimal binary search trees, we will consider "extended binary search trees", which have the keys stored at their internal nodes. Suppose "n" keys k1, k2, … k n are stored at the internal nodes of a binary search tree. It is assumed that the keys are given in sorted order, so that k1< k2 < … < kn.

An extended binary search tree is obtained from the binary search tree by adding successor nodes to each of its terminal nodes as indicated in the following figure by squares:



Binary search tree Extended binary search tree

In the extended tree:

- The squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key values that are not actually stored in the tree;
- The round nodes represent internal nodes; these are the actual keys stored in the tree;
- Assuming that the relative frequency with which each key value is accessed is known, weights can be assigned to each node of the extended tree (p1 … p6). They represent the relative frequencies of searches terminating at each node, that is, they mark the successful searches.
- If the user searches a particular key in the tree, 2 cases can occur:
- 1 – the key is found, so the corresponding weight „p" is incremented;
- 2 – the key is not found, so the corresponding „q" value is incremented.

GENERALIZATION:

The terminal node in the extended tree that is the left successor of k1 can be interpreted as representing all key values that are not stored and are less than k1. Similarly, the terminal node in the extended tree that is the right successor of kn, represents all key values not stored in the tree that are greater than kn. The terminal node that is successes between ki and ki-1 in an inorder traversal represent all key values not stored that lie between ki and ki - 1.

ALGORITHMS

We have the following procedure for determining R(i, j) and C(i, j) with 0 <= i <= j <= n:

```
PROCEDURE COMPUTE_ROOT(n, p, q; R, C)
begin
for i = 0 to n doC (i, i) ⟵0
W (i, i) ⟵q(i) for m = 0 to n do
for i = 0 to (n – m) doj ⟵i + m
W (i, j) ⟵W (i, j – 1) + p (j) + q (j)
*find C (i, j) and R (i, j) which minimize thetree cost
end
```

The following function builds an optimal binary search tree
FUNCTION CONSTRUCT(R, i, j)

```
begin
*build a new internal node N labeled (i, j)k ←R (i,
j) f i = k then
*build a new leaf node N‟ labeled (i, i)else
*N‟ ←CONSTRUCT(R, i, k)
*N‟ is the left child of node Nif k = (j – 1) then
*build a new leaf node N‟‟ labeled (j, j)else
*N‟‟ ←CONSTRUCT(R, k + 1, j)
*N‟‟ is the right child of node Nreturn
N end
```

COMPLEXITY ANALYSIS:

The algorithm requires O (n2) time and O (n2) storage. Therefore, as „n‟ increases it will run outof storage even before it runs out of time. The storage needed can be reduced by almost half by implementing the two-dimensional arrays as one-dimensional arrays.

Conclusion: This lab assignment gave us the knowledge of how to develop OBST using c++ programming language.

Oral questions

1. Define OBST, Dynamic programming
2. What is need of weight balanced tree
3. Applications of OBST

EXPERIMENT NO: 10 (D)

Problem Statement:

A Dictionary stores keywords and 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.

Outcomes:

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

Software Requirements:

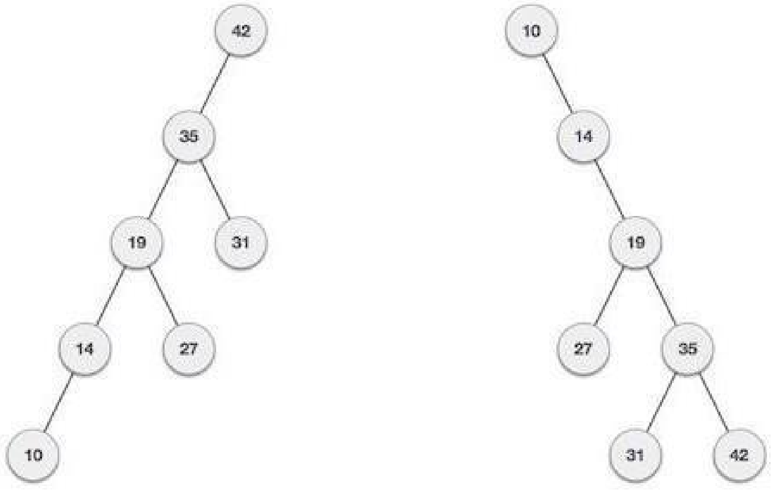
1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

