Dr. D. Y. Patil Pratishthan's
DR. D. Y. PATIL INSTITUTE OF ENGINEERING, MANAGEMENT & RESEARCH

Approved by A.I.C.T.E, New Delhi , Maharashtra State Government, Affiliated to Savitribai Phule Pune University

Sector No. 29, PCNTDA , Nigidi Pradhikaran, Akurdi, Pune 411044. Phone: 020–27654470, Fax: 020-27656566
Website : www.dypiemr.ac.in Email : principal.dypiemr@gmail.com

---

DEPARTMENT
OF
COMPUTER ENGINEERING

LAB MANUAL
Data Structures and Algorithms Laboratory
(Second Year Engineering)
AY 2023-24 Semester – II

Prepared by :

Ms. Shritika Waykar

Mrs. Akanksha Kulkarni

Mr. Prateek Meshram

Table of Contents

Sr.No		Title of the Experiment	Page No
		Group A	
1	A1	Make use of a hash table implementation to quickly look up client‘s telephone number.	3
2	A2	For given set of elements create skip list.	5
		Group B	
3	B1	Construct binary search tree by inserting the values in the order given.	8
4	B2	Construct an expression tree from the given prefix expression.	12
5	B3	Convert given binary tree into threaded binary tree.	15
		Group C	
6	C1	Represent flight paths as a graph. Use adjacency list or adjacency matrix representation of the graph.	18
7	C2	Implement Minimum Spanning Tree to set a telephone lines that connects all offices with a minimum total cost.	
		Group D	
8	D1	Build the Binary search tree that has the least search cost given the access probability for each key.	
9	D2	Use Height balance tree to implement Dictionary which stores keywords and meanings and find the complexity for finding a keyword	
		Group E	
10	E1	Read the marks obtained by students and find out maximum and minimum marks. Use heap.	
		Group F	
11	F1	Department maintains a student information. Use sequential file to main the data.	
12	F2	Company maintains employee information. Use index sequential file to maintain the data.	
		Mini Project	
13		• Design a mini project using JAVA which will use the different data structure with or without Java collection library and show the use of specific data structure on the efficiency (performance) of the code. • Design a mini project to implement Snake and Ladders Game using python. • Design a mini project to implement a Smart text editor. • Design a mini project for automated Term work assessment of student based on parameters like daily attendance, Unit Test / Prelim performance, Students achievements if any, Mock Practical.	

Assignment No 1 - A1

Problem Statement:

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

Objective:

To understand the basic concept of Hashing in Data structure.

Outcome:

To implement the basic concept of Hashing, to store a set of telephone numbers , use of concept of hashing to quickly look up in N number of Data.

Software & Hardware Requirements:

1. 64-bit Open source Linux or its derivative
2. Python

Theory Concepts:

Hashing:

- Suppose we want to design a system for storing employee records keyed using phone numbers. And we want following queries to be performed efficiently:
  Insert a phone number and corresponding information.
  Search a phone number and fetch the information.
  Delete a phone number and related information.
- We can think of using the following data structures to maintain information about different phone numbers.
  Array of phone numbers and records.
  Linked List of phone numbers and records.
  Balanced binary search tree with phone numbers as keys.
  Direct Access Table.
- For **arrays and linked lists**, we need to search in a linear fashion, which can be costly in practice. If we use arrays and keep the data sorted, then a phone number can be searched in O(Logn) time using Binary Search, but insert and delete operations become costly as we have to maintain sorted order.
- With **balanced binary search tree**, we get moderate search, insert and delete times. All of these operations can be guaranteed to be in O(Logn) time.
- Another solution that one can think of is to use a **direct access table** where we make a big array and use phone numbers as index in the array.
- An entry in array is NIL if phone number is not. present, else the array entry stores pointer to records corresponding to phone number.
- Time complexity wise this solution is the best among all, we can do all operations in O(1) time. For example to insert a phone number, we create a record with details of given phone number, use phone number as index and store the pointer to the created record in table.
- This solution has many practical limitations.
- First problem with this solution is extra space required is huge.
- For example if phone number is n digits, we need $O(m * 10^n)$ space for table where m is size of a pointer to record.
- Another problem is an integer in a programming language may not store n digits.
- Due to above limitations Direct Access Table cannot always be used. **Hashing** is the

solution that can be used in almost all such situations and performs extremely well compared to above data structures like Array, Linked List, Balanced BST in practice. With hashing we get O(1) search time on average (under reasonable assumptions) and O(n) in worst case.

Hashing is an improvement over Direct Access Table. The idea is to use hash function that converts a given phone number or any other key to a smaller number and uses the small number as index in a table called hash table.

Hash Function:

- A function that converts a given big phone number to a small practical integer value. The mapped integer value is used as an index in hash table.
- In simple terms, a hash function maps a big number or string to a small integer that can be used as index in hash table.
- A good hash function should have following properties
  Efficiently computable.
  Should uniformly distribute the keys (Each table position equally likely for each key)
- For example for phone numbers a bad hash function is to take first three digits. A better function is consider last three digits. Please note that this may not be the best hash function. There may be better ways.

Hash Table:

- An array that stores pointers to records corresponding to a given phone number. An entry in hash table is NIL if no existing phone number has hash function value equal to the index for the entry.

Collision Handling:

- Since a hash function gets us a small number for a big key, there is possibility that two keys result in same value. The situation where a newly inserted key maps to an already occupied slot in hash table is called collision and must be handled using some collision handling technique. Following are the ways to handle collisions:
- **Chaining:**The idea is to make each cell of hash table point to a linked list of records that have same hash function value. Chaining is simple, but requires additional memory outside the table.
- **Open Addressing:** In open addressing, all elements are stored in the hash table itself. Each table entry contains either a record or NIL. When searching for an element, we one by one examine table slots until the desired element is found or it is clear that the element is not in the table.

Conclusion:

We are able to implement the concept of Hashing in programming.

Assignment No 2 - A2

Problem Statement:

For given set of elements create skip list. Find the element in the set that is closest to some given value. (note: Decide the level of element in the list Randomly with some upper limit)

Objective:

To understand the basic concept of Skip List in Data structure.

Outcome:

To implement the Skip list as a better searching over linked list.

Software & Hardware Requirements:

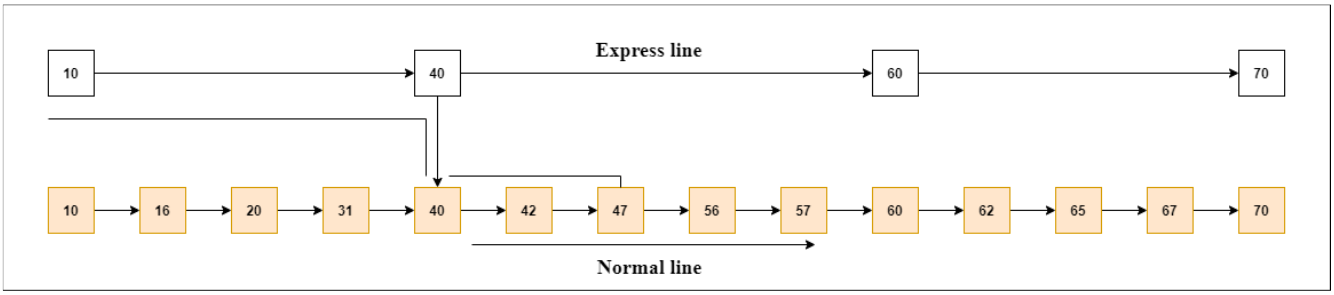
1. 64-bit Open source Linux or its derivative
2. Python

Theory Concepts:

- A skip list is a probabilistic data structure.
- The skip list is used to store a sorted list of elements or data with a linked list.
- It allows the process of the elements or data to view efficiently. In one single step, it skips several elements of the entire list, which is why it is known as a skip list.
- The skip list is an extended version of the linked list.
- It allows the user to search, remove, and insert the element very quickly.
- It consists of a base list that includes a set of elements which maintains the link hierarchy of the subsequent elements.

