NextGen Academy

Towards fulfilling a million dreams

Data Structures Using C

Module III: Trees

June 20, 2023

Contents

This unit covers

- Non-Linear Data Structures: Introducing non-linear structures in computer science, focusing on trees as a key example distinct from linear structures like arrays and linked lists.
- Basic Tree Terminologies: Defining fundamental concepts associated with trees, providing a foundational understanding before delving into specific types.
- Types of Trees: Exploring various tree types, including Binary Trees, Binary Search Trees (BST), AVL Trees, B-Trees, and Heaps, each with unique characteristics and applications.
- Representation and Implementations: Discussing how different types of trees are represented and implemented in computer memory, outlining the structures and methodologies used.
- Tree Traversal Algorithms: Covering methods to systematically visit each node in a tree data structure, such as in-order, pre-order, and post-order traversals.
- Operations on Trees: Detailing key operations performed on trees, including insertion and deletion of nodes, highlighting their significance in maintaining tree properties.
- Applications of Trees: Exploring real-world applications where trees are used, ranging from database systems to hierarchical data representation and algorithms.

1 Non linear data structure

A non-linear data structure is another important type in which data elements are not arranged sequentially; mainly, data elements are arranged in random order without forming a linear structure. Data elements are present at multiple levels, for example, a tree. In trees, the data elements are arranged in a hierarchical form, whereas in graphs, the data elements are arranged in random order, using the edges and vertices.

- Multiple runs are required to traverse through all the elements completely. Traversing in a single run is impossible to traverse the whole data structure.
- Each element can have multiple paths to reach another element.
- The data structure where data items are not organized sequentially is called a non-linear data structure. In other words, data elements of the non-linear data structure could be connected to more than one element to reflect a special relationship among them.

Types of Non-linear Data Structures

Trees and Graphs are the types of non-linear data structures.

Tree

The tree is a non-linear data structure that is comprised of various nodes. The nodes in the tree data structure are arranged in hierarchical order.

- It consists of a root node corresponding to its various child nodes, present at the next level. The tree grows on a level basis, and root nodes have limited child nodes depending on the order of the tree.
- For example, in the binary tree, the order of the root node is 2, which means it can have at most 2 children per node, not more than it.
- The non-linear data structure cannot be implemented directly, and it is implemented using the linear data structure like an array and linked list.
- The tree itself is a very broad data structure and is divided into various categories like Binary tree, Binary search tree, AVL trees, Heap, max Heap, min-heap, etc. All the types of trees mentioned above differ based on their properties.

Graph

A graph is a non-linear data structure with a finite number of vertices and edges, and these edges are used to connect the vertices.

- The graph itself is categorized based on some properties; if we talk about a complete graph, it consists of the vertex set, and each vertex is connected to the other vertices having an edge between them.

- The vertices store the data elements, while the edges represent the relationship between the vertices.
- A graph is very important in various fields; the network system is represented using the graph theory and its principles in computer networks.
- Even in Maps, we consider every location a vertex, and the path derived between two locations is considered edges. The graph representation's main motive is to find the minimum distance between two vertices via a minimum edge weight.

Properties of Non-linear Data Structures

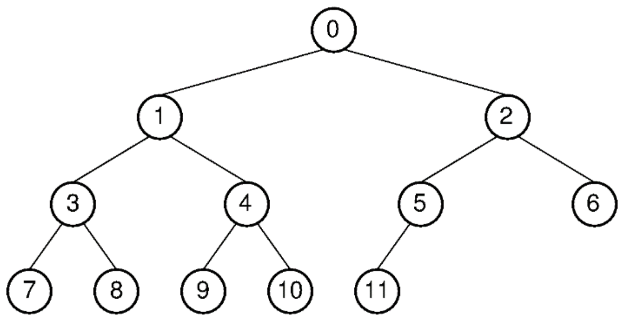
- It is used to store the data elements combined whenever they are not present in the contiguous memory locations.
- It is an efficient way of organizing and properly holding the data.
- It reduces the wastage of memory space by providing sufficient memory to every data element.

Unlike in an array, we have to define the size of the array, and subsequent memory space is allocated to that array; if we don't want to store the elements till the range of the array, then the remaining memory gets wasted. So to overcome this factor, we will use the non-linear data structure and have multiple options to traverse from one node to another.

- Data is stored randomly in memory.
- It is comparatively difficult to implement.
- Multiple levels are involved.
- Memory utilization is effective.

2 Trees: Basic Tree terminologies

Data structures are specialized formats for organizing and storing data to enable efficient operations on the data. They provide a way to manage and organize data so that it can be used effectively. Examples of data structures include arrays, linked lists, stacks, queues, trees, and graphs. Tree data structures are hierarchical structures that consist of nodes connected by edges. Each node in a tree has a parent-child relationship with other nodes, forming a hierarchy. Trees are widely used in computer science and are the foundation for many important data structures and algorithms.



(a)

Fig:- Tree

Basic Terminology:

1. **Node:** In the context of trees and graphs, a node is a fundamental unit that contains data and may have links to other nodes.
2. **Edge:** An edge is a connection between two nodes in a graph. It represents a relationship between the connected nodes.
3. **Root:** The topmost node in a tree is called the root. It serves as the starting point for traversing the tree.
4. **Leaves:** Nodes in a tree that do not have any children are called leaves or leaf nodes.
5. **Parent and Child:** In a tree, a node is a parent if it has one or more child nodes. The nodes directly connected below a parent are its children.
6. **Linked List:** A linked list is a linear data structure where elements are stored in nodes, and each node points to the next node in the sequence.
7. **Stack:** A stack is a last-in, first-out (LIFO) data structure where elements are added and removed from the same end, called the top.
8. **Queue:** A queue is a first-in, first-out (FIFO) data structure where elements are added at the rear and removed from the front.

Types of Tree:

1. **Binary Tree:** Each node has at most two children, typically referred to as the left child and the right child.
2. **Binary Search Tree (BST):** A binary tree in which the left subtree of a node contains only nodes with keys less than the node's key, and the right subtree contains only nodes with keys greater than the node's key.
3. **AVL Tree:** A self-balancing binary search tree where the height of the left and right subtrees of any node differs by at most one.
4. **B-Tree:** A self-balancing tree structure that maintains sorted data and allows searches, insertion, and deletion in logarithmic time.

Time and Space Complexity Analysis:

1. **Time Complexity:** Time complexity is a measure of the amount of time an algorithm takes to complete as a function of the input size. It's expressed using Big-O notation.
2. **Space Complexity:** Space complexity is a measure of the amount of memory an algorithm uses as a function of the input size. Like time complexity, it's expressed using Big-O notation.

Big-O Notation:

1. **Big-O Notation:** Big-O notation describes the upper bound of the growth rate of an algorithm's time or space complexity in the worst-case scenario.
2. **O(1), O(log n), O(n), O(n log n), O(n^2), ...:** Examples of different time complexities. O(1) represents constant time, O(log n) logarithmic time, O(n) linear time, O(n log n) log-linear time, O(n^2) quadratic time, and so on.

3. **Best Case, Worst Case, and Average Case:** Algorithms may have different time complexities in different scenarios. Best case represents the lowest time complexity, worst case represents the highest, and average case represents the expected time complexity.

3 Types of Trees:

A tree data structure is a hierarchical structure that is used to represent and organize data in a way that is easy to navigate and search. It is a collection of nodes that are connected by edges and has a hierarchical relationship between the nodes.

The topmost node of the tree is called the root, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.

3.1 Binary Tree

A binary tree is a hierarchical tree structure in which each node has at most two children, usually referred to as the left child and the right child. The structure of a binary tree is such that each node can have zero, one, or two children.

1. **Full (Proper/Strict) Binary Tree:**

In a full binary tree, every node has either 0 or 2 children. It means that every level of the tree is completely filled, except possibly for the last level, which is filled from left to right.

2. **Complete Binary Tree:**

A complete binary tree is a binary tree in which all levels are completely filled, except possibly for the last level, which is filled from left to right. This type of tree is often used in heap data structures.

3. **Perfect Binary Tree:**

A perfect binary tree is both full and complete. This means that all internal nodes have exactly two children, and all leaf nodes are at the same level.

4. **Degenerate (or Pathological) Tree:**

A degenerate tree, also known as a pathological tree, is a tree in which each parent node has only one associated child node. It essentially forms a linked list rather than a balanced tree.

5. **Balanced Binary Tree:**

A balanced binary tree is a tree in which the height of the left and right subtrees of every node differs by at most one. The goal is to maintain balance to ensure efficient search and retrieval operations.

Binary Tree traversal

Tree traversal is the process of visiting and processing each node in a tree data structure. There are several methods for traversing trees, and the choice of method depends on the specific task or application. The three most common types of tree traversal are:

1. **In-Order traversal:** In an in-order traversal, you visit the left subtree, then the root, and finally the right subtree. For binary search trees, this traversal visits nodes in ascending order.

   Algorithm InOrderTraversal(node):

   ```
   1. If the node is not null:
   a. Perform an in-order traversal on the left subtree (node.left).
   b. Visit the current node (node).
   c. Perform an in-order traversal on the right subtree (node.right).
   ```

   In-Order Traversal: 4, 2, 5, 1, 3

   ```
    1
   / \
   2 3
   / \
   4 5
   ```

2. **Pre-Order traversal:** In a pre-order traversal, you visit the root first, then the left subtree, and finally the right subtree.

   ```
   Algorithm InOrderTraversalRecursive(node):
   1. If the node is not null:
   a. Perform an in-order traversal on the left subtree (node.left).
   b. Visit the current node (node).
   c. Perform an in-order traversal on the right subtree (node.right).
   ```

   Pre-Order Traversal: 1, 2, 4, 5, 3

   ```
    1
   / \
   2 3
   / \
   4 5
   ```

3. **Post-Order traversal:** In a post-order traversal, you visit the left subtree, then the right subtree, and finally the root.

   ```
   Algorithm PostOrderTraversalRecursive(node):
   1. If the node is not null:
   a. Perform a post-order traversal on the left subtree (node.left).
   ```

b. Perform a post-order traversal on the right subtree (node.right).

c. Visit the current node (node).