- Need of height balanced tree.
- What is AVL tree, balance factor, all rotations

An empty tree is height balanced tree if T is a nonempty binary tree with TL and TR asits 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 searchtime, 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 theleft 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 –

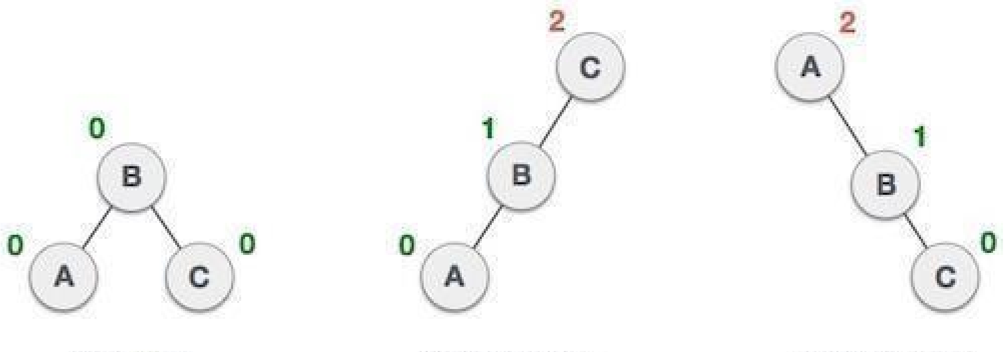


If input 'appears' non-increasing manner If input 'appears' in non-decreasing manner

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, soit is 0, and the difference is 2 again. AVL tree permits difference (balance factor) to be only 1.

BalanceFactor = height(left-sutree) – height(right-sutree)

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

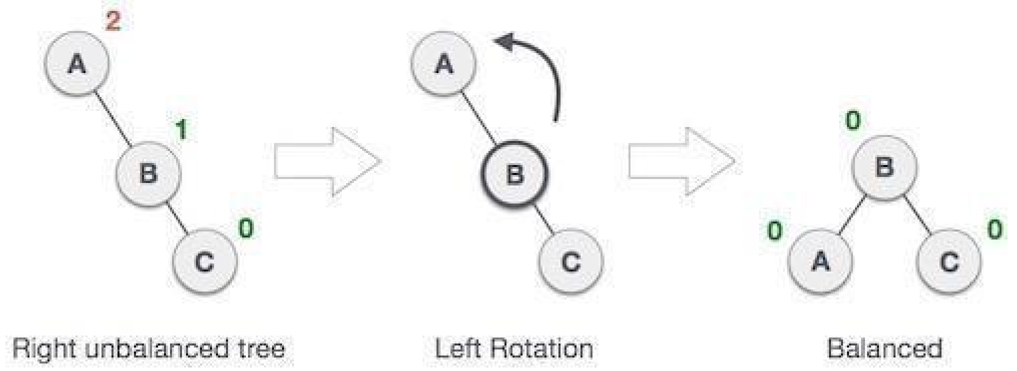
AVL Rotations

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

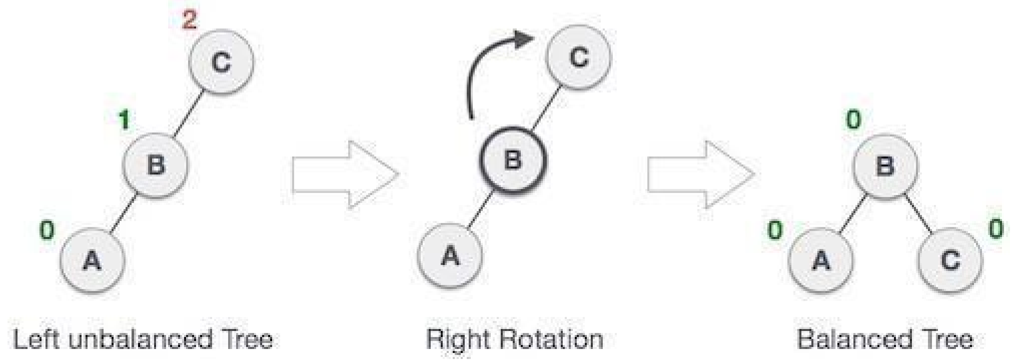
- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