Skip list structure:

- It is built in two layers: The lowest layer and Top layer. The lowest layer of the skip list is a common sorted linked list, and the top layers of the skip list are like an "express line" where the elements are skipped.
- Let's take an example to understand the working of the skip list. In this example, we have 14 nodes, such that these nodes are divided into two layers, as shown in the diagram.
- The lower layer is a common line that links all nodes, and the top layer is an express line that links only the main nodes, as you can see in the diagram.
- Suppose you want to find 47 in this example. You will start the search from the first node of the express line and continue running on the express line until you find a node that is equal a 47 or more than 47.
- You can see in the example that 47 does not exist in the express line, so you search for a node of less than 47, which is 40. Now, you go to the normal line with the help of 40, and search the 47, as shown in the diagram.



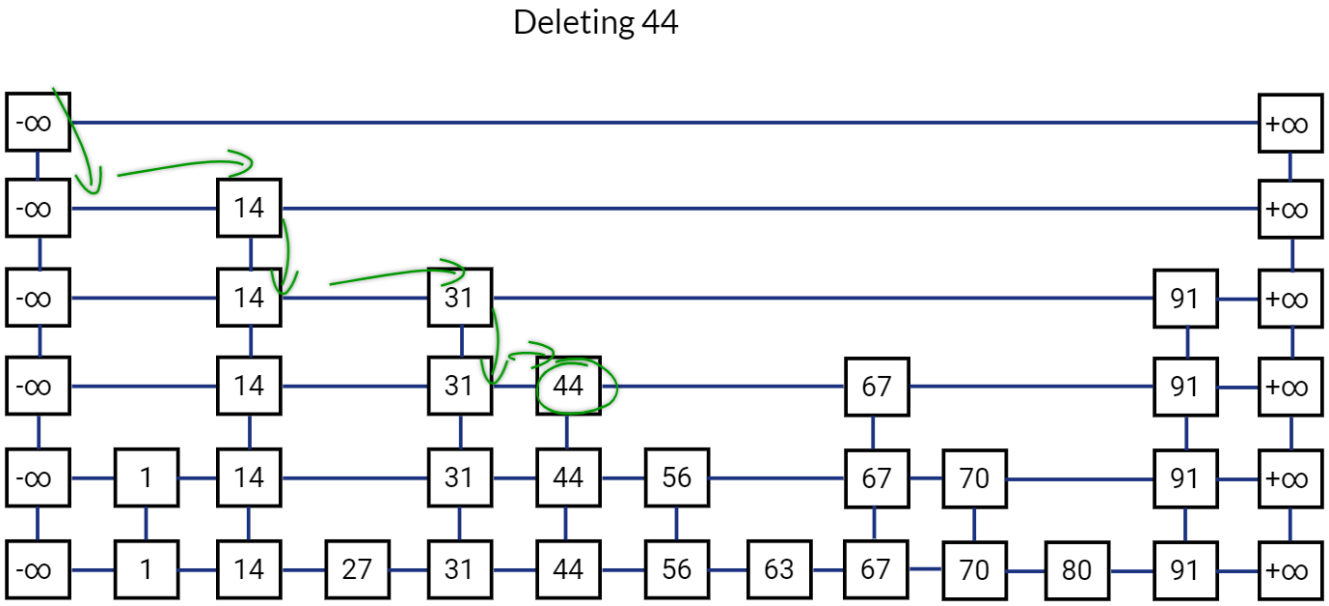
Skip List Basic Operations

There are the following types of operations in the skip list.

1. Insertion operation: It is used to add a new node to a particular location in a specific situation.

2. Deletion operation: It is used to delete a node in a specific situation.

3. Se**arch Operation**: The search operation is used to search a particular node in a skip list.



Advantages of Skip List:

- If you want to insert a new node in the skip list, then it will insert the node very fast because there are no rotations in the skip list.
- The skip list is simple to implement as compared to the hash table and the binary search tree.
- It is very simple to find a node in the list because it stores the nodes in sorted form.
- The skip list algorithm can be modified very easily in a more specific structure, such as indexable skip lists, trees, or priority queues.
- The skip list is a robust and reliable list.

Disadvantages of Skip List

- It requires more memory than the balanced tree.
- Reverse searching is not allowed.
- The skip list searches the node much slower than the linked list.

Applications of Skip List

- Skip list are used in distributed applications. In distributed systems, the nodes of skip list represents the computer systems and pointers represent network connection.
- Skip list are used for implementing highly scalable concurrent priority queues with less lock contention (struggle for having a lock on a data item) https://www.geeksforgeeks.org/implementation-of-locking-indbms/.

- 3. It is also used with the QMap template class. (Value-based template class that provides a dictionary)
- 4. The indexing of the skip list is used in running median problems.
- 5. skipdb is an open-source database format using ordered key/value pairs.

Conclusion:
Thus we have studied Skip list and its advantages and applications.

Assignment No 3 – B1

Problem Statement:

Beginning with an empty binary search tree, Construct binary search tree by inserting the values in the order given. After constructing a binary tree –

- Insert new node
- Find number of nodes in longest path.
- Minimum data value found in the tree
- Change a tree so that the roles of the left and right pointers are swapped at every node
- Search a value

Objective:

To understand the basic concept of Non Linear Data Structure "TREE "and its basic operation in Data structure.

Outcome:

To implement the basic concept of Binary Search Tree to store a numbers in it. Also perform basic Operation Insert, Delete and search,Traverse in tree in Data structure.

Software & Hardware Requirements:

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

Theory –

Tree represents the nodes connected by edges. **Binary Tree** is a special data structure used for data storage purposes. A binary tree has a special condition that each node can have a maximum of two children. A binary tree has the benefits of both an ordered array and a linked list as search is as quick as in a sorted array and insertion or deletion operation are as fast as in linked list.

Binary Search Tree Representation:

- Binary Search tree exhibits a special behavior. A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.
- **A Binary Search Tree (BST)** is a tree in which all the nodes follow the below-mentioned properties −
  - The left sub-tree of a node has a key less than or equal to its parent node's key.
  - The right sub-tree of a node has a key greater than or equal to its parent node's key.

