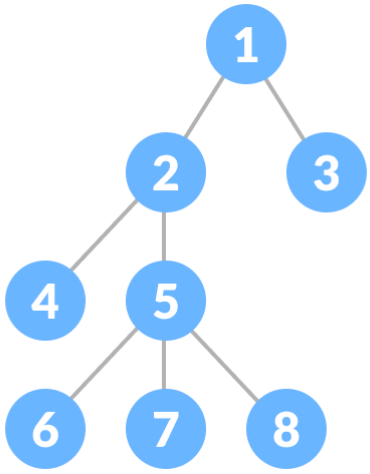
TREES

Definition

Tree data structure

The tree is a very commonly encountered data shape that allows us to represent hierarchical relationships. A tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.



It turns out that many of the structures we encounter when writing software are hierarchical. For instance, every file and directory within a file system is “inside” one and only one parent directory, up to the root directory. In an HTML document, every tag is inside one and only one parent tag, up to the root (html) tag.

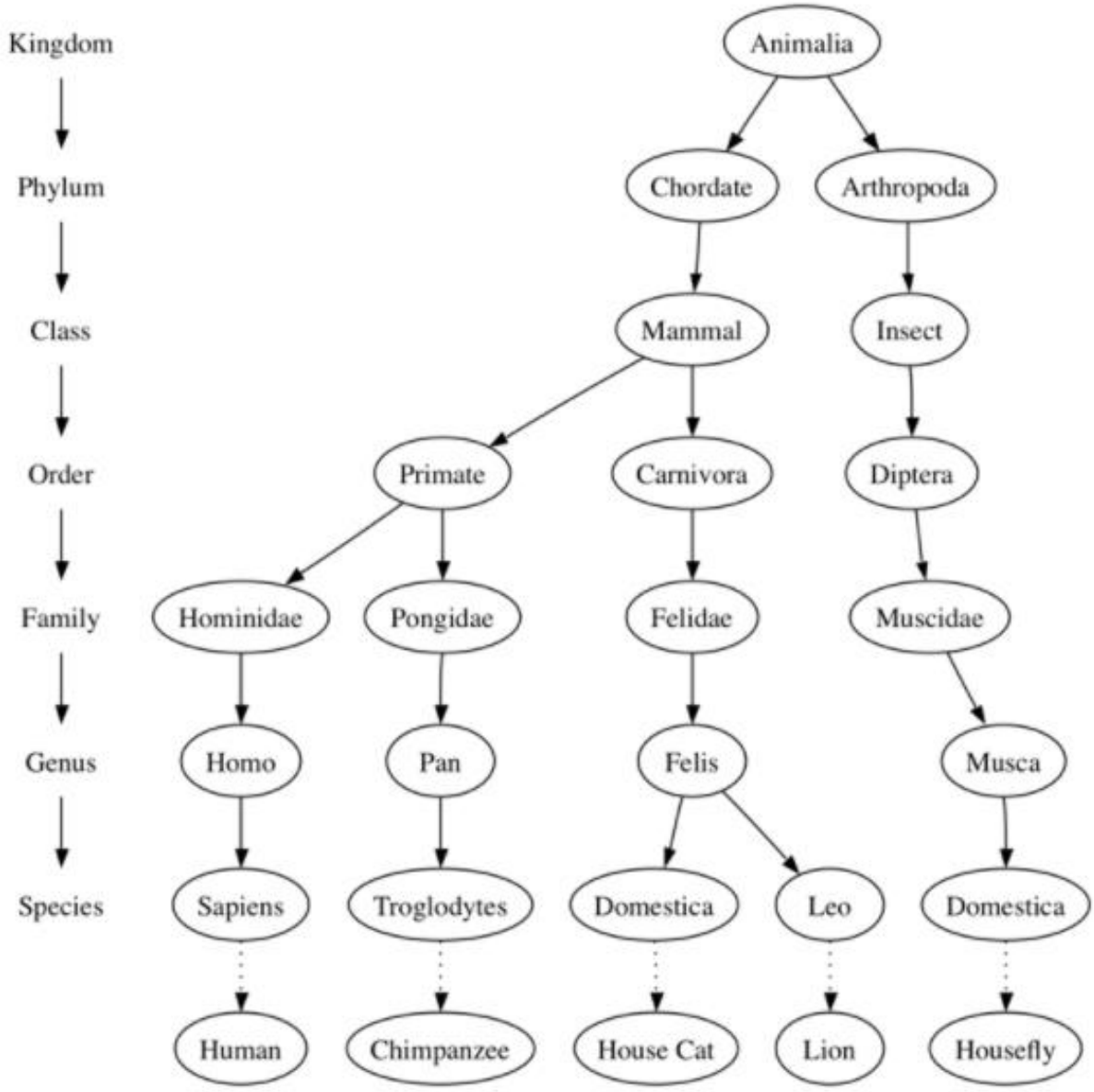
It also turns out that that we can use trees to implement useful data structures like maps, and to do fast searches. We will cover some of the many use cases for trees in this section, as well as exploring algorithms to traverse through trees.

Examples of trees

Tree data structures have many things in common with their botanical cousins. Both have a root, branches, and leaves. One difference is that we find it more intuitive to consider the root of a tree data structure to be at the “top”, for instance that the root of a file system is “above” its subdirectories.

Before we begin our study of tree data structures, let's look at a few common examples.

Our first example of a tree is a classification tree from biology. The illustration below shows an example of the biological classification of some animals. From this simple example, we can learn about several properties of trees. The first property this example demonstrates is that trees are hierarchical. By hierarchical, we mean that trees are structured in layers with the more general things near the top and the more specific things near the bottom. The top of the hierarchy is the Kingdom, the next layer of the tree (the "children" of the layer above) is the Phylum, then the Class, and so on. However, no matter how deep we go in the classification tree, all the organisms are still animals.