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 subtreeof A's right subtree. We perform the left rotation by making **A** the left-subtree of B.



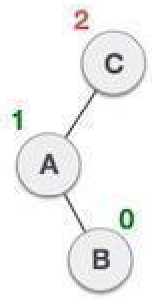
Right Rotation
AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

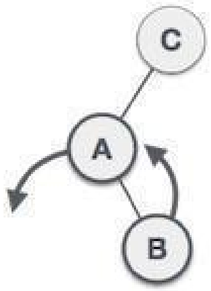
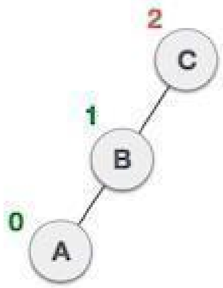
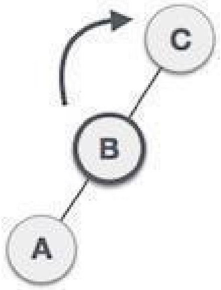


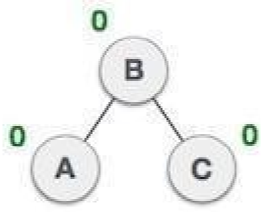
As depicted, the unbalanced node becomes the right child of its left child by performing a rightrotation.

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

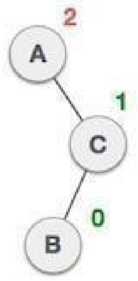
first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

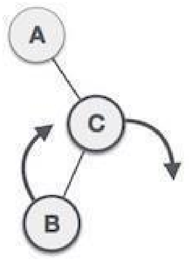
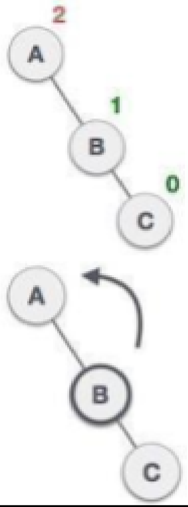
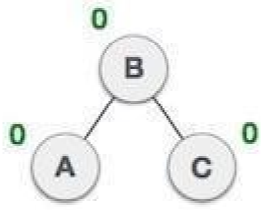
State	Action
	A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL treeto perform left-right rotation.

	We first perform the left rotation on the left subtree of **C**. Thismakes **A**, the left subtree of **B**.
	Node **C** is still unbalanced, however now, it is because of the left-subtree of the left-subtree.
	We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

	The tree is now balanced.

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
	A node has been inserted into the left subtree of the right subtree.This makes **A**, an unbalanced node with balance factor 2.

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

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>1 => x<P -
    >elementa.) insert(x, P ->left)

    b.) if height of P->left -height of P ->right =2
1. insert(x, P ->left)
2. if height(P ->left) -height(P ->right) =2if x<P ->left
    ->element P =singlerotateleft(P)
                    else
                                    P =doublerotateleft(P)
3. else     if x<P -
            >element
  a.) insert(x, P -> right)
  b.)     if height (P -> right) -height (P ->left) =2if(x<P
  ->right)->element P =singlerotateright(P)
  else
  P =doublerotateright(P)
  4.    else
  Print already exits
  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);

Conclusion: Thus we have studied and implemented concept of AVL Tree with its operations.

EXPERIMENT NO: 11 (E)

Problem Statement:

Read the marks obtained by students of second year in an online examination of particular subject. Find out maximum and minimum marks obtained in 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.