Thus, BST divides all its sub-trees into two segments; the left sub-tree and the right sub-tree and can be defined as –

left_subtree (keys) ≤ node (key) ≤ right_subtree (keys)

Tree Node

Following Structure is used for Node creation

```
Struct node {
Int data ;
Struct node *leftChild;
Struct node *rightChild;
};
```

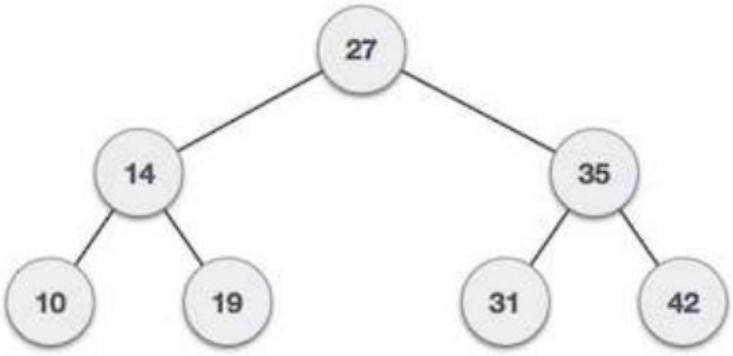


Fig: Binary Search Tree BST Basic Operations

The basic operations that can be performed on a binary search tree data structure, are the following −

- **Insert**− Inserts an element in a tree/create a tree.
- **Search**− Searches an element in a tree.
- **Traversal**− A traversal is a systematic way to visit all nodes of T -Inorder, Preprder, Postorder,

a. pre-order: Root, Left, Right

Parent comes before children; overall root first

B.post-order: Left, Right, Root

Parent comes after children; overall root last

c. In Order: In-order: Left, Root, Right,

Insert Operation: Algorithm

```
If root is NULL
   then create root node
return

If root exists then
   compare the data with node.data

   while until insertion position is located

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree
```

Search Operation: Algorithm

```
If root.data is equal to search.data
   return root
else
   while data not found

      If data is greater than node.data
         goto right subtree
      else
         goto left subtree

      If data found
         return node

   endwhile
```

Tree Traversal

In order traversal algorithm

sdsds

```
Until all nodes are traversed -

Step 1 - Recursively traverse left subtree.

Step 2 - Visit root node.

Step 3 - Recursively traverse right subtree.
```

Pre order traversal Algorithm

```
Until all nodes are traversed -
Step 1 - Visit root node.
Step 2 - Recursively traverse left subtree.
Step 3 - Recursively traverse right subtree.
```

Post order traversal Algorithm

```
Until all nodes are traversed -
Step 1 - Recursively traverse left subtree.
Step 2 - Recursively traverse right subtree.
Step 3 - Visit root node.
```

Deleting in a BST

case 1: delete a node with zero child-

if x is left of its parent, set parent(x).left = null

else set parent(x).right = null

case 2: delete a node with one child

link parent(x) to the child of x

case 3: delete a node with 2 children

Replace inorder successor to deleted node position.

Conclusion/analysis

We are able to implement Binary search Tree and its operation in Data Structure.

Assignment No 4 - B2

Problem Statement:

For given expression eg. a-b*c-d/e+f construct inorder sequence and traverse it using postorder traversal(non recursive).

Objective:

To understand the basic concept of Non Linear Data Structure "TREE "and its construction by given inorder sequence in Data structure.

Outcome:

To construct the Binary Tree by given inorder sequence and traverse it using postorder traversal.

Software & Hardware Requirements:

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

Theory –

Concept in brief: In this assignment first receive the infix expression from user than convert it into postfix and then construct tree from this postfix expression. At the end do all the traversal on the constructed tree non recursively.

Infix expression:

The expression of the form a op b. When an operator is in-between every pair of operands.

Postfix expression: The expression of the form a b op. When an operator is followed for every pair of operands.

Why postfix representation of the expression?

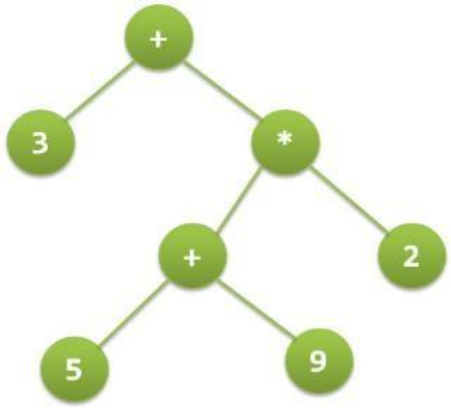
- The compiler scans the expression either from left to right or from right to left.
- Consider the below expression: a op1 b op2 c op3 d If op1 = +, op2 = *, op3 = +
- The compiler first scans the expression to evaluate the expression b * c, then again scan the expression to add a to it.
- The result is then added to d after another scan.
- The repeated scanning makes it very in-efficient. It is better to convert the expression to postfix(or prefix) form before evaluation.
- The corresponding expression in postfix form is: abc*d++. The postfix expressions can be evaluated easily using a stack. We will cover postfix expression evaluation in a separate post.

Algorithm

1. Scan the infix expression from left to right.
2. If the scanned character is an operand, output it.
3. Else,
…..**3.1** If the precedence of the scanned operator is greater than the precedence of the operator in the stack(or the stack is empty), push it.
…..**3.2** Else, Pop the operator from the stack until the precedence of the scanned operator is less-equal to the precedence of the operator residing on the top of the stack. Push the scanned operator to the stack.
4. If the scanned character is an '(', push it to the stack.
5. If the scanned character is an ')', pop and output from the stack until an '('is encountered.
6. Repeat steps 2-6 until infix expression is scanned.
7. Pop and output from the stack until it is not empty.

Expression Tree:

Expression tree is a binary tree in which each internal node corresponds to operator and each leaf node corresponds to operand so for example expression tree for 3 + ((5+9)*2) would be:



Construction of Expression Tree:

Now For constructing expression tree we use a stack. We loop through input expression and do following for every character.

1) If character is operand push that into stack
2) If character is operator pop two values from stack make them its child and push current node again.

At the end only element of stack will be root of expression tree.

Inorder Tree Traversal without Recursion

Using Stack is the obvious way to traverse tree without recursion. Below is an algorithm for traversing binary tree using stack. See this for step wise step execution of the algorithm.

1) Create an empty stack S.
2) Initialize current node as root
3) Push the current node to S and set current = current->left until current is NULL
4) If current is NULL and stack is not empty then
   a) Pop the top item from stack.
   b) Print the popped item, set current = popped_item->right
   c) Go to step 3.
5) If current is NULL and stack is empty then we are done.

Iterative Preorder Traversal

Given a Binary Tree, write an iterative function to print Preorder traversal of the given binary tree.

To convert an inherently recursive procedures to iterative, we need an explicit stack. Following is a simple stack based iterative process to print Preorder traversal.

1) Create an empty stack *nodeStack* and push root node to stack.
2) Do following while *nodeStack* is not empty.
….**a)** Pop an item from stack and print it.
….**b)** Push right child of popped item to stack
….**c)** Push left child of popped item to stack

Right child is pushed before left child to make sure that left subtree is processed first.

Iterative Post order Traversal | Set 2 (Using One Stack)

We have discussed a simple iterative postorder traversal using stacks .The idea is to move down to leftmost node using left pointer. While moving down, push root and root's right child to stack. Once we reach leftmost node, print it if it doesn't have a right child. If it has a right child, then change root so that the right child is processed before.

Following is detailed algorithm.

1.1 Create an empty stack

2.1 Do following while root is not NULL

a) Push root's right child and then root to stack.

b) Set root as root's left child.

2.2 Pop an item from stack and set it as root.

a) If the popped item has a right child and the right child is at top of stack, then remove the right child from stack, push the root back and set root as root's right child.

b) Else print root's data and set root as NULL.

2.3 Repeat steps 2.1 and 2.2 while stack is not empty.

Conclusion/analysis: We are able to construct the Binary Tree by given in order sequence and traverse it using post order traversal.

Assignment No 5 - B3

Problem Statement:

Convert given binary tree into threaded binary tree. Analyze time and space complexity of the algorithm.