Post-Order Traversal: 4, 5, 2, 3, 1

```
1
/ \
2 3
/ \
4 5
```

C- Code for Tree Traversal:

```
#include <stdio.h>
#include <stdlib.h>
// Definition for a binary tree node
struct TreeNode {
int val;
struct TreeNode* left;
struct TreeNode* right;
};
// Function to perform in-order traversal
void inOrderTraversal(struct TreeNode* root) {
if (root != NULL) {
// Traverse the left subtree
inOrderTraversal(root->left);
// Visit the current node
printf("%d ", root->val);
// Traverse the right subtree
inOrderTraversal(root->right);
}
}
// Function to perform pre-order traversal
void preOrderTraversal(struct TreeNode* root) {
if (root != NULL) {
```

```
// Visit the current node
printf("%d ", root->val);
// Traverse the left subtree
preOrderTraversal(root->left);
// Traverse the right subtree
preOrderTraversal(root->right);
}
}
// Function to perform post-order traversal
void postOrderTraversal(struct TreeNode* root) {
if (root != NULL) {
// Traverse the left subtree
postOrderTraversal(root->left);
// Traverse the right subtree
postOrderTraversal(root->right);
// Visit the current node
printf("%d ", root->val);
}
}
// Function to create a new tree node
struct TreeNode* createNode(int value) {
struct TreeNode* newNode = (struct TreeNode*)malloc(sizeof(struct TreeNode));
newNode->val = value;
newNode->left = NULL;
newNode->right = NULL;
return newNode;
}
int main() {
// Create a sample binary tree
struct TreeNode* root = createNode(1);
root->left = createNode(2);
root->right = createNode(3);
root->left->left = createNode(4);
```

```
root->left->right = createNode(5);
// In-order traversal
printf("In-Order Traversal: ");
inOrderTraversal(root);
printf("\n");
// Pre-order traversal
printf("Pre-Order Traversal: ");
preOrderTraversal(root);
printf("\n");
// Post-order traversal
printf("Post-Order Traversal: ");
postOrderTraversal(root);
printf("\n");
// Free allocated memory (not shown in the basic example)
return 0;
}
```

3.2 Binary Search Tree (BST)

A Binary Search Tree (BST) is a binary tree data structure with the following properties:

1. **Binary Tree Structure:**

- Each node in the tree has at most two children, commonly referred to as the left child and the right child.

2. **Key Property:**

- The key (or value) of each node in the tree is greater than or equal to all the keys in its left subtree and less than or equal to all the keys in its right subtree.

3. **Ordering:**

- This key property ensures that a BST maintains a specific order among its elements. It facilitates efficient searching, insertion, and deletion operations.

The ordering principle of a Binary Search Tree allows for quick searching of elements, making it a valuable data structure for various applications where ordered data is essential. The structure of a BST enables efficient implementations of algorithms for tasks like searching for a particular element, finding the minimum and maximum elements, and performing range queries.

Example:-

10

/ \

5 15

/ \ \

3 8 20

Binary Search Tree Operations

1. **Creating a binary search tree**

Creating a binary search tree involves inserting elements into the tree while maintaining the properties of a binary search tree (BST).

```
Algorithm insertIntoBSTInOrder(elements):
1. Initialize the root of the tree as NULL.
2. For each element in the 'elements' list:
a. If the root is NULL, insert the element as the root of the tree.
b. If the element is smaller than the root's value:
i. Create a new node with the element and make it the root of the left subtree.
c. If the element is larger than the root's value:
i. Create a new node with the element and make it the root of the right subtree.
3. Return the root of the binary search tree.
```

Example

elements = [45, 30, 60, 20, 40, 50, 70]

Insert 45: 45

Insert 30: /

30

Insert 60: 45

/ \

30 60

Insert 20,40: 45

/ \

30 60

/ /

20 40

Insert 50, 70: 45

/ \

30 60

/ / \

20 40 70

2. **Searching:**

The algorithm for searching an element in a Binary Search Tree (BST) involves recursively traversing the tree based on the value of the target element and comparing it with the values stored in each node. Here's the search algorithm:

```
Algorithm searchBST(root, target):
1. If the root is NULL or the root's value is equal to the target, return the root.
2. If the target is less than the root's value, recursively search in the left subtree.
3. If the target is greater than the root's value, recursively search in the right subtree.
4. If the target is not found in the tree, return NULL.
```

Example Usage:

result = searchBST(root, target)

50

/ \

30 70

/ \ / \

20 40 60 80

Suppose you want to search for the target value 40. The algorithm would work as follows:

- Start at the root (50).
- Since 40 is less than 50, move to the left subtree.
- In the left subtree, 40 is found. Return the node containing 40.

3. **Deletion in Binary search tree**

The Deletion in a Binary Search Tree (BST) involves removing a node with a specific value while maintaining the properties of a BST. The process depends on the number of children the node to be deleted has:

1. **Node with No Children (Leaf Node):**

- Simply remove the node.

2. **Node with One Child:**

- Replace the node with its child.

3. **Node with Two Children:**

- Find the in-order successor (or predecessor) of the node to be deleted. The in-order successor is the smallest (or largest) node in its right (or left) subtree.
- Replace the node's value with the in-order successor's (or predecessor's) value.
- Recursively delete the in-order successor (or predecessor).

Algorithm deleteNode(root, target):

```
1. If the root is NULL, return NULL.
2. If the target is less than the root's value, recursively delete in the left subtree.
3. If the target is greater than the root's value, recursively delete in the right subtree.
4. If the target is equal to the root's value, handle the deletion based on the number of children:
5. If the root has no children (leaf node), return NULL.
6. If the root has one child, return the child.
7. If the root has two children:
8. Find the in-order successor (or predecessor).
9. Replace the root's value with the in-order successor's (or predecessor's) value.
10. Recursively delete the in-order successor (or predecessor).
11. Return the modified root.
```

Example Usage:

newRoot = deleteNode(root, target)

Example of deleting a node in a Binary Search Tree (BST)

```
50
/ \
30 70
```

/ \ / \

20 40 60 80

Let's say we want to delete the node with the value 30.

50

/ \

30 70

/ \ / \

20 40 60 80

Step 1: Find the Node to Delete

The target node to delete is 30. The node has two children.

Step 2: Find the In-Order Successor (or Predecessor)

The in-order successor of the node to be deleted is the smallest node in its right subtree. In this case, the in-order successor of 30 is 40.

Step 3: Replace the Node Value

Replace the value of the node to be deleted (30) with the value of its in-order successor (40).

Step 4: Recursively Delete the In-Order Successor

Now, recursively delete the in-order successor (40) from its new position.

Final BST:

50

/ \

40 70

/ \

60 80

3.3 AVL Tree

An AVL tree (Adelson-Velsky and Landis tree) is a self-balancing binary search tree. In an AVL tree, the heights of the two child subtrees of any node differ by at most one. If they differ by more than one, rebalancing is performed to restore this property. AVL trees are named after the inventors Adelson-Velsky and Landis.

3.3.1 Properties of AVL Trees:

1. **Binary Search Tree Property:**

Like any binary search tree, the AVL tree maintains the ordering of keys, where for any node x, all nodes in its left subtree have keys less than x, and all nodes in its right subtree have keys greater than x.

2. **Balance Factor:**

For every node in the tree, the absolute difference in heights between its left and right subtrees (called the balance factor) is at most 1. Mathematically, for every node n, |height(n.left) - height(n.right)| <= 1.

3.3.2 AVL Tree Rotations:

To maintain the balance property during insertion and deletion operations, AVL trees use rotations. There are four types of rotations:

1. **Left-Left (LL) Rotation:**

- Perform a right rotation when the left subtree of the left child of a node is higher.
- Suppose we have an AVL tree, and the balance factor of a node (A) becomes greater than 1, indicating that it's left-heavy. Furthermore, the left subtree of A (B) is also left-heavy. In this case, a LL Rotation is performed.

A

/

B

/

C

LL Rotation Process:

Identify the unbalanced node A and its left child B.

Perform a right rotation to balance the tree.

B

/ \

C A

2. **Right-Right (RR) Rotation:**

- Perform a left rotation when the right subtree of the right child of a node is higher.

- Suppose we have an AVL tree, and the balance factor of a node (A) becomes less than -1, indicating that it's right-heavy. Furthermore, the right subtree of A (B) is also right-heavy. In this case, an RR Rotation is performed.

A

\

B

\

C

RR Rotation Process:

- Identify the unbalanced node A and its right child B.
- Perform a left rotation to balance the tree.

B

/ \

A C

3. **Left-Right (LR) Double Rotation:**

- Perform a left rotation on the left child followed by a right rotation on the node when the right subtree of the left child is higher.
- Suppose we have an AVL tree, and the balance factor of a node (A) becomes greater than 1, indicating that it's left-heavy. Furthermore, the right subtree of A (B) is left-heavy. In this case, an LR Rotation is performed.

A

\

B

/

C

LR Rotation Process:

- Perform a left rotation on the right child B.
- Perform a right rotation on the unbalanced node A.

C

/ \

A B

4. **Right-Left (RL) Double Rotation:**

- Perform a right rotation on the right child followed by a left rotation on the node when the left subtree of the right child is higher.
- Suppose we have an AVL tree, and the balance factor of a node (A) becomes less than -1, indicating that it's right-heavy. Furthermore, the left subtree of A (B) is left-heavy. In this case, an RL Rotation is performed.

A

/

B

\

C

RL Rotation Process:

- Perform a right rotation on the left child B.
- Perform a left rotation on the unbalanced node A.

C

/ \

B A

3.3.3 AVL Tree Operations

1. **AVL Tree Creation:**

Suppose we have the following sequence of numbers to insert into an AVL tree: 30, 20, 40, 10, 25, 35, 50.

Insert 30 (Root): 30

Insert 20:

Inserting 20 does not violate the AVL property.

30

/

20

Insert 40:

Inserting 40 creates a right child of 30.

30

/ \

20 40

Insert 10:

Inserting 10 creates a left child of 20.

30

**/ **

20 40

/

10

Inserting 25 creates a right child of 20.

30

**/ **

20 40

**/ **

10 25

Algorithm createAVLTree(root, key):

1. If the root is NULL, create a new node with the key and return it.
2. If the key is less than the root's key, recursively insert in the left subtree.
3. If the key is greater than the root's key, recursively insert in the right subtree.
4. Update the height of the current node.
5. Calculate the balance factor of the current node.
6. If the node becomes unbalanced, perform the necessary rotations to restore balance.
7. Return the modified root.