Outcomes:

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

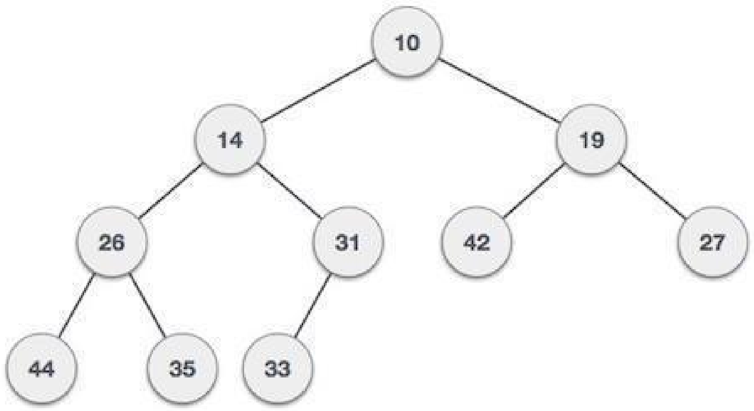
Software Requirements:

4. 64-bit Open source Linux or its derivative
5. Open Source C++ Programming tool like G++/GCC.
6. Turbo C++ compiler

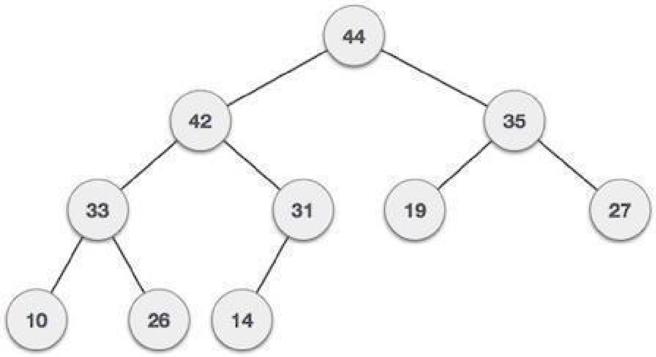
Theory:-

- Explain Max heap , e.g. and its array representation
- Explain Min heap , e.g. and its array representation
- How heap sort is appropriate to find minimum and maximum marks obtained in particular subject

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 **α** has child node **β** then −
key(α) ≥ key(β)
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 → 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 inputsample that we used earlier.

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

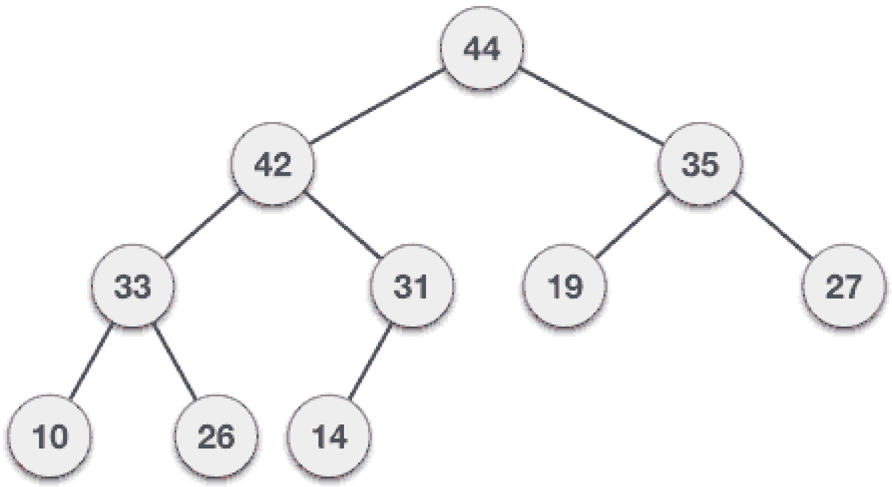
Max Heap Deletion Algorithm

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.



Conclusion: Thus we have studied and implemented concept of heap data structure using Object Oriented features.

EXPERIMENT NO: 12 (F)

Problem Statement:

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.