Objective:

To understand the basic concept of Non Linear Data Structure "**threaded binary tree** " and its use in Data structure.

Outcome:

To convert the Binary Tree into **threaded binary tree and analyze its time and space complexity.**

Software & Hardware Requirements:

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

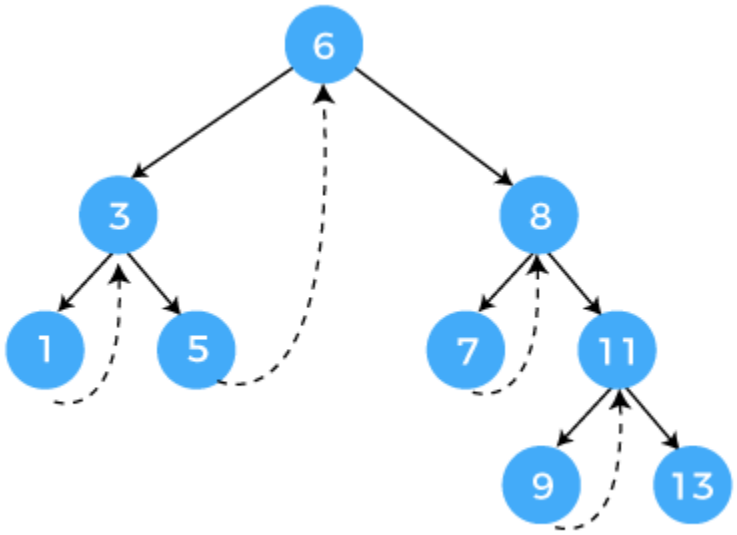
Theory –

Threaded Binary Tree

Inorder traversal of a Binary tree is either be done using recursion or with the use of a auxiliary stack. The idea of threaded binary trees is to make inorder traversal faster and do it without stack and without recursion. A binary tree is made threaded by making all right child pointers that would normally be NULL point to the inorder successor of the node (if it exists).

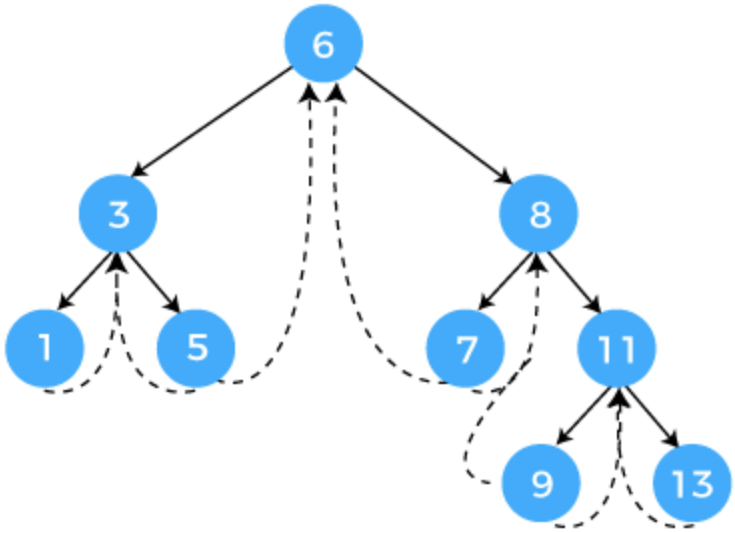
There are two types of threaded binary trees.

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



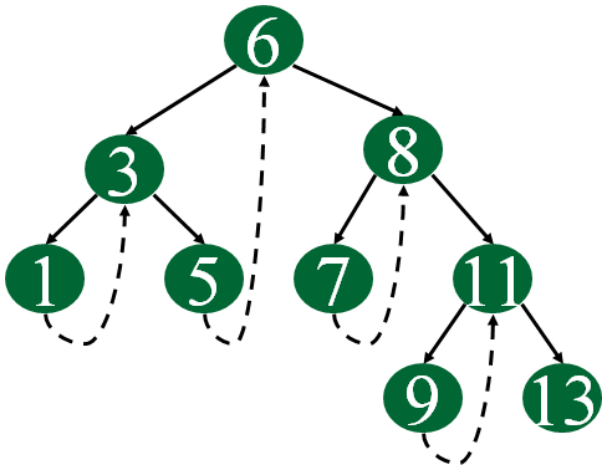
Single Threaded Binary Tree

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.The threads are also useful for fast accessing ancestors of a node.



Double Threaded Binary Tree

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



C representation of a Threaded Node:

Following is C representation of a single threaded node.

```
struct Node
{
   int data;
   Node *left, *right;
   bool rightThread;
}
```

How to convert a Given Binary Tree to Threaded Binary Tree?

- We basically need to set NULL right pointers to inorder successor.
- We first do an inorder traversal of the tree and store it in a queue (we can use a simple array also) so that the inorder successor becomes the next node.
- We again do an inorder traversal and whenever we find a node whose right is NULL, we take the front item from queuue and make it the right of current node.
- We also set isThreaded to true to indicate that the right pointer is a threaded link.

Advantages of Threaded Binary Tree:

- In threaded binary tree, linear and fast traversal of nodes in the tree so there is no requirement of stack. If the stack is used then it consumes a lot of memory and time.
- It is more general as one can efficiently determine the successor and predecessor of any node by simply following the thread and links. It almost behaves like a circular linked list.

Disadvantages of Threaded Binary Tree:

- When implemented, the threaded binary tree needs to maintain the extra information for each node to indicate whether the link field of each node points to an ordinary node or the node's successor and predecessor.
- Insertion into and deletion from a threaded binary tree are more time consuming since both threads and ordinary links need to be maintained.

Conclusion/analysis

Able to convert the Binary Tree into threaded binary tree and analyze its time and space complexity.

Assignment No 6 - C1

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 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.

Objective:

To understand the basic concept of Non Linear Data Structure "**Graph** " and its representations in Data structure.

Outcome:

To represent the Graph into **Adjacency Matrix and Adjacency List and analyze its time and space complexity.**

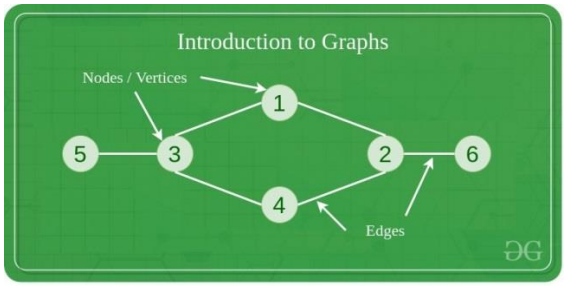
Software & Hardware Requirements:

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

Theory –

What is Graph Data Structure?

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by G(E, V).



Components of a Graph:

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
- **Edges:** Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.

Applications:

- Graphs are used to solve many real-life problems. Graphs are used to represent networks.
- The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.
- Graphs can broadly be categorized into **Undirected** (Fig 2a) or **Directed** (Fig 2b). An undirected graph is directionless. This means that the edges have no directions. In other words, the relationship is mutual. For example, a Facebook or a LinkedIn connection. Contrarily, edges of directed graphs

have directions associated with them. An asymmetric relationship between a boss and an employee or a teacher and a student can be represented as a directed graph in data structure. Graphs can also be **weighted** (Fig 2c) indicating real values associated with the edges. Depending upon the specific use of the graph, edge weights may represent quantities such as distance, cost, similarity etc.