2. **Insertion in AVL Tree:**

Algorithm insertAVL(root, key):

1. If the root is NULL, create a new node with the key and return it.

2. If the key is less than the root's key, recursively insert in the left subtree.

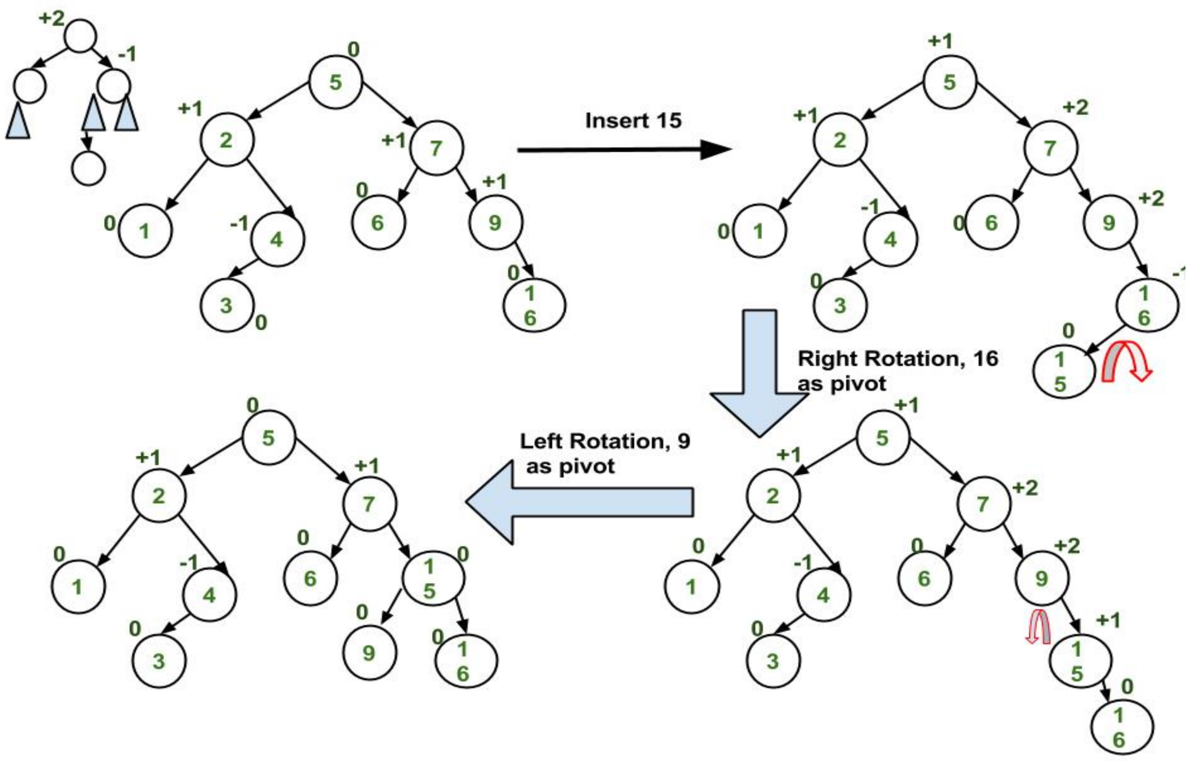
3. If the key is greater than the root's key, recursively insert in the right subtree.

4. Update the height of the current node.

5. Calculate the balance factor of the current node.

6. If the node becomes unbalanced, perform the necessary rotations to restore balance.

7. Return the modified root.



3. **Deletion in AVL Tree:**

Algorithm deleteAVL(root, key):

1. If the root is NULL, return NULL.

2. If the key is less than the root's key, recursively delete in the left subtree.

3. If the key is greater than the root's key, recursively delete in the right subtree.

4. If the key is equal to the root's key, perform the deletion:

a. If the node has no children or one child, remove the node.

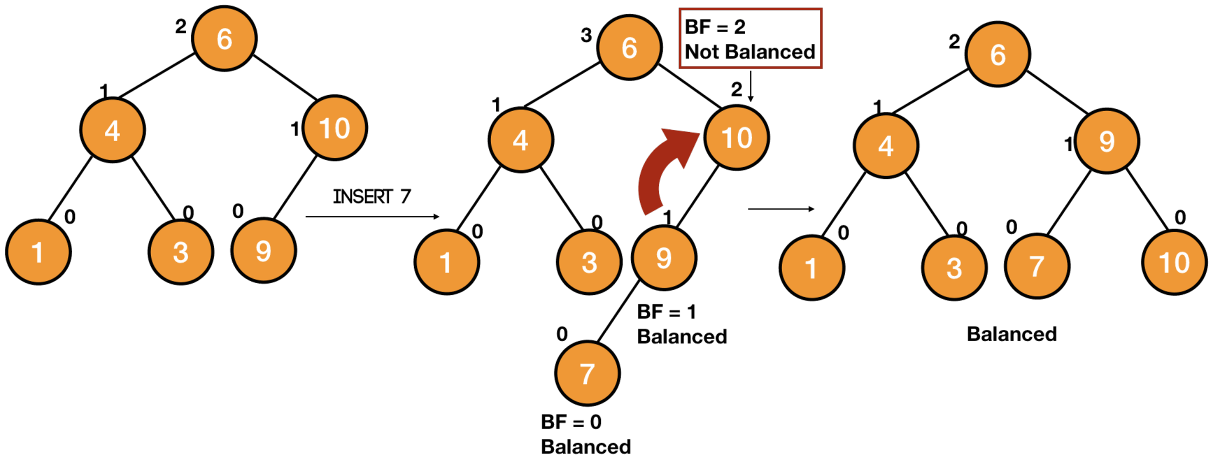
b. If the node has two children, find the in-order successor, replace the node's key with the successor's key, and recursively delete the successor.

5. Update the height of the current node.

6. Calculate the balance factor of the current node.

7. If the node becomes unbalanced, perform the necessary rotations to restore balance.

8. Return the modified root.



4. **Searching in AVL tree**

Algorithm searchAVL(root, key):

1. If the root is NULL or the key is equal to the root's key, return the root.

2. If the key is less than the root's key, recursively search in the left subtree.

3. If the key is greater than the root's key, recursively search in the right subtree.

Example Usage:

node = searchAVL(root, key)

if node is not NULL:

// Key found

else:

// Key not found

C-Code for AVT Tree Implementation:

#include <stdio.h>

#include <stdlib.h>

// **Structure for a node in the AVL tree** struct AVLNode {

int key;

struct AVLNode* left;

```
struct AVLNode* right;

int height;

};

// Function to get the height of a node

int height(struct AVLNode* node) {

if (node == NULL)

return 0;

return node->height;

}

// Function to get the maximum of two integers

int max(int a, int b) {

return (a > b) ? a : b;

}

// Function to create a new AVL node

struct AVLNode* newNode(int key) {

struct AVLNode* node = (struct AVLNode*)malloc(sizeof(struct AVLNode));

node->key = key;

node->left = NULL;

node->right = NULL;

node->height = 1; // New node is initially at height 1

return node;

}

// Function to right rotate a subtree rooted with y

struct AVLNode* rightRotate(struct AVLNode* y) {

struct AVLNode* x = y->left;

struct AVLNode* T2 = x->right;

// Perform rotation

x->right = y;
```

```
y->left = T2;

// Update heights

y->height = max(height(y->left), height(y->right)) + 1;

x->height = max(height(x->left), height(x->right)) + 1;

// Return new root

return x;

}

// Function to left rotate a subtree rooted with x

struct AVLNode* leftRotate(struct AVLNode* x) {

struct AVLNode* y = x->right;

struct AVLNode* T2 = y->left;

// Perform rotation

y->left = x;

x->right = T2;

// Update heights

x->height = max(height(x->left), height(x->right)) + 1;

y->height = max(height(y->left), height(y->right)) + 1;

// Return new root

return y;

}

// Function to get the balance factor of a node

int getBalance(struct AVLNode* node) {

if (node == NULL)

return 0;

return height(node->left) - height(node->right);

}

// Function to insert a key into the AVL tree

struct AVLNode* insert(struct AVLNode* root, int key) {
```

```
// Perform the normal BST insertion

if (root == NULL)

return newNode(key);

if (key < root->key)

root->left = insert(root->left, key);

else if (key > root->key)

root->right = insert(root->right, key);

else // Duplicate keys are not allowed in AVL tree

return root;

// Update height of the current node

root->height = 1 + max(height(root->left), height(root->right));

// Get the balance factor to check whether this node became unbalanced

int balance = getBalance(root);

// Left Left Case

if (balance > 1 && key < root->left->key)

return rightRotate(root);

// Right Right Case

if (balance < -1 && key > root->right->key)

return leftRotate(root);

// Left Right Case

if (balance > 1 && key > root->left->key) {

root->left = leftRotate(root->left);

return rightRotate(root);

}

// Right Left Case

if (balance < -1 && key < root->right->key) {

root->right = rightRotate(root->right);

return leftRotate(root);
}

// No rotation needed, return the unchanged node

return root;

}
```

// Function to print the inorder traversal of the AVL tree

```
void inOrderTraversal(struct AVLNode* root) {

if (root != NULL) {

inOrderTraversal(root->left);

printf("%d ", root->key);

inOrderTraversal(root->right);

}

}
```

// Driver program to test the AVL tree implementation

```
int main() {

struct AVLNode* root = NULL;

    root = insert(root, 10);

    root = insert(root, 20);

    root = insert(root, 30);

    root = insert(root, 40);

    root = insert(root, 50);

    root = insert(root, 25);

printf("Inorder traversal of the constructed AVL tree: ");

inOrderTraversal(root);

printf("\n");

return 0;

}
```

3.4 B-Tree

B-Tree, the multi-talented data structure that can handle massive amounts of data with ease. When it comes to storing and searching large amounts of data, traditional binary search trees can become impractical due to their poor performance and high memory usage. B-Trees, also known as B-Tree or Balanced Tree, are a type of self-balancing tree that was specifically designed to overcome these limitations.

Unlike traditional binary search trees, B-Trees are characterized by the large number of keys that they can store in a single node, which is why they are also known as "large key" trees. Each node in a B-Tree can contain multiple keys, which allows the tree to have a larger branching factor and thus a shallower height. This shallow height leads to less disk I/O, which results in faster search and insertion operations. B-Trees are particularly well suited for storage systems that have slow, bulky data access such as hard drives, flash memory, and CD-ROMs.