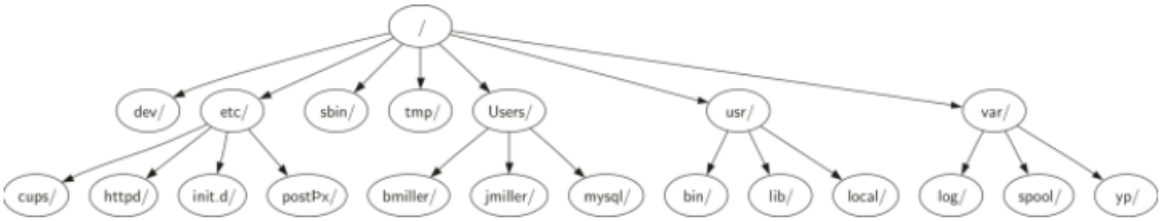
Taxonomy of some common animals shown as a tree

Notice that you can start at the top of the tree and follow a path made of circles and arrows all the way to the bottom. At each level of the tree we might ask ourselves a question and then follow the path that agrees with our answer. For example, we might ask, “Is this animal a Chordate or an Arthropod?” If the answer is “Chordate” then we follow that path and ask, “Is this Chordate a Mammal?” If not, we are stuck (but only in this simplified example). When we are at the Mammal level we ask, “Is this Mammal a Primate or a Carnivore?” We can keep following paths until we get to the very bottom of the tree where we have the common name.

A second property of trees is that all of the children of one node are independent of the children of another node. For example, the Genus Felis has the children Domestica and Leo. The Genus Musca also has a child named Domestica, but it is a different node and is independent of the Domestica child of Felis. This means that we can change the node that is the child of Musca without affecting the child of Felis.

A third property is that each leaf node is unique. We can specify a path from the root of the tree to a leaf that uniquely identifies each species in the animal kingdom; for example, $Animalia \rightarrow Chordate \rightarrow Mammal \rightarrow Carnivora \rightarrow Felidae \rightarrow Felis \rightarrow Domestica.$

Another example of a tree structure that you probably use every day is a file system. In a file system, directories, or folders, are structured as a tree:



A small part of the unix file system hierarchy

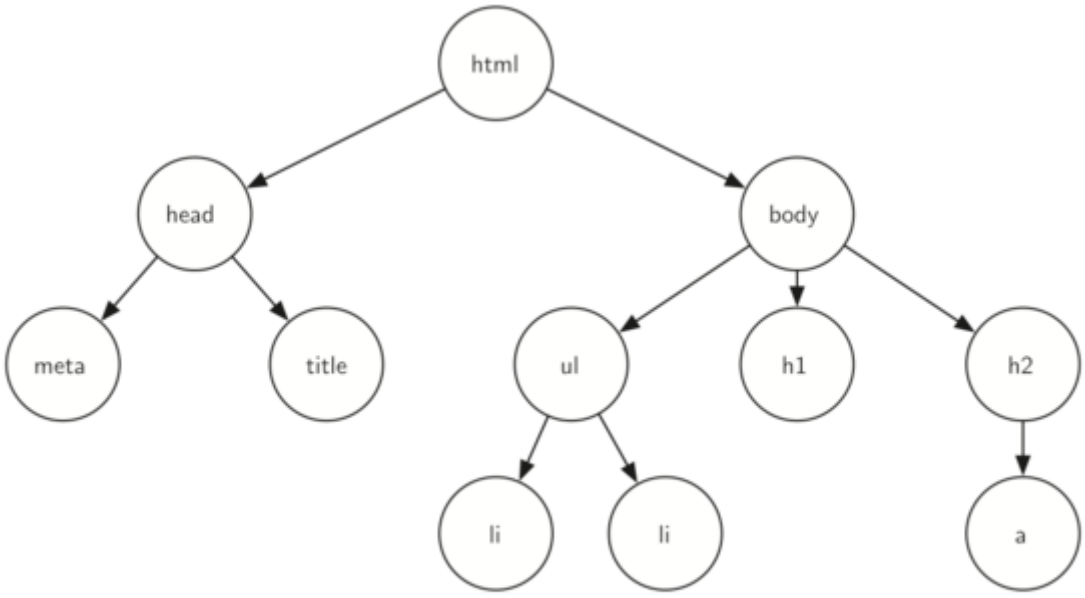
The file system tree has much in common with the biological classification tree. You can follow a path from the root to any directory. That path will uniquely identify that subdirectory (and all the files in it). Another important property of trees, derived from their hierarchical nature, is that you can move entire sections of a tree (called a subtree) to a different position in the tree without affecting the lower levels of the hierarchy. For example, we could take the entire subtree starting with /etc/, detach etc/ from the root and reattach it under usr/. This would change the unique pathname to httpd from /etc/httpd

to /usr/etc/httpd, but would not affect the contents or any children of the httpd directory.

A final example of a tree is a web page. The following is an example of a simple web page written using HTML.

```
<html>
<head>
    <title>simple</title>
</head>
<body>
    <h1>A simple web page</h1>
    <ul>
        <li>List item one</li>
        <li>List item two</li>
    </ul>
    <h2><a href="https://www.google.com">Google</a><h2>
</body>
</html>
```

Here is the tree that corresponds to each of the HTML tags used to create the page.



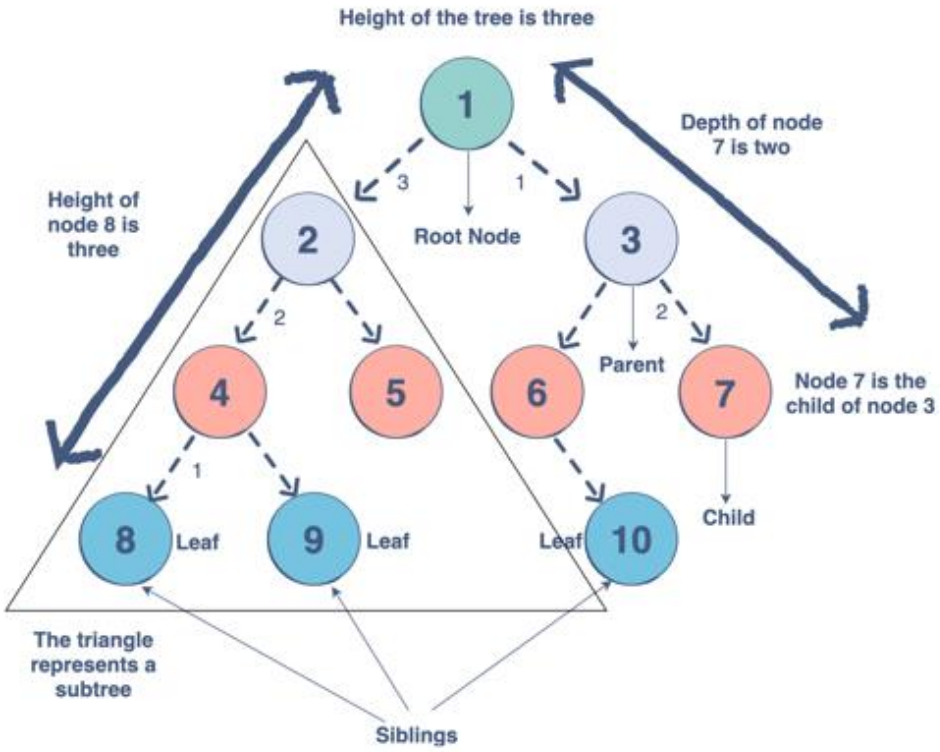
A tree corresponding to the markup elements of a web page