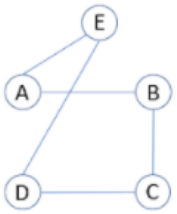


Fig 2a: Undirected Graph

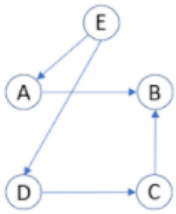
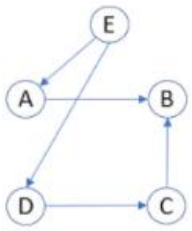


Fig 2b: Directed Graph

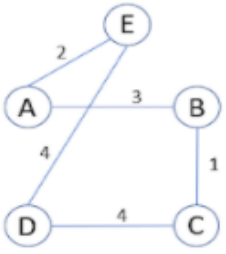
Representing Graphs:

- A graph can be represented using 3 data structures- **adjacency matrix, adjacency list and adjacency set.**
- An **adjacency matrix** can be thought of as a table with rows and columns. The row labels and column labels represent the nodes of a graph. An adjacency matrix is a square matrix where the number of rows, columns and nodes are the same. Each cell of the matrix represents an edge or the relationship between two given nodes. For example, adjacency matrix **Aij** represents the number of links from i to j, given two nodes i and j.



	A	B	C	D	E
A	0	1	0	0	0
B	0	0	0	0	0
C	0	1	0	0	0
D	0	0	1	0	0
E	1	0	0	1	0

Fig 3: Adjacency Matrix for a directed graph



	A	B	C	D	E
A	0	3	0	0	2
B	3	0	1	0	0
C	0	1	0	4	0
D	0	0	4	0	4
E	2	0	0	4	0

Fig 5: Adjacency Matrix for a weighted graph

Fig 4: A

- In **adjacency list** representation of a graph, every vertex is represented as a node object. The node may either contain data or a reference to a linked list. This linked list provides a list of all nodes that are adjacent to the current node. Consider a graph containing an edge connecting node A and node B. Then, the node A will be available in node B's linked list. Fig 6 shows a sample graph of 5 nodes and its corresponding adjacency list.

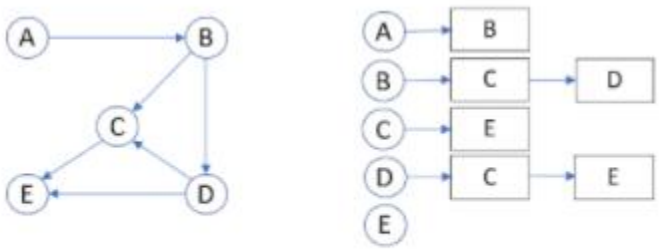


Fig 6: Adjacency list for a directed graph

- Note that the list corresponding to node E is empty while lists corresponding to nodes B and D have 2 entries each.
- Similarly, adjacency lists for an undirected graph can also be constructed. Fig 7 provides an example of an undirected graph along with its adjacency list for better understanding.

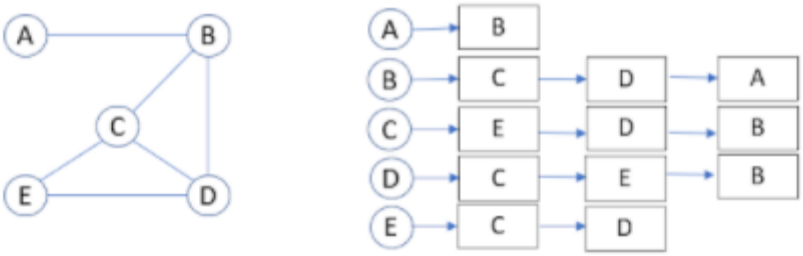


Fig 7: Adjacency list for an undirected graph

- Adjacency list enables faster search process in comparison to adjacency matrix. However, it is not the best representation of graphs especially when it comes to adding or removing nodes. For example, deleting a node would involve looking through all the adjacency lists to remove a particular node from all lists.

Conclusion:

- Thus, we studied a Graph data structure with its representations.

Assignment No 7 – C2

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.

Objective:

To understand the concept and basic of spanning and to find the minimum distance between the vertices of Graph in Data structure.

Outcome:

To implement the concept and basic of spanning and to find the minimum distance between the vertices of Graph in Data structure.

Software & Hardware Requirements:

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

Theory :

Properties of a Greedy Algorithm:

1. At each step, the best possible choice is taken and after that only the sub-problem is solved.
2. Greedy algorithm might be depending on many choices. But, it cannot ever be depending upon any choices of future and neither on sub-problems solutions.
3. The method of greedy algorithm starts with a top and goes down, creating greedy choices in a series and then reduce each of the given problem to even smaller ones.

Minimum Spanning Tree:

A Minimum Spanning Tree (MST) is a kind of a sub graph of an undirected graph in which, the sub graph spans or includes all the nodes has a minimum total edge weight.

To solve the problem by a prim's algorithm, all we need is to find a spanning tree of minimum length, where a spanning tree is a tree that connects all the vertices together and a minimum spanning tree is a spanning tree of minimum length.

Properties of Prim's Algorithm:

Prim's Algorithm has the following properties:

1. The edges in the subset of some minimum spanning tree always form a single tree.
2. It grows the tree until it spans all the vertices of the graph.
3. An edge is added to the tree, at every step, that crosses a cut if its weight is the minimum of any edge crossing the cut, connecting it to a vertex of the graph.

Algorithm:

1. Begin with any vertex which you think would be suitable and add it to the tree.
2. Find an edge that connects any vertex in the tree to any vertex that is not in the tree. Note that, we don't have to form cycles.
3. Stop when n - 1 edges have been added to the tree

Conclusion:

Understood the concept and basic of spanning and to find the minimum distance between the vertices of Graph in Data structure.

Assignment No 8 – D1

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.

Objective:

To understand the concept and basic of OBST in Data structure.

Outcome:

To implement the basic concept of OBST tree and to perform basic operation insert, search and delete an element in OBST tree also able to apply correct rotation when tree is unbalance after insertion and deletion operation in Data structure.

Software & Hardware Requirements:

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

Theory:

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:

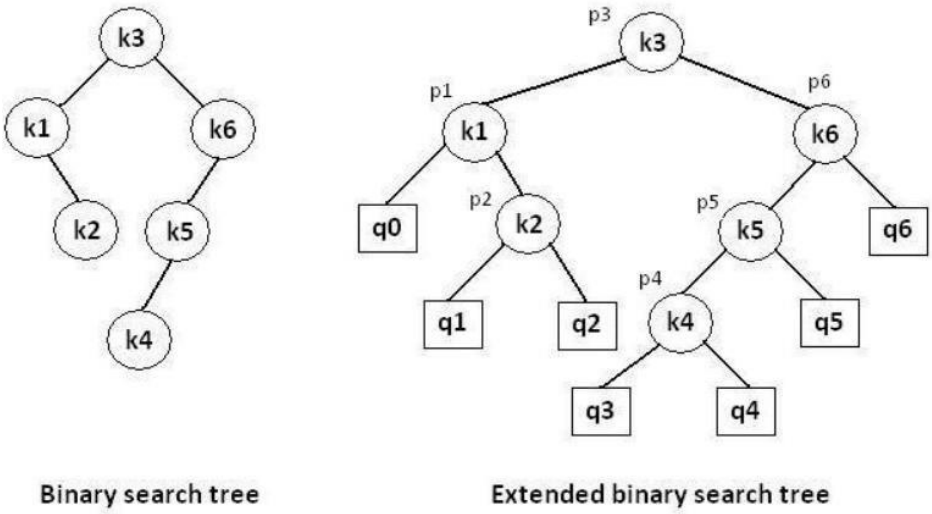
he squares represent terminal nodes. These terminal nodes represent unsuccessful searches of the tree for key values. The searches did not end successfully, that is, because they represent key v alues that are not actually stored in the tree; Optimal Binary Search Trees 2 the round nodes represent internal nodes; these are the actual keys stored in the tree; assuming that the relative frequency with which eac h 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 k1can 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 successed between kiand ki-1in an inorder traversal represents all key values not stored that lie between ki and ki – 1.