B-Trees maintain balance by ensuring that each node has a minimum number of keys, so the tree is always balanced. This balance guarantees that the time complexity for operations such as insertion, deletion, and searching is always O(log n), regardless of the initial shape of the tree.

Time Complexity of B-Tree:

Sr. No.	**Algorithm**	**Time Complexity**
1.	Search	O(log n)
2.	Insert	O(log n)
3.	Delete	O(log n)

Note: "n" is the total number of elements in the B-tree

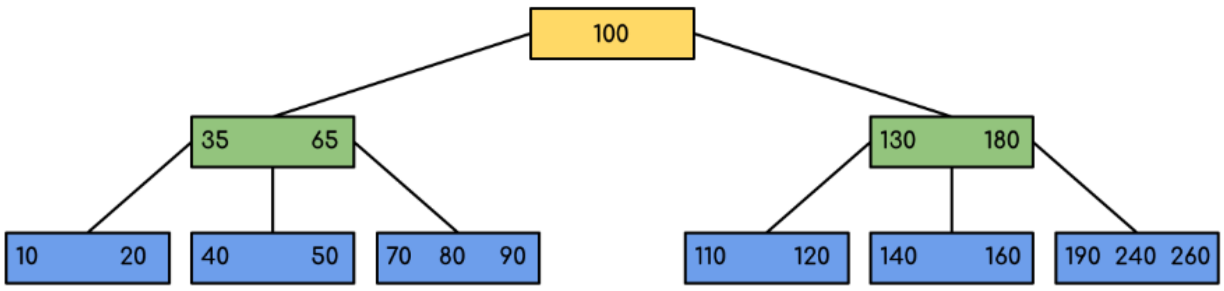
Properties of B-Tree:

- All leaves are at the same level.
- B-Tree is defined by the term minimum degree '**t**'. The value of '**t**' depends upon disk block size.
- Every node except the root must contain at most t-1 keys. The root may contain a minimum of **1** key.
- All nodes (including root) may contain at most (**2*t** – **1**) keys.
- Number of children of a node is equal to the number of keys in it plus **1**.
- All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.
- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.

- Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(log n).
- Insertion of a Node in B-Tree happens only at Leaf Node.

Following is an example of a B-Tree of minimum order 5

Note: that in practical B-Trees, the value of the minimum order is much more than 5.



We can see in the above diagram that all the leaf nodes are at the same level and all non-leafs have no empty sub-tree and have keys one less than the number of their children.

Interesting Facts about B-Trees:

- *The minimum height of the B-Tree that can exist with n number of nodes and m is the maximum number of children of a node can have is:*
- *The maximum height of the B-Tree that can exist with n number of nodes and t is the minimum number of children that a non-root node can have is: and*

Traversal in B-Tree:

Traversal is also similar to Inorder traversal of Binary Tree. We start from the leftmost child, recursively print the leftmost child, then repeat the same process for the remaining children and keys. In the end, recursively print the rightmost child.

Search Operation in B-Tree:

Search is similar to the search in Binary Search Tree. Let the key to be searched is k.

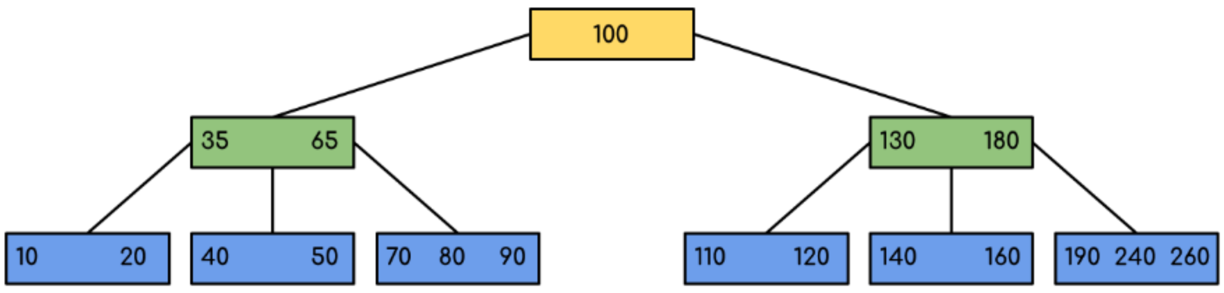
- Start from the root and recursively traverse down.
- For every visited non-leaf node,
  - If the node has the key, we simply return the node.
  - Otherwise, we recur down to the appropriate child (The child which is just before the first greater key) of the node.
- If we reach a leaf node and don't find k in the leaf node, then return NULL.

Searching a B-Tree is similar to searching a binary tree. The algorithm is similar and goes with recursion. At each level, the search is optimized as if the key value is not present in the range of the parent then the key is present in another branch. As these values limit the search they are also known as limiting values or separation values. If we reach a leaf node and don't find the desired key then it will display NULL.

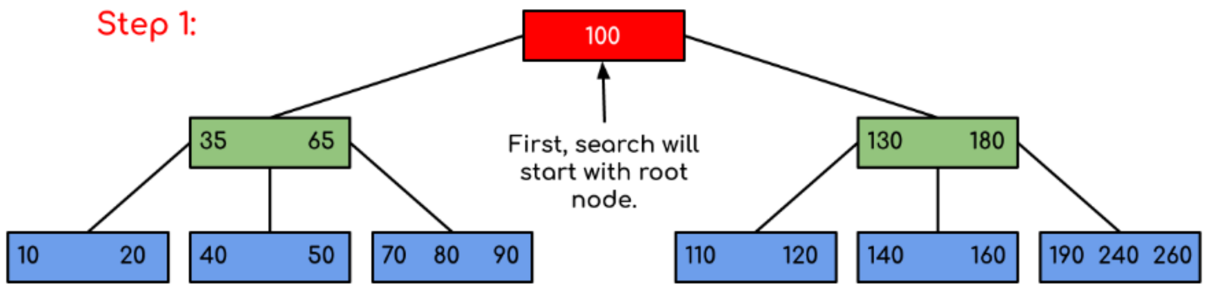
Algorithm for Searching an Element in a B-Tree:-

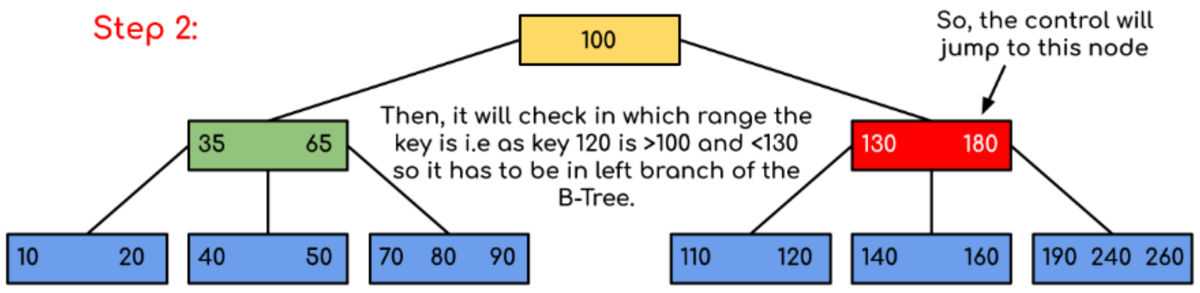
```
BtreeSearch(x, k)
i = 1
// n[x] means number of keys in x node
while i ≤ n[x] and k ≥ keyi[x]
do i = i + 1
if i n[x] and k = keyi[x]
then return (x, i)
if leaf [x]
then return NIL
else
return BtreeSearch(ci[x], k)
```

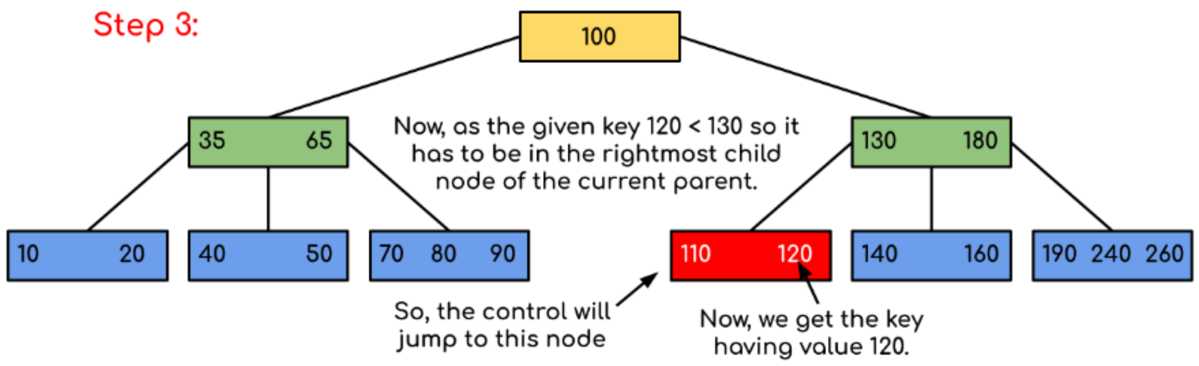
Input: *Search 120 in the given B-Tree.*



Solution:




Step 2:
100
So, the control will jump to this node
35 65
Then, it will check in which range the key is i.e as key 120 is >100 and <130 so it has to be in left branch of the B-Tree.
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260


Step 3:
100
35 65
Now, as the given key 120 < 130 so it has to be in the rightmost child node of the current parent.
130 180
10 20
40 50
70 80 90
110 120
140 160
190 240 260
So, the control will jump to this node
Now, we get the key having value 120.

In this example, we can see that our search was reduced by just limiting the chances where the key containing the value could be present. Similarly if within the above example we've to look for 180, then the control will stop at step 2 because the program will find that the key 180 is present within the current node. And similarly, if it's to seek out 90 then as 90 < 100 so it'll go to the left subtree automatically, and therefore the control flow will go similarly as shown within the above example.