Outcomes:

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

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

File organization refers to the relationship of the key of the record to the physical locationof 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 handis 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.

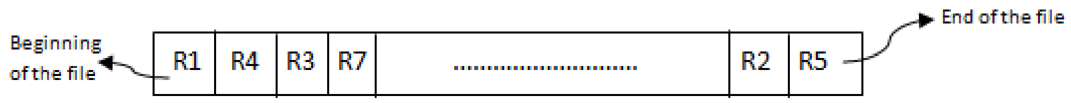
Some of the file organizations are

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

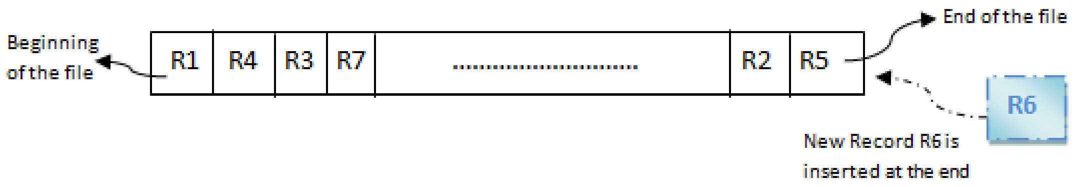
Sequential File Organization:

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:

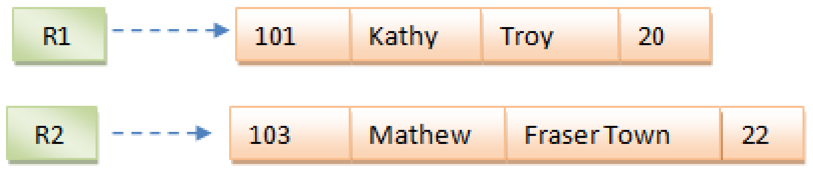
 Records are stored one after the other as they are inserted into the tables. This method is called pile file method. When a new record is inserted, it is placed at the end of the file.In the case of any modification or deletion of record, the record will be searched in the memory blocks. Once it is found,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



Inserting a new record:

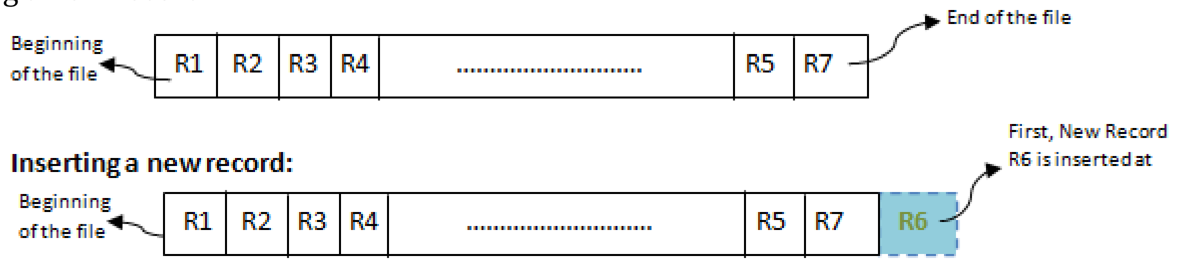


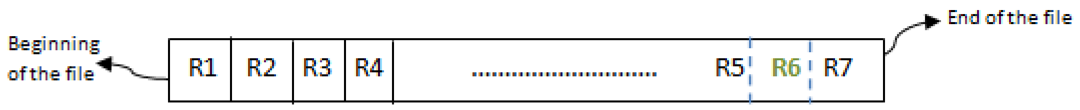
In the diagram above, R1, R2, R3 etc are the records. They contain all the attribute of a row. i.e.; when we say student record, it will have his id, name, address, course, DOB etc. Similarly R1, R2, R3 etc can be considered as one full set of attributes.



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 thefile to place the updated record in the right place. Same is the case with delete.