.



Binary search tree Extended binary search tree

ALGORITHMS IN PSEUDOCODE

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 do
            C (i, i) ← 0
            W (i, i) ← q(i)

      for m = 0 to n do
            for i = 0 to (n - m) do
               j ← i + m
               W (i, j) ← W (i, j - 1) + p (j) + q (j)
               *find C (i, j) and R (i, j) which minimize the tree 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)

      if 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 N
      if 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 N
      return N
end
```

Conclusion: We understand the concept of OBST tree.

Assignment No 9 – D2

Problem Statement:

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

Objective:

To understand the concept and basic of Height balanced Binary Search tree or AVL tree in Data structure.

Outcome:

To implement the basic concept of AVL tree and to perform basic operation insert, search and delete an element in AVL tree also able to apply correct rotation when tree is unbalance after insertion and deletion operation in Data structure.

Software & Hardware Requirements:

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

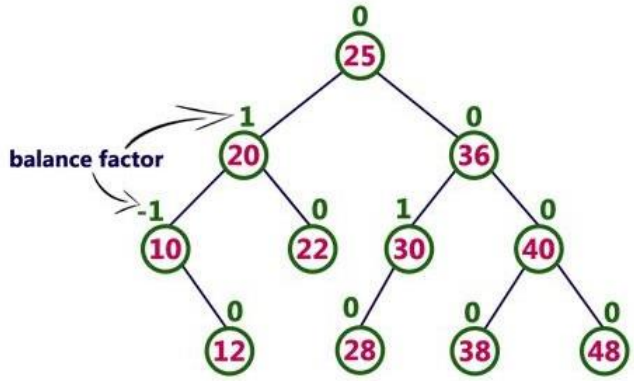
Theory Concepts:

AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the hieghts of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as **balance factor**. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor = heightOfLeftSubtree – heightOfRightSubtree



AVL Tree

AVL tree is a self balanced binary search tree. That means, an AVL tree is also a binary search tree but it is a balanced tree. A binary tree is said to be balanced, if the difference between the hieghts of left and right subtrees of every node in the tree is either -1, 0 or +1. In other words, a binary tree is said to be balanced if for every node, height of its children differ by at most one. In an AVL tree, every node maintains a extra information known as **balance factor**. The AVL tree was introduced in the year of 1962 by G.M. Adelson-Velsky and E.M. Landis.

An AVL tree is defined as follows...

An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

Balance factor of a node is the difference between the heights of left and right subtrees of that node. The balance factor of a node is calculated either **height of left subtree - height of right subtree** (OR) **height of right subtree - height of left subtree**. In the following explanation, we are calculating as follows...

Balance factor = heightOfLeftSubtree - heightOfRightSubtree

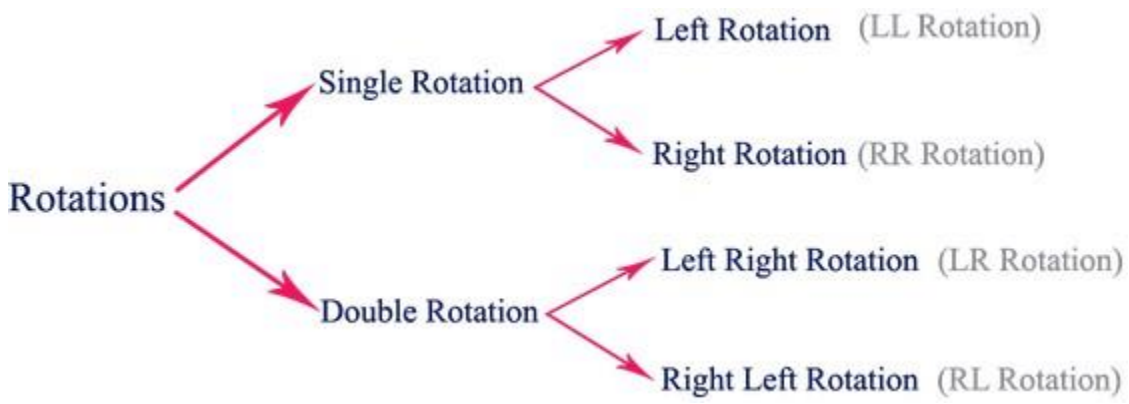
Every AVL Tree is a binary search tree but all the Binary Search Trees need not to be AVL trees.

AVL Tree Rotations

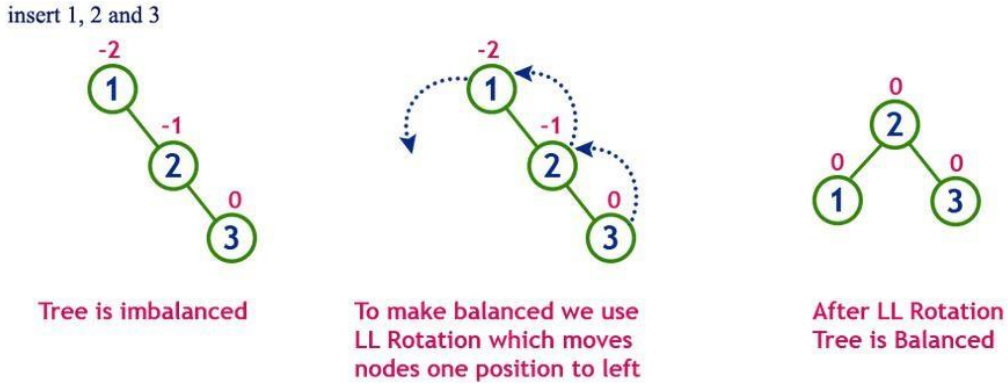
In AVL tree, after performing every operation like insertion and deletion we need to check the **balance factor** of every node in the tree. If every node satisfies the balance factor condition then we conclude the operation otherwise we must make it balanced. We use **rotation** operations to make the tree balanced whenever the tree is becoming imbalanced due to any operation.