There are two conventions to define a B-Tree, one is to define by minimum degree, second is to define by order. We have followed the minimum degree convention and will be following the same in coming posts on B-Tree. The variable names used in the above program are also kept the same

Applications of B-Trees:

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.
- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

Advantages of B-Trees:

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

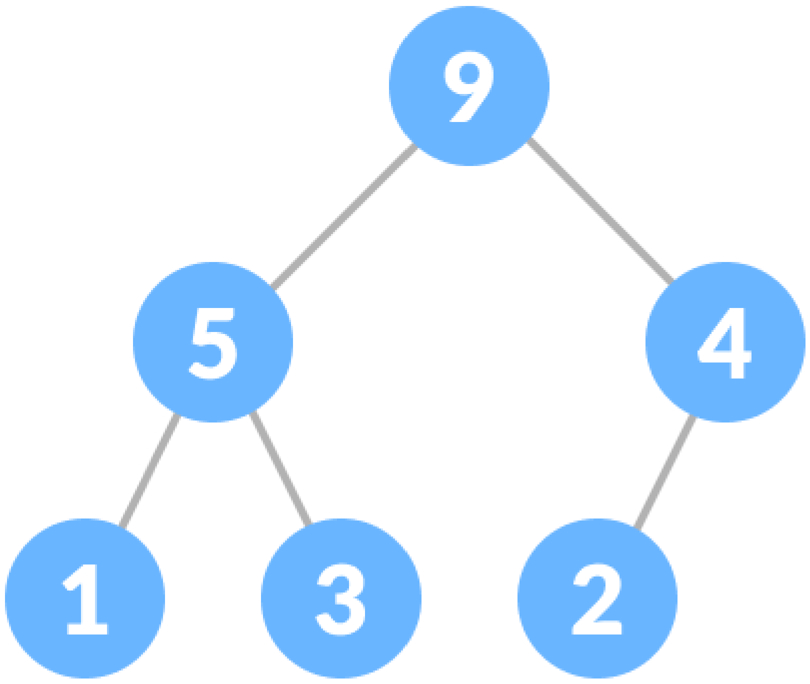
Disadvantages of B-Trees:

- B-Trees are based on disk-based data structures and can have a high disk usage.
- Not the best for all cases.
- Slow in comparison to other data structures.

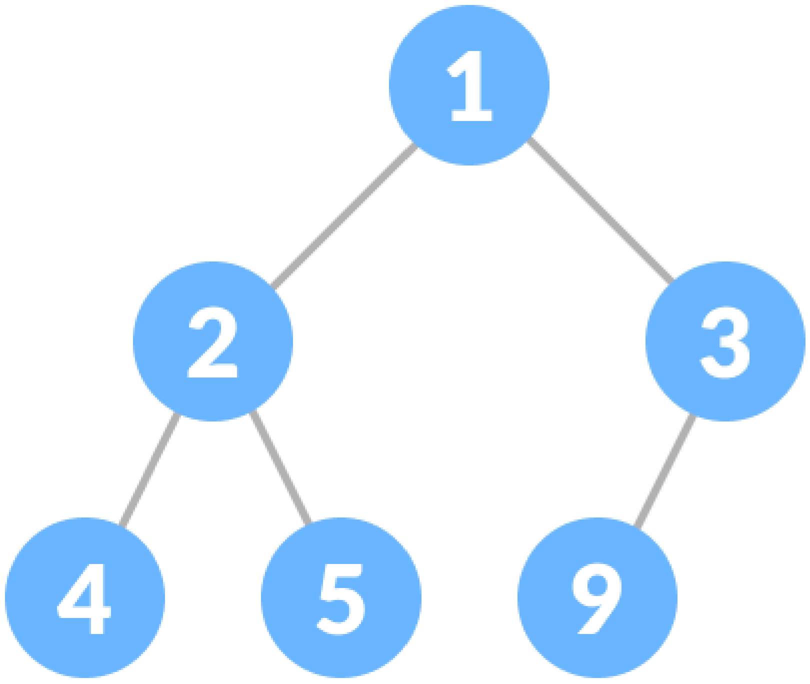
4 Heap.

Heap data structure is a complete binary tree that satisfies **the heap property**, where any given node is

- always greater than its child node/s and the key of the root node is the largest among all other nodes. This property is also called **max heap property**.
- always smaller than the child node/s and the key of the root node is the smallest among all other nodes. This property is also called **min heap property**.



Max-heap



Min-heap

This type of data structure is also called a **binary heap**.

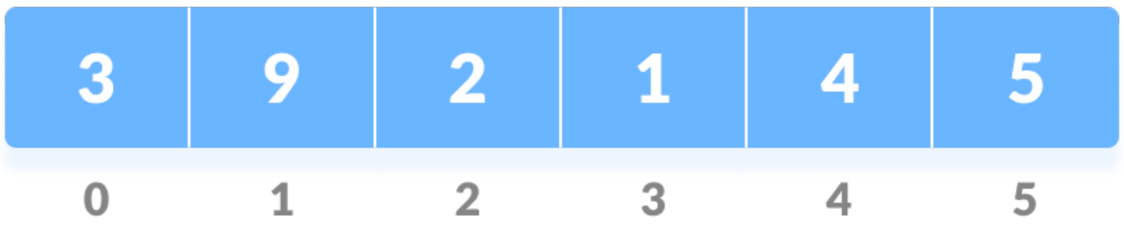
4.1 Heap Operations

Some of the important operations performed on a heap are described below along with their algorithms.

4.1.1 Heapify

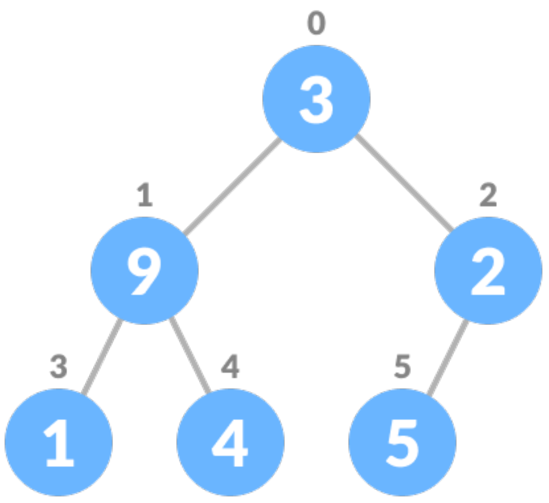
Heapify is the process of creating a heap data structure from a binary tree. It is used to create a Min-Heap or a Max-Heap.

1. Let the input array be



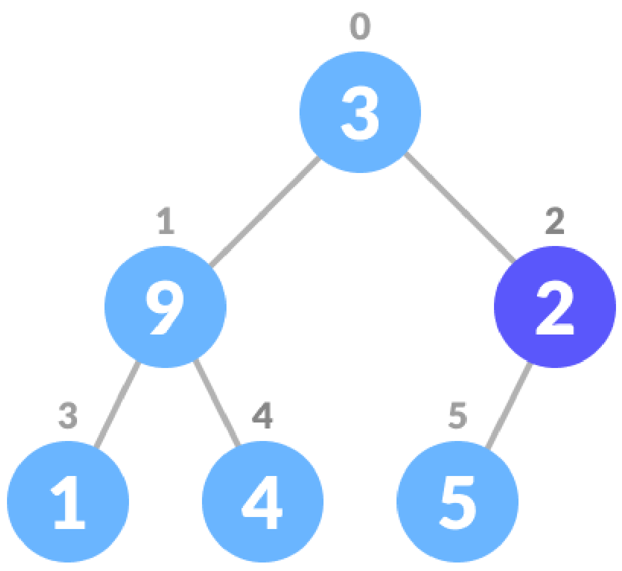
Initial Array

2. Create a complete binary tree from the array



Complete binary tree

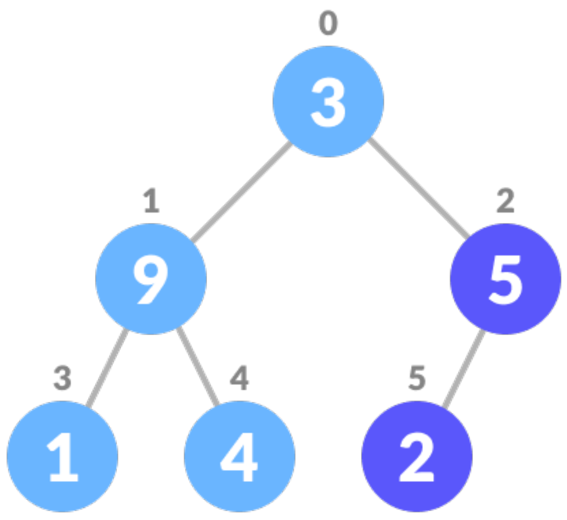
3. Start from the first index of non-leaf node whose index is given by n/2 -



4. Start from the first on leaf node
5. Set current element i as largest.
6. The index of left child is given by 2i + 1 and the right child is given by 2i + 2.

   If leftChild is greater than currentElement (i.e. element at ith index), set leftChildIndex as largest.

   If rightChild is greater than element in largest, set rightChildIndex as largest.
7. Swap largest with currentElement



8. Swap if necessary
9. Repeat steps 3-7 until the subtrees are also heapified.

Algorithm

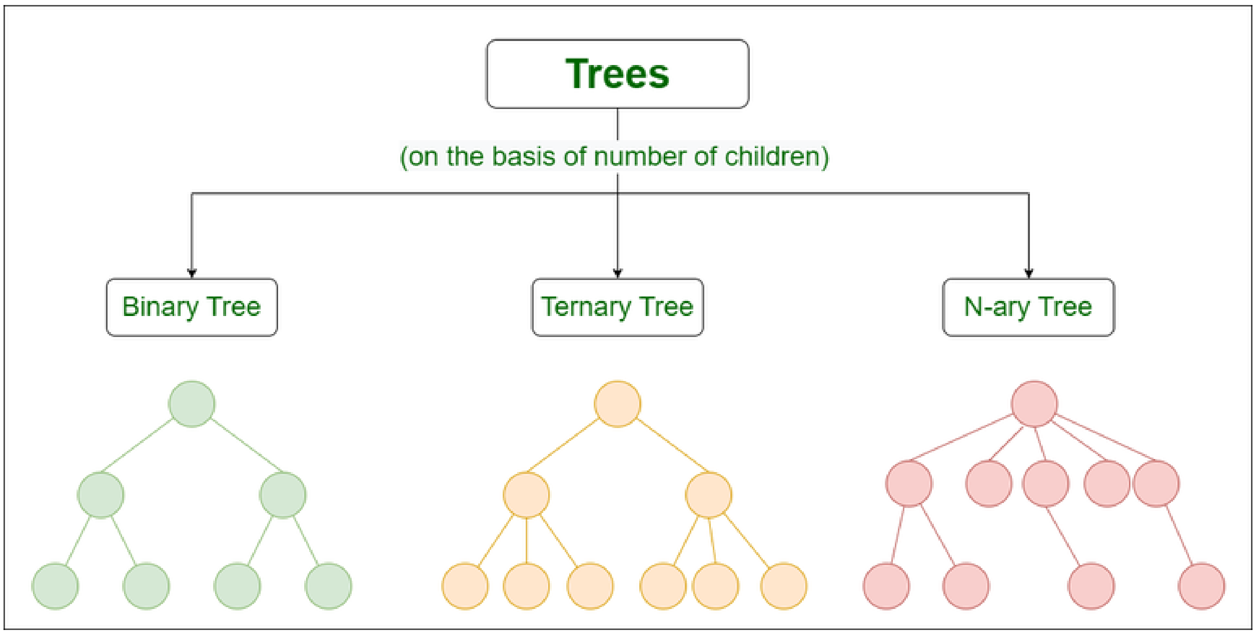
Heapify(array, size, i)

set i as largest

```
leftChild = 2i + 1
rightChild = 2i + 2
if leftChild > array[largest]
set leftChildIndex as largest
if rightChild > array[largest]
set rightChildIndex as largest
swap array[i] and array[largest]
```