The HTML source code and the tree accompanying the source illustrate another hierarchy. Notice that each level of the tree corresponds to a level of nesting inside the HTML tags. The first tag in the source is <html> and the last

is </html> All the rest of the tags in the page are inside the pair. If you check, you will see that this nesting property is true at all levels of the tree.

Tree Terminologies

Now that we have looked at examples of trees, we will formally define a tree and its components.

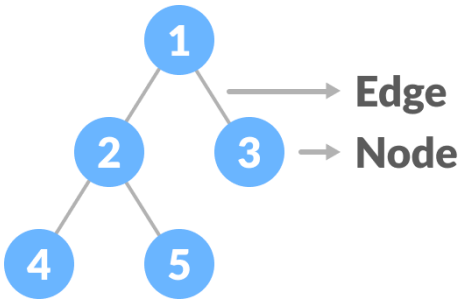


Node

A node is a fundamental part of a tree. It can have a unique name, which we sometimes call the “key.” A node may also have additional information, which we refer to in this book as the “payload.” While the payload information is not central to many tree algorithms, it is often critical in applications that make use of trees.

Edge

It is the link between any two nodes. An edge is another fundamental part of a tree. An edge connects two nodes to show that there is a relationship between them. Every node other than the root is connected by exactly one incoming edge from another node. Each node may have several outgoing edges.



Root

It is the topmost node of a tree. The root of the tree is the only node in the tree that has no incoming edges. In a file system, / is the root of the tree. In an HTML document, the <html> tag is the root of the tree.

Path

A path is an ordered list of nodes that are connected by edges. For example, $Mammal \rightarrow Carnivora \rightarrow Felidae \rightarrow Felis \rightarrow Domestica$ is a path.

Children

The set of nodes cc that have incoming edges from the same node are said to be the children of that node. In our file system example, nodes log/, spool/, and yp/ are the children of node var/.

Parent

A node is the parent of all the nodes to which it connects with outgoing edges. In our file system example, the node var/ is the parent of nodes log/, spool/, and yp/.

Sibling

Nodes in the tree that are children of the same parent are said to be siblings. The nodes etc/ and usr/ are siblings in the file system tree.

Subtree

A subtree is a set of nodes and edges comprised of a parent and all the descendants of that parent.

Leaf Node

A leaf node is a node that has no children. For example, Human and Chimpanzee are leaf nodes in our animal taxonomy example.

Level

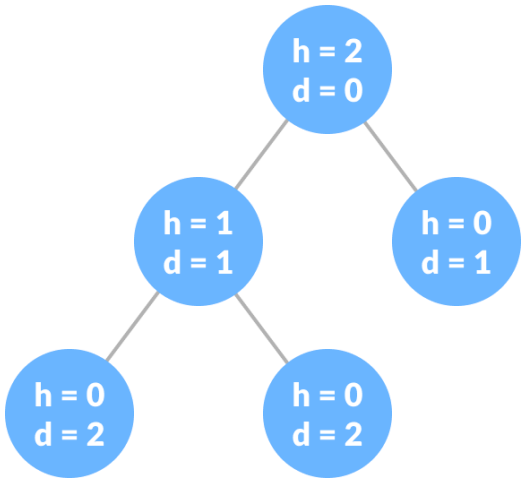
The level of a node '*n*' is the number of edges on the path from the root node to '*n*'. For example, the level of the Felis node in our animal taxonomy example is five. By definition, the level of the root node is zero.

Depth

The depth of a node is the number of edges from the root to the node.

Height

The height of a node is the number of edges from the node to the deepest leaf (i.e., the longest path from the node to a leaf node). The height of the tree in our file system example is two.



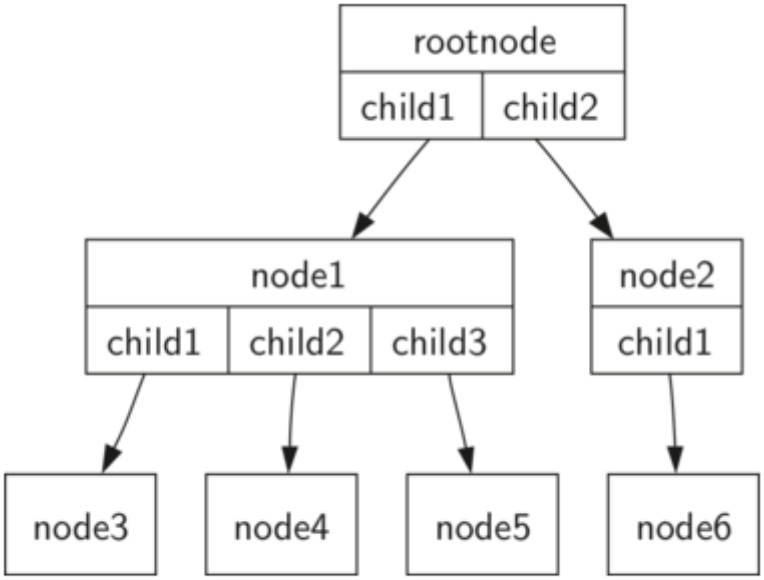
Formal definition of trees

With the basic vocabulary now defined, we can move on to two formal definitions of a tree: one involving nodes and edges, and the other a recursive definition.