Rotation operations are used to make a tree balanced.

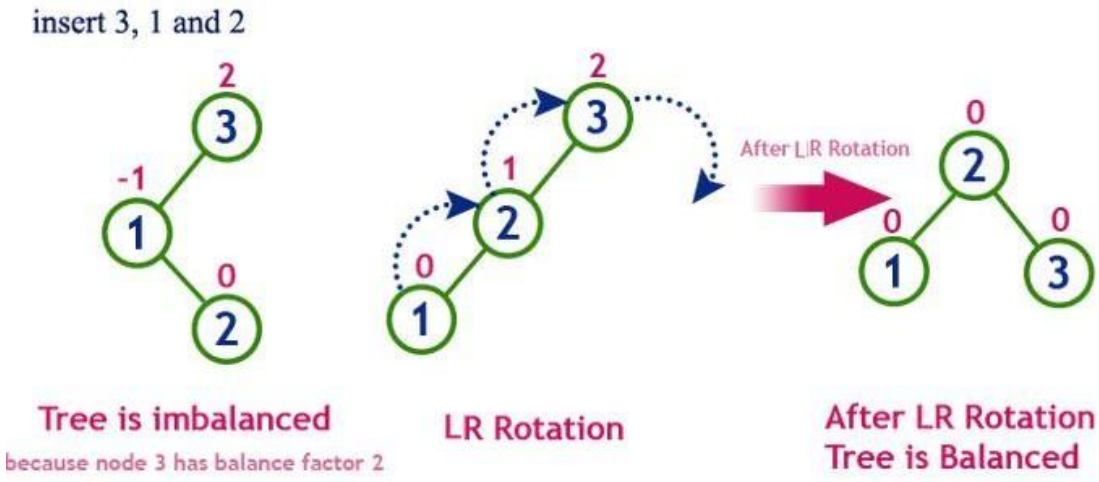
There are **four** rotations and they are classified into **two** types.



Single Left Rotation (LL Rotation)



Double Rotation (LR Rotation)



Similarly RR and RL rotation can be performed

Algorithm:

The following operations are performed on an AVL tree...

1. Search
2. Insertion
3. Deletion

Search Operation in AVL Tree

In an AVL tree, the search operation is performed with O(log n) time complexity. The search operation is performed similar to Binary search tree search operation. We use the following steps to search an element in AVL tree...

- Step 1: Read the search element from the user
- Step 2: Compare, the search element with the value of root node in the tree.
- Step 3: If both are matching, then display "Given node found!!!" and terminate the function
- Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.
- Step 5: If search element is smaller, then continue the search process in left subtree.
- Step 6: If search element is larger, then continue the search process in right subtree.
- Step 7: Repeat the same until we found exact element or we completed with a leaf node
- Step 8: If we reach to the node with search value, then display "Element is found" and terminate the function.
- Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

Insertion Operation in AVL Tree

In an AVL tree, the insertion operation is performed with O(log n) time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

- Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.
- Step 2: After insertion, check the Balance Factor of every node.

- Step 3: If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.
- Step 4: If the Balance Factor of any node is other than 0 or 1 or -1 then tree is said to be imbalanced. Then perform the suitable Rotation to make it balanced. And go for next operation.

Conclusion: We are able to implement AVL Binary Search tree and maintain its height after every operation

Assignment No 10 – E1

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.

Objective:

To understand the basic concept of Heap Data structure.

Outcome:

To implement the concept and basic of Max Heap and Min Heap Data Structure and its use in Data structure.

Software & Hardware Requirements:

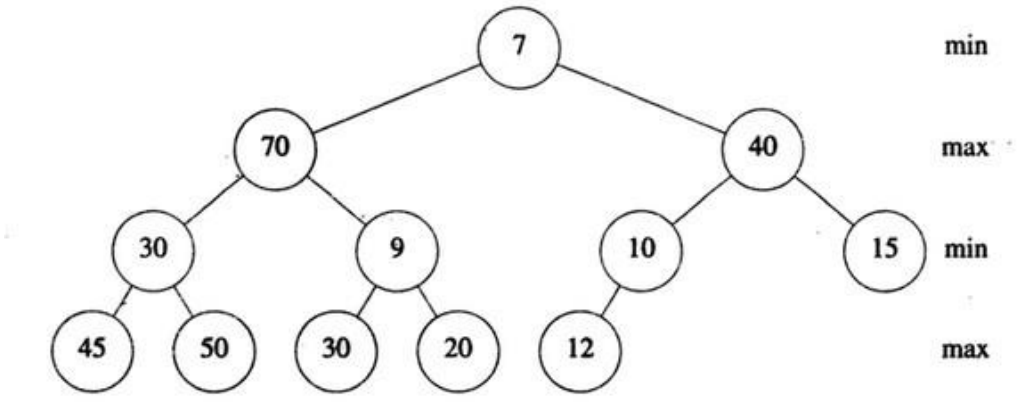
1. 64-bit Open source Linux or its derivative
2. Open Source C++ Programming tool like G++/GCC

Theory:

A double ended priority queue is a data structure that support the following operation
1. Inserting an element with an arbitrary key
2. Deleting an element with largest key
3. Deleting an element with smallest key

When only insertion and one of the two deletion operation re to be supported,a max heap or min heap may be used. A max-min heap support all of above operations.

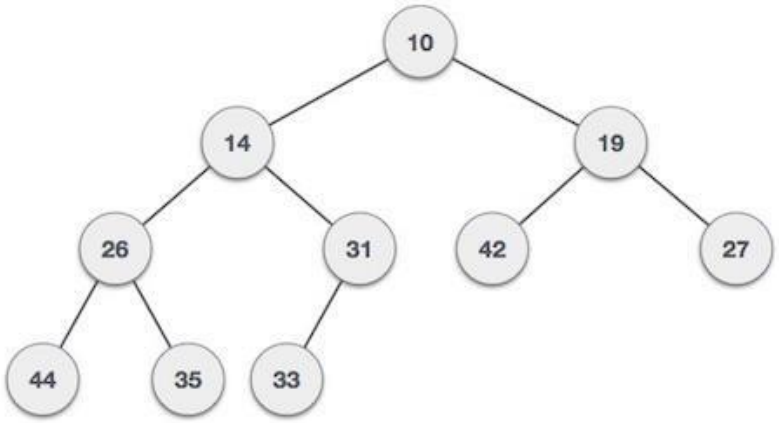
Definition: A min-Max heap is a complete binary tree such that if it is not empty,each element has a data member called key.Alternating levels of this tree are min levels and max level,respectively.the root is on min level.



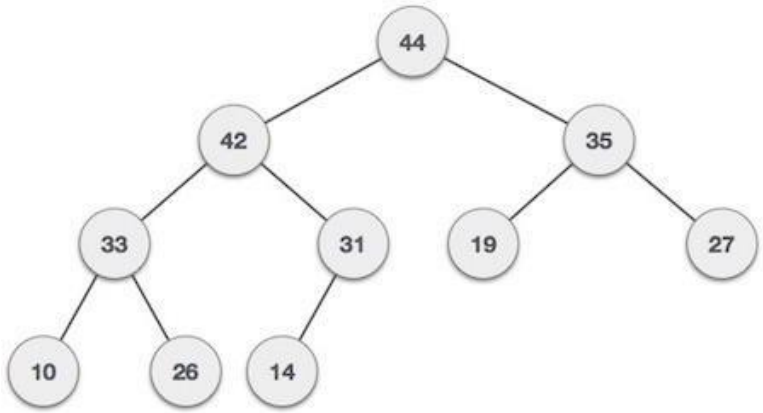
A 12 element Max-Min heap