5 Representation and Implementations of different types of trees

5.1 Types of Trees in Data Structure based on the number of children:



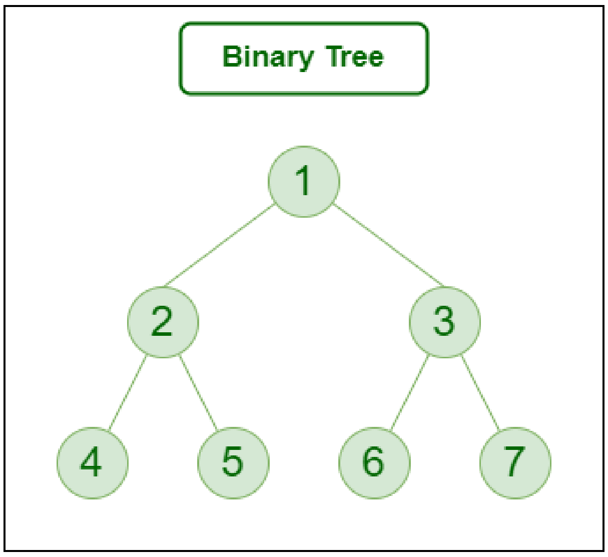
Types of Trees in Data Structure based on the number of children

5.1.1 Binary Tree

A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Example:

Consider the tree below. Since each node of this tree has only 2 children, it can be said that this tree is a Binary Tree



Binary Tree

5.1.2 Types of Binary Tree:

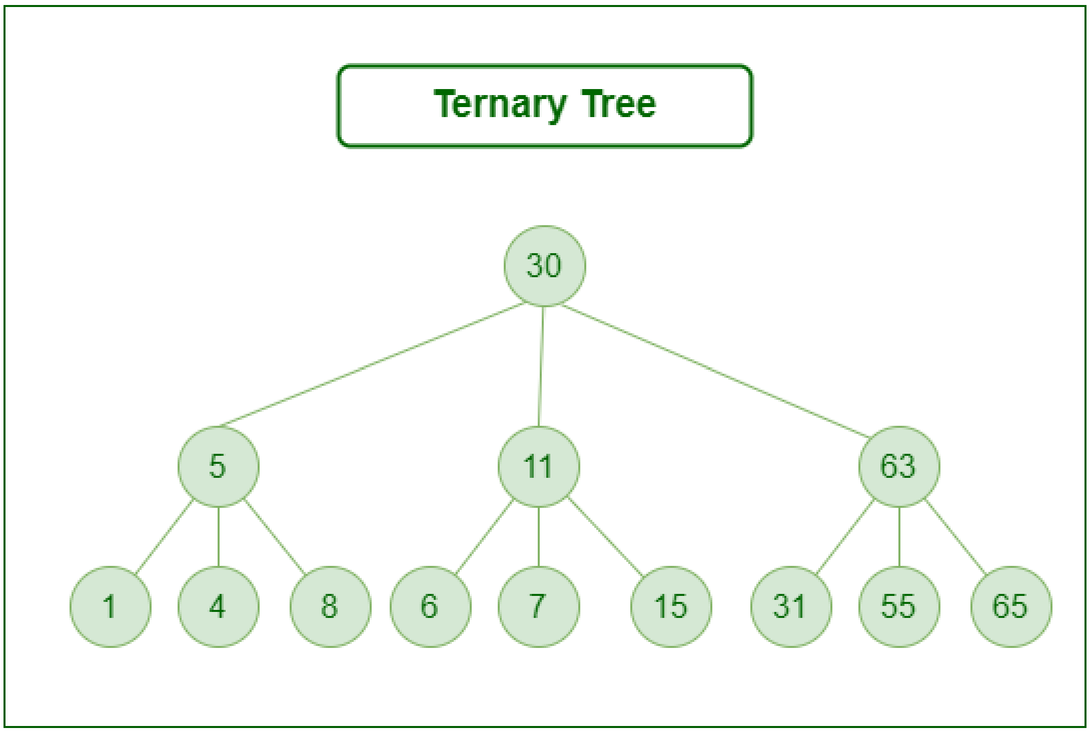
- Binary Tree consists of following types **based on the number of children**:
  1. Full Binary Tree
  2. Degenerate Binary Tree
- **On the basis of completion of levels**, the binary tree can be divided into following types:
  1. Complete Binary Tree
  2. Perfect Binary Tree
  3. Balanced Binary Tree

5.2 Ternary Tree

A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as "left", "mid" and "right".

Example:

Consider the tree below. Since each node of this tree has only 3 children, it can be said that this tree is a Ternary Tree



Ternary Tree

5.2.1 Types of Ternary Tree:

Ternary Search Tree

A ternary search tree is a special trie data structure where the child nodes of a standard trie are ordered as a binary search tree.

Unlike trie(standard) data structure where each node contains 26 pointers for its children, each node in a ternary search tree contains only 3 pointers:

1. The left pointer points to the node whose value is less than the value in the current node.
2. The equal pointer points to the node whose value is equal to the value in the current node.
3. The right pointer points to the node whose value is greater than the value in the current node.

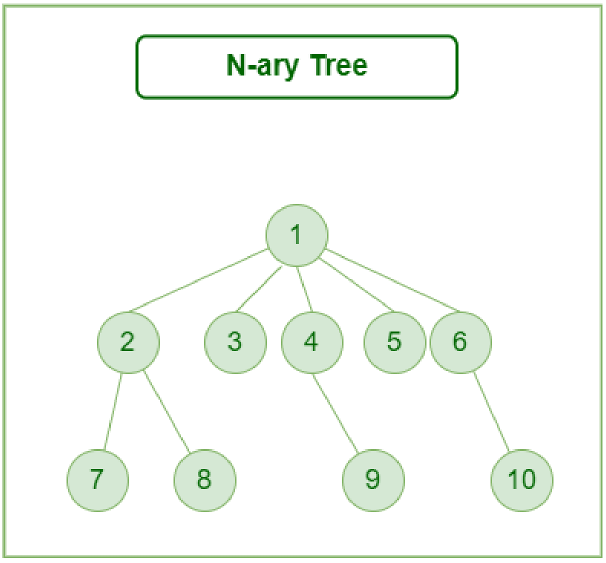
5.3 N-ary Tree (Generic Tree)

Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

Every node stores the addresses of its children and the very first node's address will be stored in a separate pointer called root.

1. Many children at every node.
2. The number of nodes for each node is not known in advance.

Example:



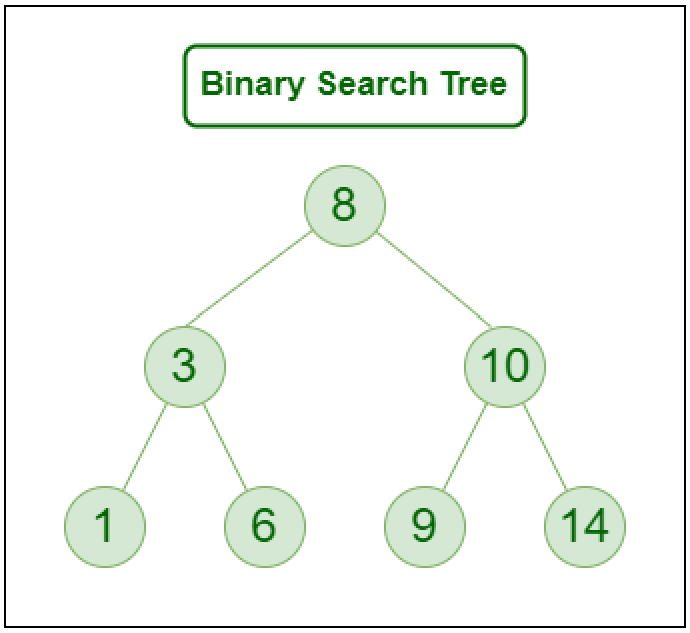
N-ary Tree

Special Types of Trees in Data Structure based on the nodes' values:

5.4 Binary Search Tree

A **binary Search Tree** is a node-based binary tree data structure that has the following properties:

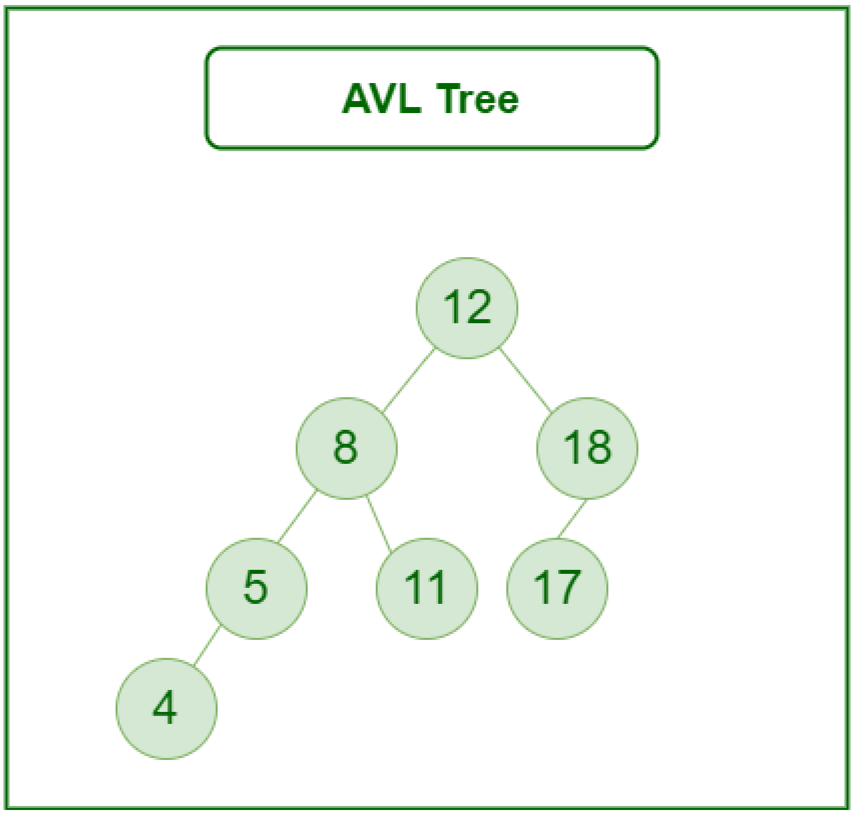
- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.