Definition one: A tree consists of a set of nodes and a set of edges that connect pairs of nodes. A tree has the following properties:

- One node of the tree is designated as the root node.
- Every node '*n*', except the root node, is connected by an edge from exactly one other node p*p*, where p*p* is the parent of '*n*'.
- A unique path traverses from the root to each node.
- If each node in the tree has a maximum of two children, we say that the tree is a **binary tree**.

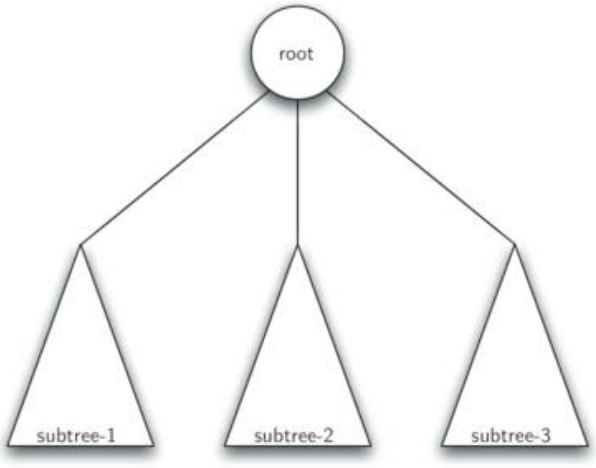
The diagram below illustrates a tree that fits definition one. The arrowheads on the edges indicate the direction of the connection.



A Tree consisting of a set of nodes and edges

Definition two: A tree is either empty or consists of a root and zero or more subtrees, each of which is also a tree. The root of each subtree is connected to the root of the parent tree by an edge.

The diagram below illustrates this recursive definition of a tree. Using the recursive definition of a tree, we know that the tree below has at least four nodes, since each of the triangles representing a subtree must have a root. It may have many more nodes than that, but we do not know unless we look deeper into the tree.



A recursive definition of a tree

Applications of trees

Trees can be applied to many things. The hierarchical structure gives a tree unique property for storing, manipulating, and accessing data. Trees form some of the most basic organization of computers. We can use a tree for the following:

- **Storage as hierarchy.** Storing information that naturally occurs in a hierarchy. File systems on a computer and PDF use tree structures.
- **Searching.** Storing information that we want to search quickly. Trees are easier to search than a Linked List. Some types of trees (like AVL and Red-Black trees) are designed for fast searching.
- **Inheritance.** Trees are used for inheritance, XML parser, machine learning, and DNS, amongst many other things.
- **Indexing.** Advanced types of trees, like B-Trees and B+ Trees, can be used for indexing a database.
- **Networking.** Trees are ideal for things like social networking and computer chess games.
- **Shortest path.** A Spanning Tree can be used to find the shortest paths in routers for networking.

How to make a tree

But, how does that all look in code? To build a tree in Java, for example, we start with the root node.

$$Node < String > \ root \ = \ new \ Node <> (\text{"}root\text{"});$$

Once we have our root, we can add our first child node using $addChild$, which adds a child node and assigns it to a parent node. We refer to this process as **insertion** (adding nodes) and **deletion** (removing nodes).

$$Node < String > \ node1 \ = \ root.addChild(new \ Node < String > (\text{"}node \ 1\text{"}));$$

We continue adding nodes using that same process until we have a complex hierarchical structure. In the next section, let's look at the different kinds of trees we can use.

Explanation

Representing a tree

For the implementation, we'll use an auxiliary *Node* class that will store *int* values, and keep a reference to each child:

```
class Node {
  int value;
  Node left;
  Node right;

  Node(int value) {
    this.value = value;
    right = null;
    left = null;
  }
}
```

Then we'll add the starting node of our tree, usually called the *root:*

```
public class BinaryTree {
  Node root;
  // ...
}
```

Common operations

Now let's see the most common operations we can perform on a binary tree.

Inserting Elements

The first operation we're going to cover is the insertion of new nodes.

First, we have to find the place where we want to add a new node in order to keep the tree sorted. We'll follow these rules starting from the root node:

- if the new node's value is lower than the current node's, we go to the left child
- if the new node's value is greater than the current node's, we go to the right child
- when the current node is *null,* we've reached a leaf node and we can insert the new node in that position

Then we'll create a recursive method to do the insertion:

```
private Node addRecursive(Node current, int value) {
    if (current == null) {
        return new Node(value);
    }
    if (value < current.value) {
        current.left = addRecursive(current.left, value);
    } else if (value > current.value) {
        current.right = addRecursive(current.right, value);
    } else {
        // value already exists
        return current;
    }
    return current;
}
```

Next, we'll create the public method that starts the recursion from the root node:

```
public void add(int value) {
    root = addRecursive(root, value);
}
```

Let's see how we can use this method to create the tree from our example:

```
private BinaryTree createBinaryTree() {
    BinaryTree bt = new BinaryTree();
    bt.add(6);
    bt.add(4);
    bt.add(8);
    bt.add(3);
```

```
    bt.add(5);
    bt.add(7);
    bt.add(9);
    return bt;
}
```

Finding an Element

Now let's add a method to check if the tree contains a specific value.

As before, we'll first create a recursive method that traverses the tree:

```
private boolean containsNodeRecursive(Node current, int value) {
    if (current == null) {
        return false;
    }
    if (value == current.value) {
        return true;
    }
    return value < current.value
      ? containsNodeRecursive(current.left, value)
      : containsNodeRecursive(current.right, value);
}
```

Here we're searching for the value by comparing it to the value in the current node; we'll then continue in the left or right child depending on the outcome.

Next, we'll create the public method that starts from the root:

```
public boolean containsNode(int value) {
    return containsNodeRecursive(root, value);
}
```

All the nodes added should be contained in the tree.

Deleting an Element

Another common operation is the deletion of a node from the tree.