```
template <class KeyType>
class DEPQ {
public:
    virtual void Insert(const Element<KeyType>&) = 0;
    virtual Element<KeyType>* DeleteMax(Element<KeyType>&) = 0;
    virtual Element<KeyType>* DeleteMin(Element<KeyType>&) = 0;
};
```

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.



Algorithm:

Max heap Construction

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

Max heap Deletion Algorithm

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.

Max-Min Heap

Insert:

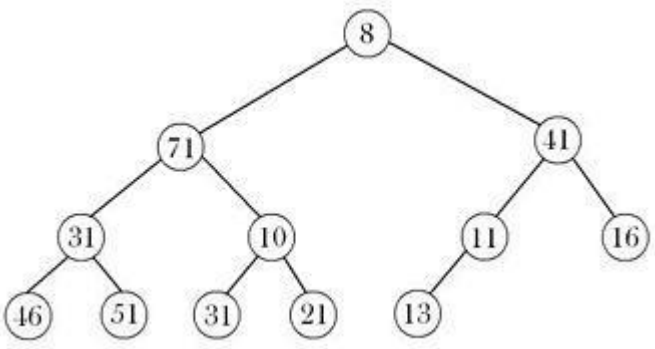
To add an element to a min-max heap perform following operations:

1. Append the required key to the array representing the min-max heap. This will likely break the min-max heap properties, therefore we need to adjust the heap.
2. Compare this key with its parent:
    1. If it is found to be smaller (greater) compared to its parent, then it is surely smaller (greater) than all other keys present at nodes at max(min) level that are on the path from the present position of key to the root of heap. Now, just check for nodes on Min(Max) levels.
    2. If the key added is in correct order then stop otherwise swap that key with its parent.

Example

Here is one example for inserting an element to a Min-Max Heap.

Say we have the following min-max heap and want to install a new node with value 6.



Initially, element 6 is inserted at the position indicated by *j*. Element 6 is less than its parent element. Hence it is smaller than all max levels and we only need to check the min levels. Thus, element 6 gets moved to the root position of the heap and the former root, element 8, gets moved down one step.

If we want to insert a new node with value *81* in the given heap, we advance similarly. Initially the node is inserted at the position *j*. Since element 81 is larger than its parent element and the

parent element is at min level, it is larger than all elements that are on min levels. Now we only need to check the nodes on max levels.

Delete :

Delete the minimum element
To delete min element from a Min-Max Heap perform following operations.
The smallest element is the root element.

1. Remove the root node and the node which is at the end of heap. Let it be x.
2. Reinsert key of x into the min-max heap

Reinsertion may have 2 cases:

1. If root has no children, then x can be inserted into the root.
2. Suppose root has at least one child. Find minimum value (Let this is be node m). m is in one of the children or grandchildren of the root. The following condition must be considered:
    1. x.key <= h[m].key: x must be inserted into the root.
    2. x.key > h[m].key and m is child of the root L: Since m is in max level, it has no descendants. So, the element h[m] is moved to the root and x is inserted into node m.
    3. x.key > h[m].key and m is grandchild of the root: So, the element h[m] is moved to the root. Let p be parent of m. if x.key > h[p].key then h[p] and x are interchanged.

Algorithm	**Average**	**Worst Case**
Insert	$O(\log_2 n)$	$O(\log_2 n)$
Delete	$O(\log_2 n)$	$O(\log_2 n)$

Conclusion: We are able to implement Max /Min heap concept in Data Structure

Assignment No 11 – F1

Department maintains a student information. The file contains roll number, name, and 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.

Objective:

To understand the concept and basic of sequential file and its use in Data structure

Outcome:

To implement the concept and basic of sequential file and to perform basic operation as adding record, display all record, search record from sequential file and its use in Data structure.

Software & Hardware Requirements:

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

Theory Concepts:

Types of File Organization:

Types of file organization are

1. Sequential Access File Organization
2. Direct Access File Organization
3. Index Sequential Access File Organization

Sequential Access File Organization:

1. All records are stored in a sequential order.
2. That is, the records are arranged in the ascending or descending order of a key field.
    - a) In a student information system, the file would contain roll number, name, division, marks obtained in the examination.
    - In a payroll application, the records are stored with employee number as a key field.
3. To locate a particular record in such file organization, we have to start searching from the beginning of the file until it is found in the file.
4. It is time consuming process.
5. Normally created and maintained on magnetic tapes. e.g. Audio Cassettes.
6. There is no need for any storage space identification

Advantages:

1. Simple to understand
2. Easier to organize, maintain
3. Economical
4. Error in files remain localized

Disadvantages:

1. Entire file has to be processed
2. Transactions must be sorted in a particular sequence before processing
3. Time consuming searching
4. High data redundancy
5. Random enquiries are not possible to handle

Conclusion: we understand the concept and basic of sequential file and its use in Data structure

Assignment No 12 – F2

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.

Objective:

To understand Index sequential file and its use in Data structure.

Outcome:

To implement the concept of index sequential file like employee information as employee ID, name, designation and salary and to perform basic operation as adding record, deleting record, search record.

Software & Hardware Requirements:

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

Theory –

File Organization

File organization ensures that records are available for processing. It is used to determine an efficient file organization for each base relation. For example, if we want to retrieve employee records in alphabetical order of name. Sorting the file by employee name is a good file organization. However, if we want to retrieve all employees whose marks are in a certain range, a file is ordered by employee name would not be a good file organization.

Types of File Organization

There are three types of organizing the file:

1. Sequential access file organization

2. Direct access file organization

3. Indexed sequential access file organization

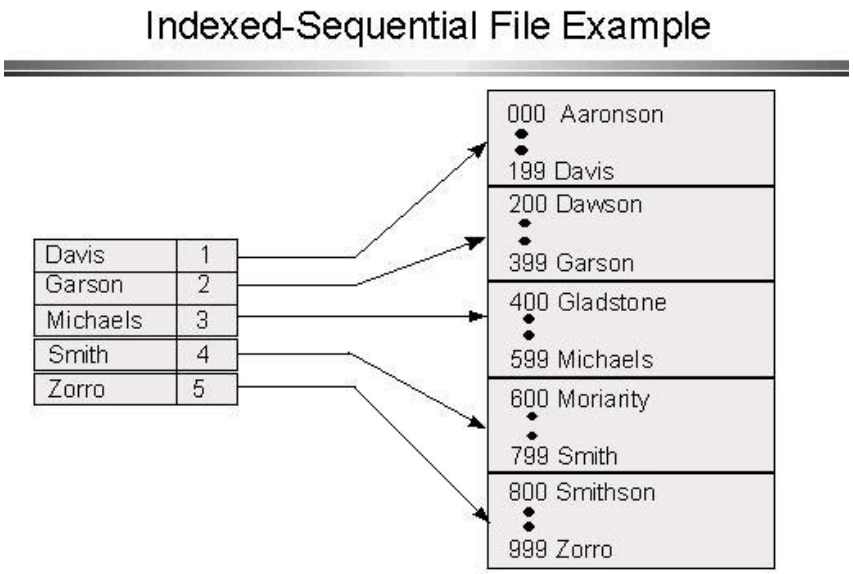
Indexed sequential access file organization

• Indexed sequential access file combines both sequential file and direct access file organization.

• In indexed sequential access file, records are stored randomly on a direct access device such as magnetic disk by a primary key.

• This file have multiple keys. These keys can be alphanumeric in which the records are ordered is called primary key.

• The data can be access either sequentially or randomly using the index. The index is stored in a file and read into memory when the file is opened.



Advantages of Indexed sequential access file organization

• In indexed sequential access file, sequential file and random file access is possible.

• It accesses the records very fast if the index table is properly organized.

• The records can be inserted in the middle of the file.

• It provides quick access for sequential and direct processing.

• It reduces the degree of the sequential search.

Disadvantages of Indexed sequential access file organization

• Indexed sequential access file requires unique keys and periodic reorganization.

• Indexed sequential access file takes longer time to search the index for the data access or retrieval.

• It requires more storage space.

• It is expensive because it requires special software.

• It is less efficient in the use of storage space as compared to other file organizations.

Conclusion:

- Thus, we have studied an **Indexed sequential file** and its operations.

**********ALL THE BEST**********