Binary Search Tree

5.5 AVL Tree

AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.



AVL Tree

5.6 Red-Black Tree

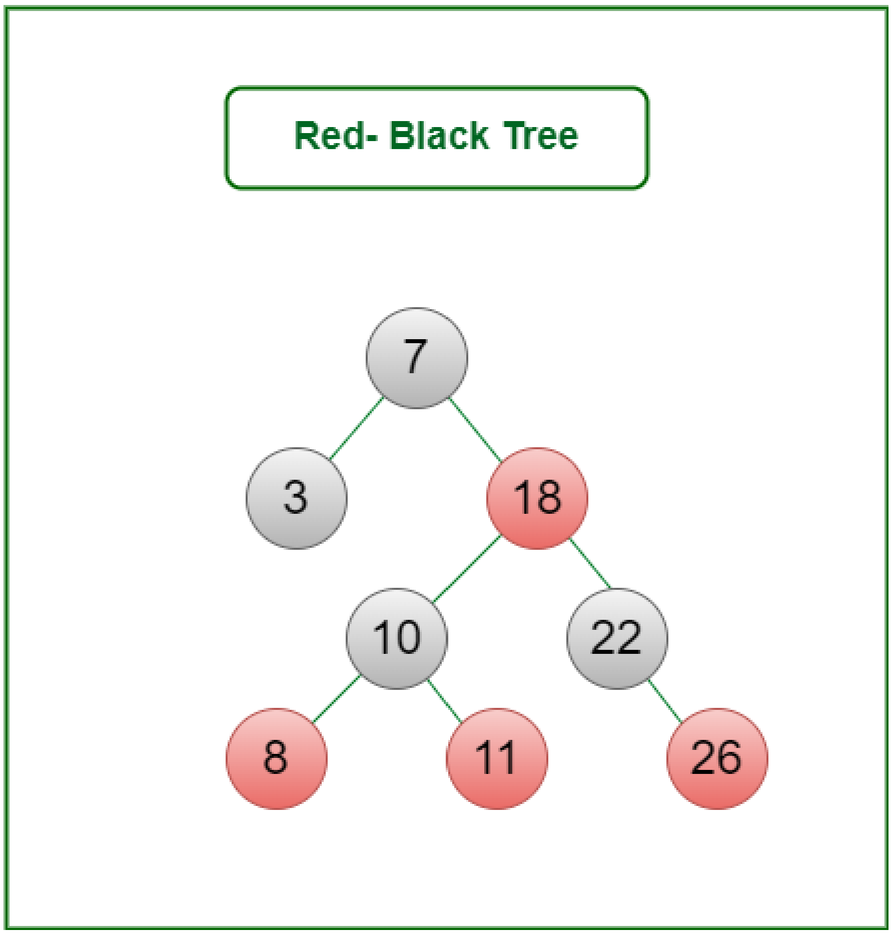
A red-black tree is a kind of self-balancing binary search tree where each node has an extra bit, and that bit is often interpreted as the color (red or black). These colors are used to ensure that the tree remains balanced during insertions and deletions.

Although the balance of the tree is not perfect, it is good enough to reduce the searching time and maintain it around O(log n) time, where n is the total number of elements in the tree.

Rules That Every Red-Black Tree Follows:

1. Every node has a color either red or black.
2. The root of the tree is always black.
3. There are no two adjacent red nodes (A red node cannot have a red parent or red child).
4. Every path from a node (including root) to any of its descendants' NULL nodes has the same number of black nodes.

5. All leaf (NULL) nodes are black nodes.



Red-Black Tree

5.7 B-Tree

B-Tree is a self-balancing search tree. In most of the other self-balancing search trees (like AVL and Red-Black Trees), it is assumed that everything is in the main memory.

5.8 Properties of B-Tree:

- All leaves are at the same level.
- B-Tree is defined by the term minimum degree '**t**'. The value of '**t**' depends upon disk block size.
- Every node except the root must contain at least t-1 keys. The root may contain a minimum of **1** key.
- All nodes (including root) may contain at most (**2*t** − **1**) keys.
- The number of children of a node is equal to the number of keys in it plus **1**.
- All keys of a node are sorted in increasing order. The child between two keys **k1** and **k2** contains all keys in the range from **k1** and **k2**.

- B-Tree grows and shrinks from the root which is unlike Binary Search Tree. Binary Search Trees grow downward and also shrink from downward.
- Like other balanced Binary Search Trees, the time complexity to search, insert and delete is O(log n).
- Insertion of a Node in B-Tree happens only at Leaf Node.

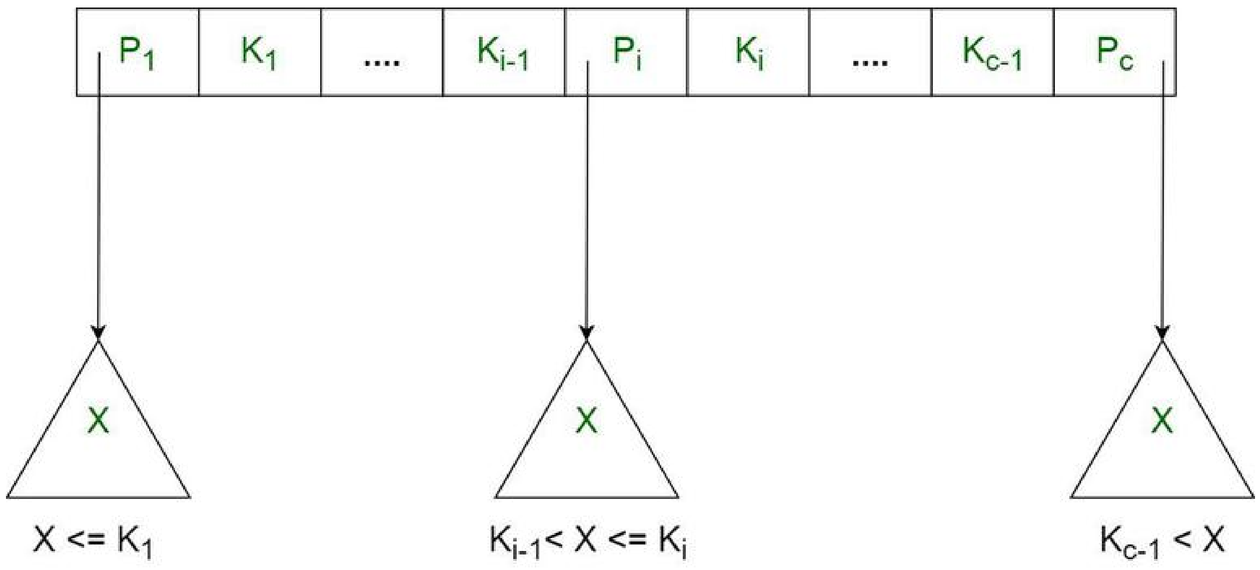
5.9 B+ Tree

B+ tree eliminates the drawback B-tree used for indexing by storing data pointers only at the leaf nodes of the tree. Thus, the structure of leaf nodes of a B+ tree is quite different from the structure of internal nodes of the B tree.

It may be noted here that, since data pointers are present only at the leaf nodes, the leaf nodes must necessarily store all the key values along with their corresponding data pointers to the disk file block, to access them. Moreover, the leaf nodes are linked to providing ordered access to the records. The leaf nodes, therefore form the first level of the index, with the internal nodes forming the other levels of a multilevel index. Some of the key values of the leaf nodes also appear in the internal nodes, to simply act as a medium to control the search of a record.

The structure of the internal nodes of a B+ tree of order 'a' is as follows:

1. Each internal node is of the form: <P1, K1, P2, K2, ..., Pc-1, Kc-1, Pc> where c <= a and each **Pi is a tree pointer (i.e points to another node of the tree)** and, each **Ki is a key-value** (see diagram-I for reference).
2. Every internal node has : K1 < K2 < . . . . < Kc-1
3. For each search field values 'X' in the sub-tree pointed at by Pi, the following condition holds: Ki-1 < X <= Ki, for 1 < i < c and, Ki-1 < X, for i = c (See diagram I for reference)
4. Each internal node has at most '**a**' tree pointers.
5. The root node has, at least two tree pointers, while the other internal nodes have at least \ceil(a/2) tree pointers each.
6. If an internal node has 'c' pointers, c <= a, then it has 'c – 1' key values.