Inserting a new record:





Advantages:

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

Disadvantages:

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

Conclusion: Thus we have studied and implemented concept of file organization in data structure

EXPERIMENT NO: 13 (F)

Problem Statement:

Company maintains employee information as employee ID, name, designation and salary. Allow user to add, delete information of employee. Display information of particular employee. If employee does not exist an appropriate message is displayed. If it is, then the system displays the employee details. Use index sequential file to maintain the data.

Objectives:

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

Outcomes:

Student will be able to understand concept of file organization in data structure.

Software Requirements:

1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC.
3. Turbo C++ compiler

Theory:-

A *file* is the basic entity for permanent retention of data in secondary storage devices such as hard drives, CDs, DVDs, flash drives, and tapes. But, the problem is that the common file imposes no structure of data storage and is not fit for high level-data processing. Files store a sequence of data as raw data bytes.

In C++, typically a file is opened if it already exists or created as an object associated to the stream is created. The most common stream objects in C++ are: *cin*, *cout*, *cerr*, and *clog*. These objects are created as we include the *iostream* header into a program. The *cin* object associates with the standard input stream, *cout* with the standard output stream, and *cerr* and *clog* with the standard error stream.

File Processing: There are two indispensable headers that must be included to perform file processing in C++: <*iostream*> and <*fstream*>. There are three important stream class templates in the <*fstream*> header: the *basic_ifstream* for performing file input, *basic_ofstream* for file output, and *basic_fstream* for both file input and output. .

Note that the *ifstream*, *ofstream*, and *fstream* objects that we often use ina C++ program is nothing but a specialization of *basic_ifstream*, *basic_ofstream*, and *basic_fstream*, respectively. The <*fstream*> library provides theses *typedef* aliases for the basic template specialization classes to provide convenience by performing character I/O to and from files.

Sequential Files: A file created with the help of C++ standard library functions does not impose any structure on how the data is to be persisted. However, we are able to impose structure programmatically according to the application requirement. Suppose a minimal “payroll” application stores an employee record in a sequential file as employee id, name of the employee, and salary. The data obtained for each employee constitutes a record. The records are stored and written to the file sequentially and retrieved or read from the file in the same manner.

Opening a File: Here, we have opened the file using an instance of the *ofstream* class using two arguments: filename and opening mode. A file can be opened in different modes, as shown in the following table.

File Opening mode	***Description***
ios::in	Open file for reading data.
ios::out	Creates file if it does not exist. All existing data is erased from the file as new data is written into the file.
ios::app	Creates file if it does not exist. New data is written at the end of the file without disturbing the existing data.
ios::ate	Open file for output. Data can be written anywhere in the file, similar to append.
ios::trunc	Discard file content, which is the default action for ios::out.
ios::binary	Open file in binary (non-text) mode.

There is an *open* function that also can be used to open a file and set the mode. This is particularly useful when we first create an *ofstream* object and then open it as follows:

```
ofstream ofile;          ofile.open(filename, ios::out);
```

After creating an *ofstream* or *ifstream* object, it typically is checked whether it has successfully opened or not. We do this with the following code:

```
if(!fin){ cerr<<"Cannot open file for writing"<<endl;          exit(EXIT_FAILURE); }
```

Data Processing

Data processing occurs after a file has been opened successfully. The stream insertion (>>) and extraction (<<) operators are overloaded for convenient writing and reading data to and from a file. The file we have created is a simple text file and can be viewed by any text editor.

Closing a File:A file opened must be closed. In fact, as we invoke the *close* operation associated with the *ofstream* or *ifstream* object, it invokes the destructor, which closes the file. The operation is automatically invoked as the stream object goes out of scope. As a result, an explicit invocation of the *close* function is optional, but still is good programming practice.

Conclusion: Thus we have studied and implemented concept of index sequential file organization.