First, we must find the node to delete in a similar way as before:

```
private Node deleteRecursive(Node current, int value) {
  if (current == null) {
    return null;
  }

  if (value == current.value) {
    // Node to delete found
    // ...code to delete the node will go here
  }
  if (value < current.value) {
    current.left = deleteRecursive(current.left, value);
    return current;
  }
  current.right = deleteRecursive(current.right, value);
  return current;
}
```

Once we find the node to delete, there are 3 main different cases:

- **a node has no children –** this is the simplest case; we just need to replace this node with *null* in its parent node
- **a node has exactly one child –** in the parent node, we replace this node with its only child.
- **a node has two children** – this is the most complex case because it requires a tree reorganization

Let's see how we would implement the first case when the node is a leaf node:

```
if (current.left == null && current.right == null) {
  return null;
```

```
}
```

Now let's continue with the case when the node has one child:

```
if (current.right == null) {
  return current.left;
}


if (current.left == null) {
  return current.right;
}
```

Here we're returning the non-null child so it can be assigned to the parent node.

Finally, we have to handle the case where the node has two children.

First, we need to find the node that will replace the deleted node. We'll use the smallest node of the soon to be deleted node's right sub-tree:

```
private int findSmallestValue(Node root) {
  return root.left == null ? root.value
          : findSmallestValue(root.left);
}
```

Then we assign the smallest value to the node to delete, and after that, we'll delete it from the right sub-tree:

```
int smallestValue = findSmallestValue(current.right);
current.value = smallestValue;
current.right = deleteRecursive(current.right, smallestValue);
return current;
```

Finally, we'll create the public method that starts the deletion from the root:

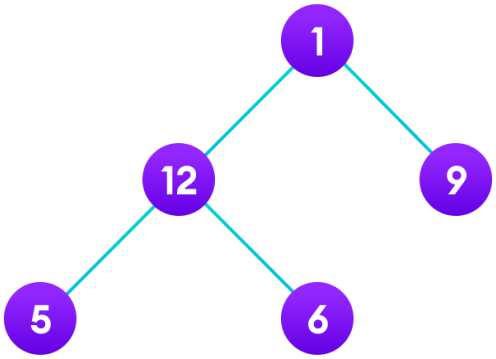
```
public void delete(int value) {
  root = deleteRecursive(root, value);
}
```

Tree traversal

In order to perform any operation on a tree, you need to reach to the specific node. The tree traversal algorithm helps in visiting a required node in the tree.

Traversing a tree means visiting every node in the tree. You might, for instance, want to add all the values in the tree or find the largest one. For all these operations, you will need to visit each node of the tree.

Linear data structures like arrays, stacks, queues, and linked list have only one way to read the data. But a hierarchical data structure like a tree can be traversed in different ways.



Let's think about how we can read the elements of the tree in the image shown above.

Starting from top, Left to right

$$1 -> 12 -> 5 -> 6 -> 9$$

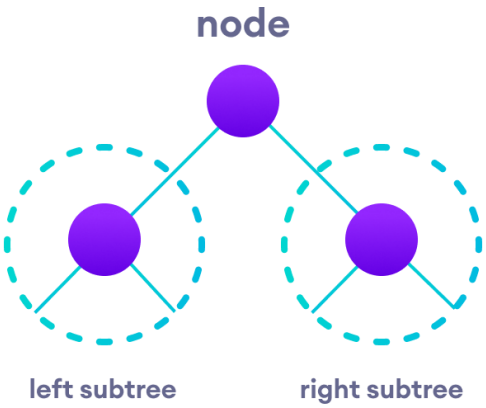
Starting from bottom, Left to right

$$5 -> 6 -> 12 -> 9 -> 1$$

Although this process is somewhat easy, it doesn't respect the hierarchy of the tree, only the depth of the nodes. The difference between these patterns is the order in which each node is visited.

Every tree is a combination of

- A node carrying data
- Two subtrees



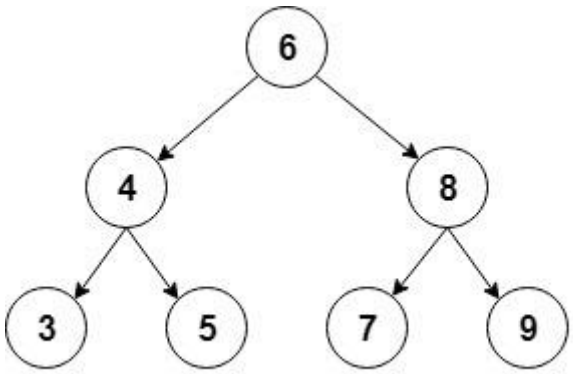
Depth-First Search

Depth-first search is a type of traversal that goes deep as much as possible in every child before exploring the next sibling. We follow a path from the starting node to the ending node and then start another path until all nodes are visited. This is commonly implemented using stacks, and it requires less memory than BFS. It is best for topographical sorting, such as graph backtracking or cycle detection.

The steps for the DFS algorithm are as follows:

1. Pick a node. Push all adjacent nodes into a stack.
2. Pop a node from that stack and push adjacent nodes into another stack.
3. Repeat until the stack is empty or you have reached your goal. As you visit nodes, you must mark them as visited before proceeding, or you will be stuck in an infinite loop.

Take this tree as an example:



Remember that our goal is to visit each node, so we need to visit all the nodes in the subtree, visit the root node and visit all the nodes in the right subtree as well. We call this visitation of the nodes a "traversal." The three traversals we will look at are called preorder, inorder, and postorder. Let's start out by defining these three traversals more carefully, then look at some examples where these patterns are useful.

- **preorder**: In a preorder traversal, we visit the root node first, then recursively do a preorder traversal of the left subtree, followed by a recursive preorder traversal of the right subtree.

  $public\ void\ traversePreOrder(Node\ node)\ \{$
  $\quad if\ (node\ !=\ null)\ \{$
  $\quad\quad System.out.print(" "\ +\ node.value);$
  $\quad\quad traversePreOrder(node.left);$
  $\quad\quad traversePreOrder(node.right);$
  $\quad \}$
  $\}$

  Let's check the pre-order traversal in the console output:

  6 4 3 5 8 7 9
- **inorder**: In an inorder traversal, we recursively do an inorder traversal on the left subtree, visit the root node, and finally do a recursive inorder traversal of the right subtree.