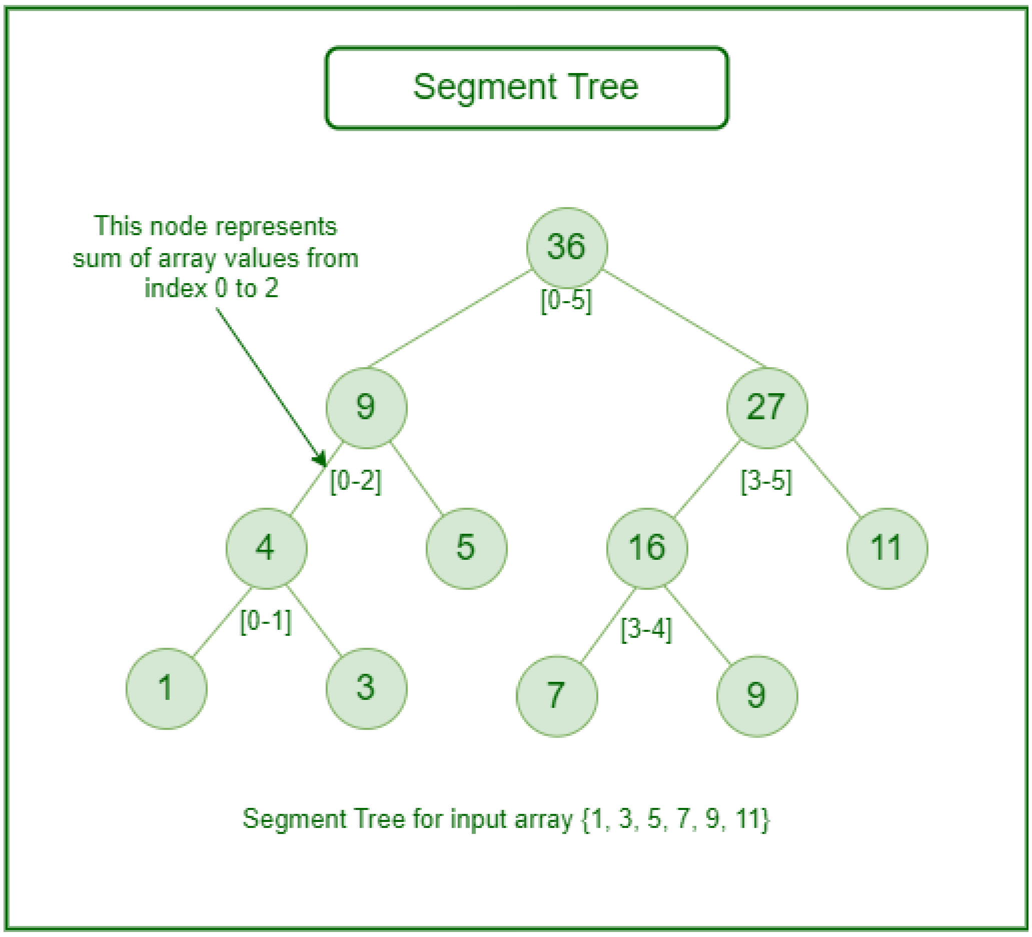


B+ Tree

5.10 Segment Tree

In computer science, a Segment Tree, also known as a statistic tree, is a tree data structure used for storing information about intervals, or segments. It allows querying which of the stored segments contain a given point. It is, in principle, a static structure; that is, it's a structure that cannot be modified once it's built. A similar data structure is the interval tree.

A **segment tree** for a set I of n intervals uses O(n log n) storage and can be built in O(n log n) time. Segment trees support searching for all the intervals that contain a query point in time O(log n + k), k being the number of retrieved intervals or segments.


Segment Tree
This node represents sum of array values from index 0 to 2
36
[0-5]
9
27
[0-2]
[3-5]
4
5
16
11
[0-1]
[3-4]
1
3
7
9
Segment Tree for input array {1, 3, 5, 7, 9, 11}

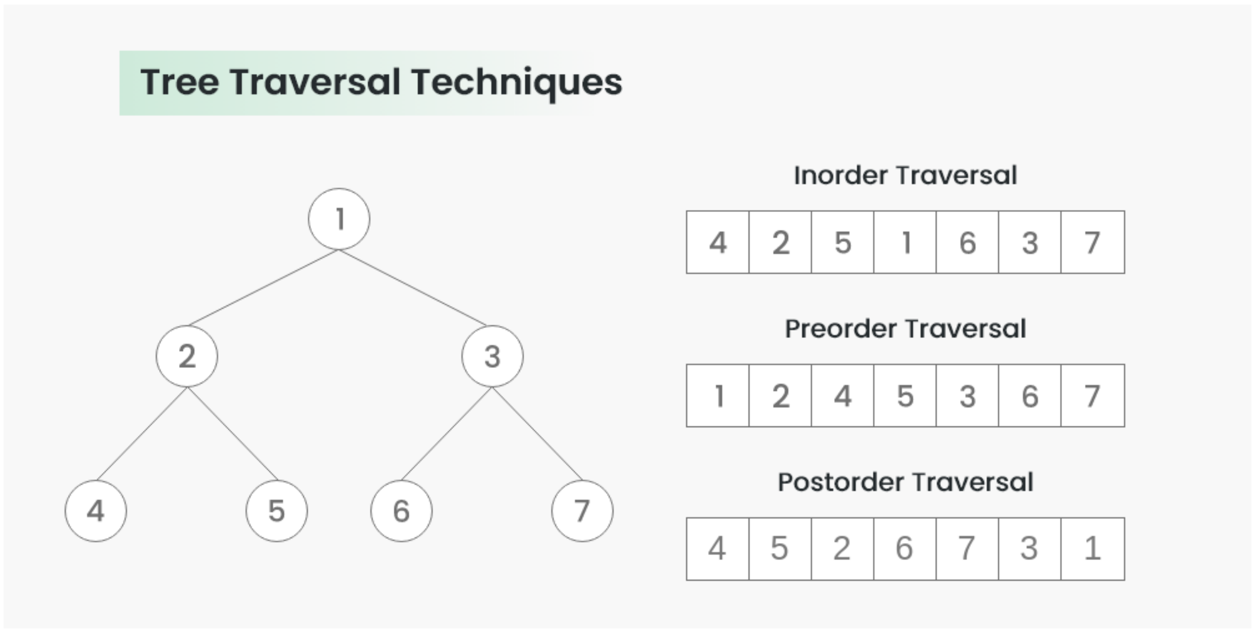
Segment Tree

6 Tree Traversal algorithms

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

A Tree Data Structure can be traversed in following ways:

1. Depth First Search or DFS
    1. Inorder Traversal
    2. Preorder Traversal
    3. Postorder Traversal
2. Level Order Traversal or Breadth First Search or BFS
3. Boundary Traversal
4. Diagonal Traversal



Tree Traversal

6.1 Inorder Traversal:

Algorithm Inorder(tree)

1. *Traverse the left subtree, i.e., call Inorder(left->subtree)*
2. *Visit the root.*
3. *Traverse the right subtree, i.e., call Inorder(right->subtree)*

6.1.1 Uses of Inorder Traversal:

In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

Time Complexity: O(N)
Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

6.2 Preorder Traversal:

Algorithm Preorder(tree)

1. *Visit the root.*
2. *Traverse the left subtree, i.e., call Preorder(left->subtree)*
3. *Traverse the right subtree, i.e., call Preorder(right->subtree)*

6.2.1 Uses of Preorder:

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.

Time Complexity: O(N)
Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

6.3 Postorder Traversal:

Algorithm Postorder(tree)

1. *Traverse the left subtree, i.e., call Postorder(left->subtree)*
2. *Traverse the right subtree, i.e., call Postorder(right->subtree)*
3. *Visit the root*

6.3.1 Uses of Postorder:

Postorder traversal is used to delete the tree. Postorder traversal is also useful to get the postfix expression of an expression tree.

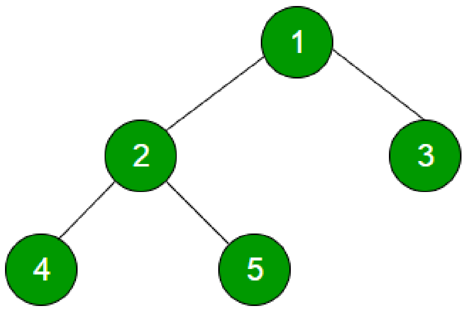
Some of the other tree traversals are:

6.4 Level Order Treversal

For each node, first, the node is visited and then it's child nodes are put in a FIFO queue. Then again the first node is popped out and then it's child nodes are put in a FIFO queue and repeat until queue becomes empty.

Example:

Input:



Level Order Treversal:

1

2 3

4 5

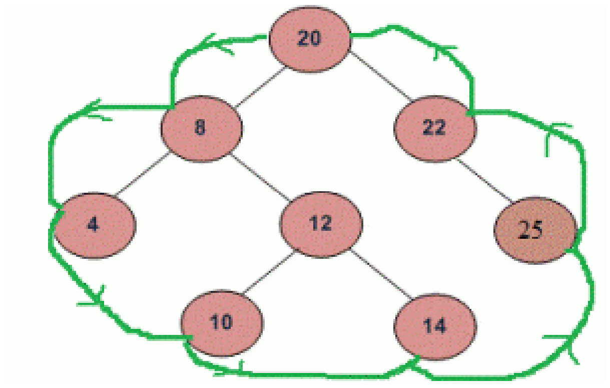
6.5 Boundary Traversal

The Boundary Traversal of a Tree includes:

1. left boundary (nodes on left excluding leaf nodes)

2. leaves (consist of only the leaf nodes)

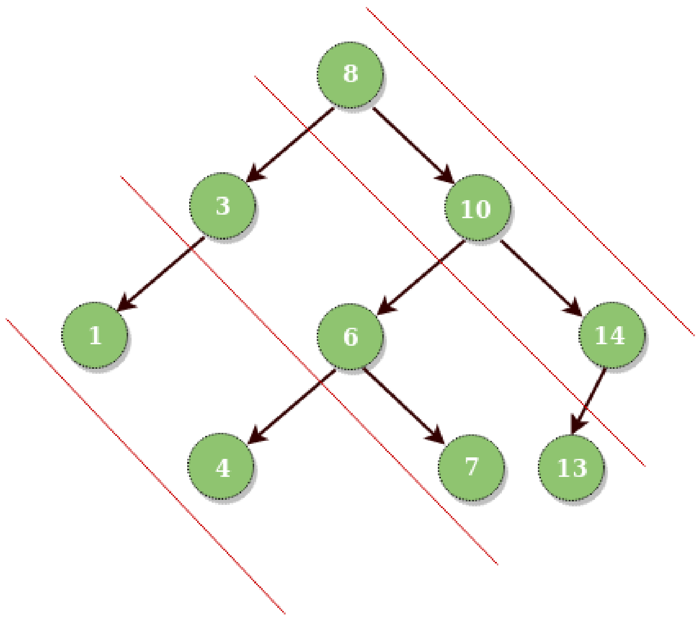
3. right boundary (nodes on right excluding leaf nodes)



6.6 Diagonal Traversal

In the Diagonal Traversal of a Tree, all the nodes in a single diagonal will be printed one by one.

Input :



Diagonal Traversal of binary tree:

$8\ 10\ 14$

$3\ 6\ 7\ 13$

$1\ 4$