EXPERIMENT NO: 14

Problem Statement: Mini Project

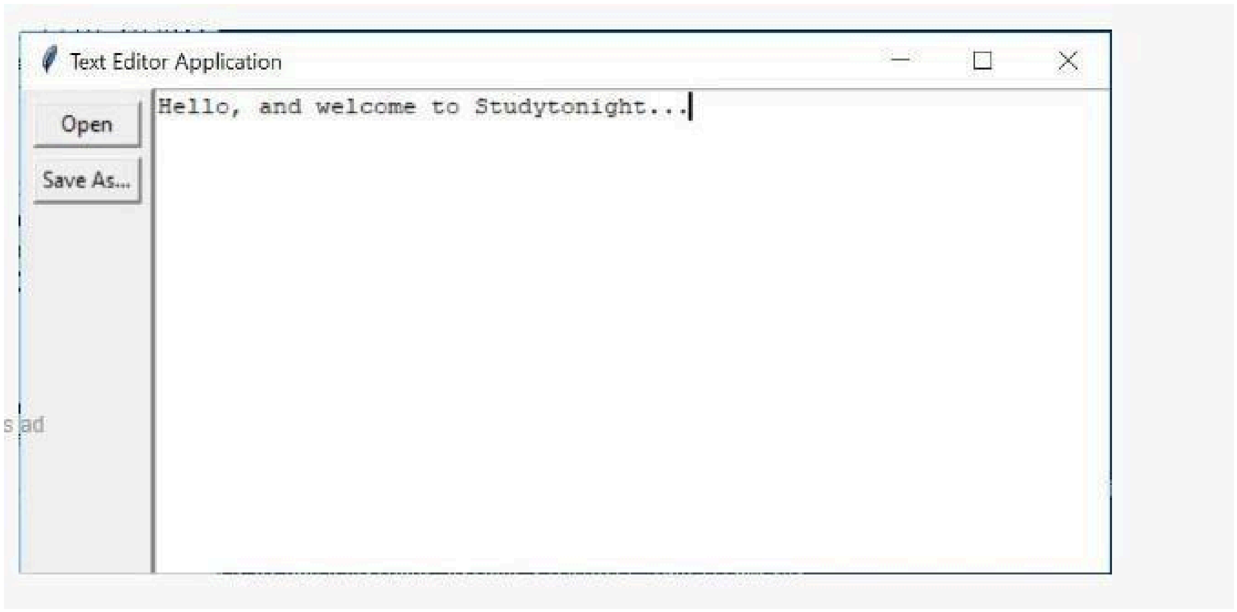
SMART TEXT EDITOR USING PYTHON:

```
import tkinter as tk
from tkinter.filedialog import askopenfilename, asksaveasfilename

def open_file():
    """Open a file for editing."""
    filepath = askopenfilename(
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")]
    )
    if not filepath:
        return
    txt_edit.delete(1.0, tk.END)
    with open(filepath, "r") as input_file:
        text = input_file.read()
        txt_edit.insert(tk.END, text)
    window.title(f"Text Editor Application - {filepath}")

def save_file():
    """Save the current file as a new file."""
    filepath = asksaveasfilename(
        defaultextension="txt",
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")],
    )
    if not filepath:
        return
    with open(filepath, "w") as output_file:
        text = txt_edit.get(1.0, tk.END)
        output_file.write(text)
    window.title(f"Text Editor Application - {filepath}")

window = tk.Tk()
window.title("Text Editor Application")
window.rowconfigure(0, minsize=800, weight=1)
window.columnconfigure(1, minsize=800, weight=1)

txt_edit = tk.Text(window)
fr_buttons = tk.Frame(window, relief=tk.RAISED, bd=2)
btn_open = tk.Button(fr_buttons, text="Open", command=open_file)
btn_save = tk.Button(fr_buttons, text="Save As...", command=save_file)

btn_open.grid(row=0, column=0, sticky="ew", padx=5, pady=5)
```

```
    btn_save.grid(row=1, column=0, sticky="ew", padx=5)
```

```
fr_buttons.grid(row=0, column=0, sticky="ns")
txt_edit.grid(row=0, column=1, sticky="nsew")

window.mainloop()
```

OUTPUT:



```
Hello, and welcome to Studytonight...
```