  $public\ void\ traverseInOrder(Node\ node)\ \{$
  $\quad if\ (node\ !=\ null)\ \{$
  $\quad\quad traverseInOrder(node.left);$
  $\quad\quad System.out.print(" "\ +\ node.value);$
  $\quad\quad traverseInOrder(node.right);$
  $\quad \}$
  $\}$

  If we call this method, the console output will show the in-order traversal:

  3 4 5 6 7 8 9
- **postorder**: In a postorder traversal, we recursively do a postorder traversal of the left subtree and the right subtree followed by a visit to the root node.

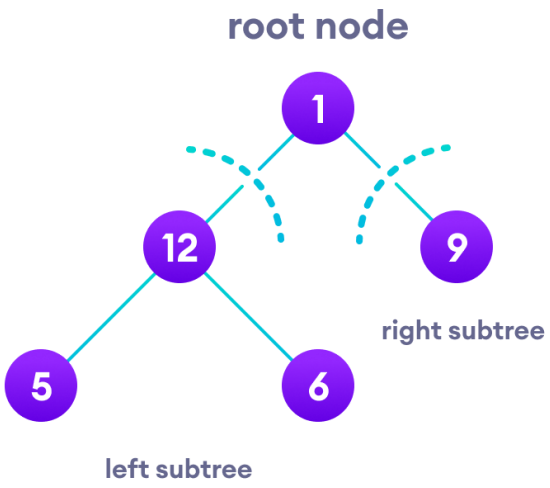
  $public\ void\ traversePostOrder(Node\ node)\ \{$
  $\quad if\ (node\ !=\ null)\ \{$
  $\quad\quad traversePostOrder(node.left);$
  $\quad\quad traversePostOrder(node.right);$

```
        System.out.print(" " + node.value);
    }
}
```

Here are the nodes in post-order:

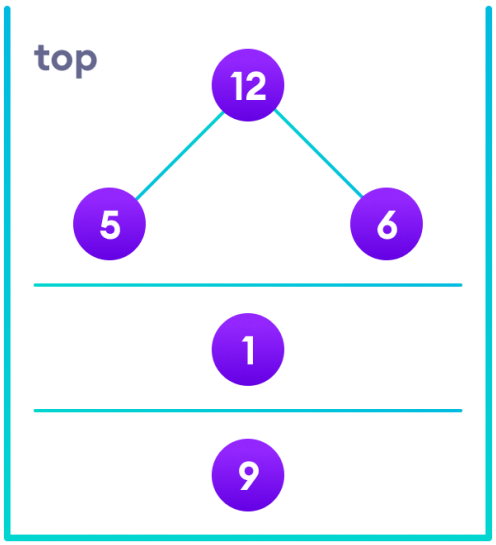
3 5 4 7 9 8 6

Let's visualize in-order traversal. We start from the root node.



We traverse the left subtree first. We also need to remember to visit the root node and the right subtree when this tree is done.

Let's put all this in a stack so that we remember.

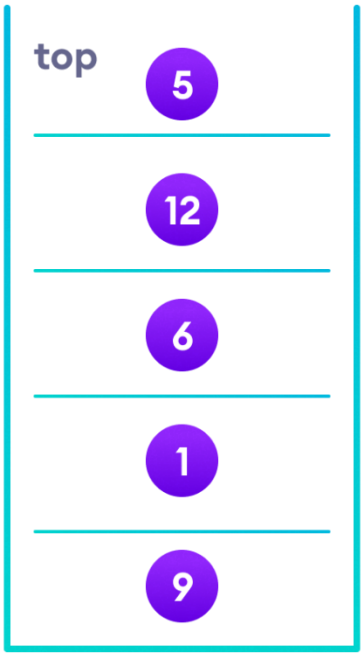


Now we traverse to the subtree pointed on the TOP of the stack.

Again, we follow the same rule of inorder:

$$Left\ subtree -> \ root -> \ right\ subtree$$

After traversing the left subtree, we are left with:



Since the node "5" doesn't have any subtrees, we print it directly. After that we print its parent "12" and then the right child "6".

Putting everything on a stack was helpful because now that the left-subtree of the root node has been traversed, we can print it and go to the right subtree.

After going through all the elements, we get the inorder traversal as:

$$5 -> \ 12 -> \ 6 -> \ 1 -> \ 9$$

We don't have to create the stack ourselves because recursion maintains the correct order for us.

Implementation:

```
class Node {
  int item;
  Node left,right;
```

```
  public Node(int key) {
  item = key;
  left = right = null;
  }
}

class BinaryTree {
  // Root of Binary Tree
  Node root;

  BinaryTree() {
  root = null;
  }

  void postorder(Node node) {
  if (node == null)
   return;

  // Traverse left
  postorder(node.left);
  // Traverse right
  postorder(node.right);
  // Traverse root
  System.out.print(node.item + "-> ");
  }
```

```
void inorder(Node node) {
if (node == null)
 return;

// Traverse left
inorder(node.left);
// Traverse root
System.out.print(node.item + "-> ");
// Traverse right
inorder(node.right);
}

void preorder(Node node) {
if (node == null)
 return;

// Traverse root
System.out.print(node.item + "-> ");
// Traverse left
preorder(node.left);
// Traverse right
preorder(node.right);
}

public static void main(String[] args) {
BinaryTree tree = new BinaryTree();
tree.root = new Node(1);
```

```
tree.root.left = new Node(12);
tree.root.right = new Node(9);
tree.root.left.left = new Node(5);
tree.root.left.right = new Node(6);

System.out.println("Inorder traversal");
tree.inorder(tree.root);

System.out.println("\nPreorder traversal ");
tree.preorder(tree.root);

System.out.println("\nPostorder traversal");
tree.postorder(tree.root);
}
}
```

Breadth-First Search

This is another common type of traversal that visits all the nodes of a level before going to the next level.

This kind of traversal is also called level-order and visits all the levels of the tree starting from the root, and from left to right.

For the implementation, we'll use a Queue to hold the nodes from each level in order. We'll extract each node from the list, print its values, then add its children to the queue:

```
public void traverseLevelOrder() {
  if (root == null) {
    return;
  }

  Queue < Node > nodes = new LinkedList <> ();
  nodes.add(root);
```

```
  while (!nodes.isEmpty()) {

    Node node = nodes.remove();

    System.out.print(" " + node.value);

    if (node.left != null) {
      nodes.add(node.left);
    }

    if (node.right != null) {
      nodes.add(node.right);
    }
  }
}
```

In this case, the order of the nodes will be:
6 4 8 3 5 7 9

Binary Search Tree (BST)

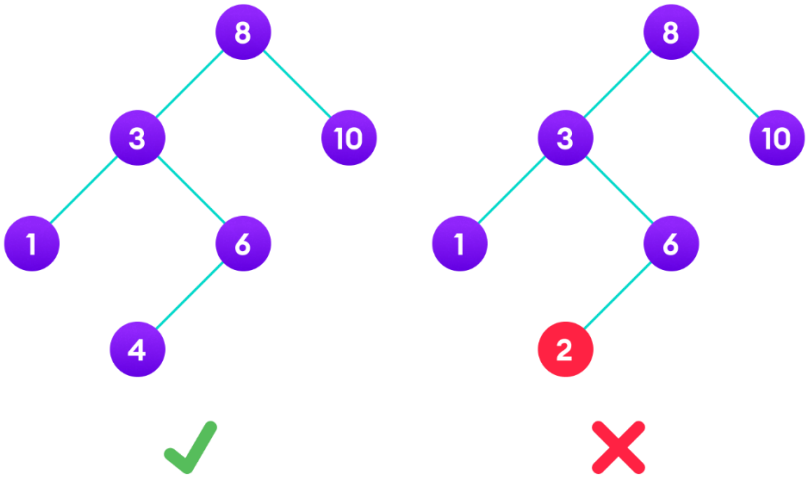
Explanation:

Binary search tree is a data structure that quickly allows us to maintain a sorted list of numbers.

- It is called a binary tree because each tree node has a maximum of two children.
- It is called a search tree because it can be used to search for the presence of a number in O(log(n)) time.

The properties that separate a binary search tree from a regular binary tree is

1. All nodes of left subtree are less than the root node
2. All nodes of right subtree are more than the root node
3. Both subtrees of each node are also BSTs i.e., they have the above two properties



The binary tree on the right isn't a binary search tree because the right subtree of the node "3" contains a value smaller than it.

BST Applications:

1. In multilevel indexing in the database
2. For dynamic sorting
3. For managing virtual memory areas in Unix kernel

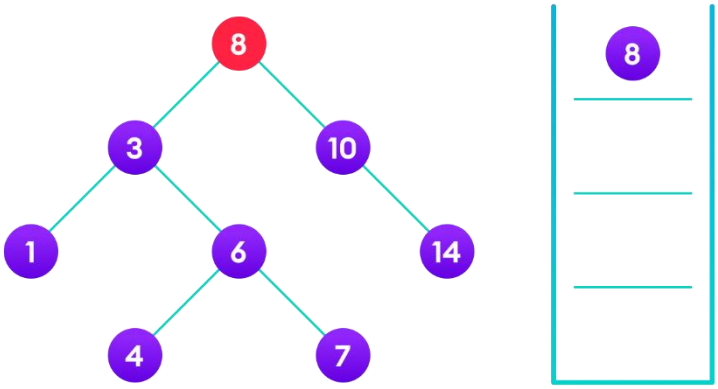
Operations in a BST:

There are two basic operations that you can perform on a binary search tree:

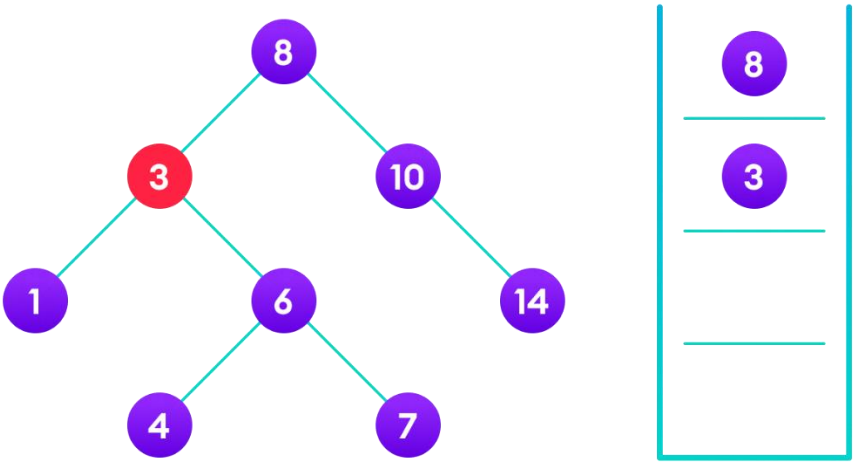
- **Search Operation:** The algorithm depends on the property of BST that if each left subtree has values below root and each right subtree has values above the root.
  If the value is below the root, we can say for sure that the value is not in the right subtree; we need to only search in the left subtree and if the value is above the root, we can say for sure that the value is not in the left subtree; we need to only search in the right subtree.
  Algorithm:

$$
\begin{aligned}
&If\ root\ ==\ NULL\\
&\quad return\ NULL;\\
&If\ number\ ==\ root{->}\ data\\
&\quad return\ root{->}\ data;\\
&If\ number\ <\ root{->}\ data\\
&\quad return\ search(root{->}\ left)\\
&If\ number\ >\ root{->}\ data\\
&\quad return\ search(root{->}\ right)
\end{aligned}
$$

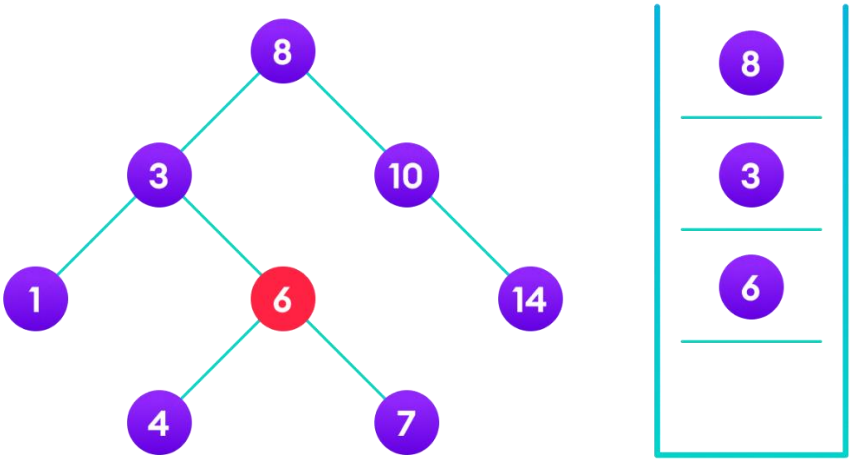
Let us try to visualize this with a diagram.



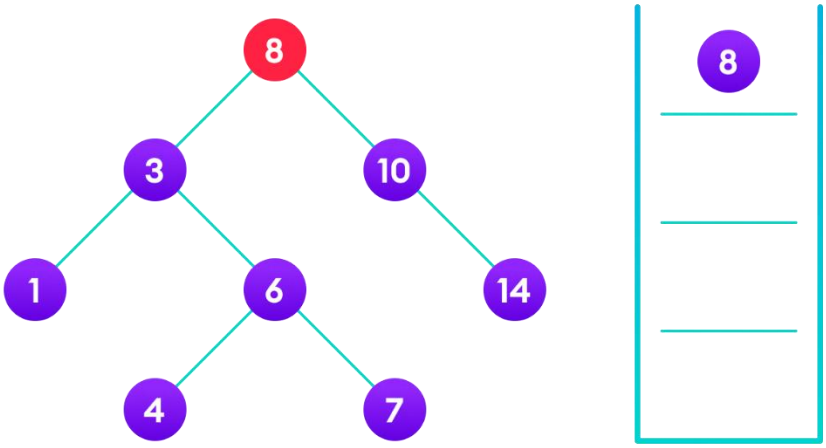
Element to be searched = 4. 4 is not found so, traverse through the left subtree of 8



4 is not found so, traverse through the right subtree of 3



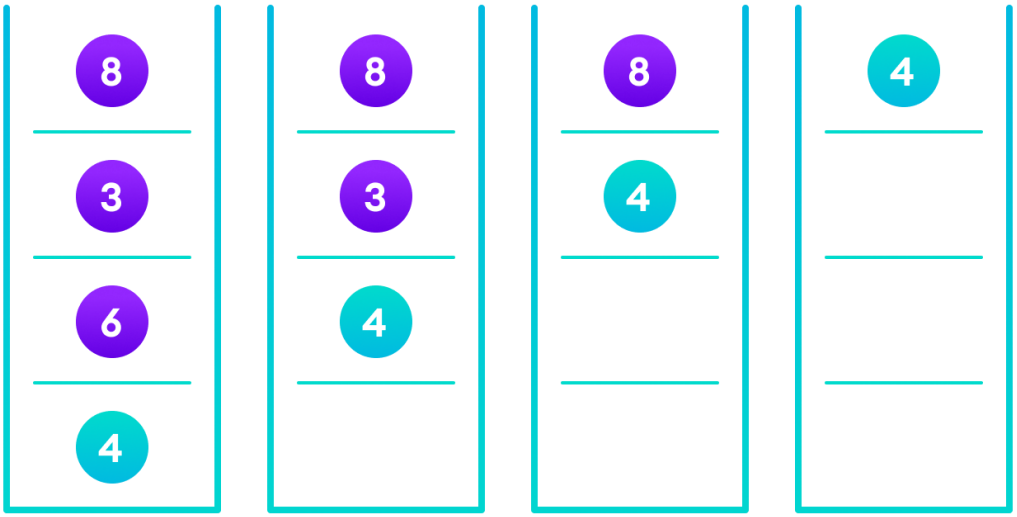
4 is not found so, traverse through the left subtree of 6



4 is found

If the value is found, we return the value so that it gets propagated in each recursion step as shown in the image below.

When we return either the new node or NULL, the value gets returned again and again until $search(root)$ returns the final result.



If the value is found in any of the subtrees, it is propagated up so that in the end it is returned, otherwise null is returned

If the value is not found, we eventually reach the left or right child of a leaf node which is NULL and it gets propagated and returned.

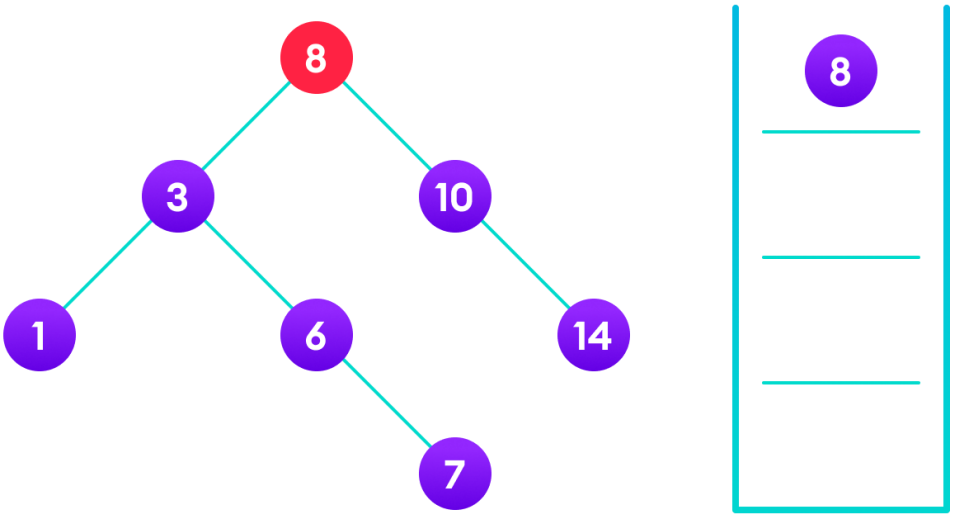
- **Insert Operation:** Inserting a value in the correct position is similar to searching because we try to maintain the rule that the left subtree is lesser than root and the right subtree is larger than root.

  We keep going to either right subtree or left subtree depending on the value and when we reach a point left or right subtree is null, we put the new node there.

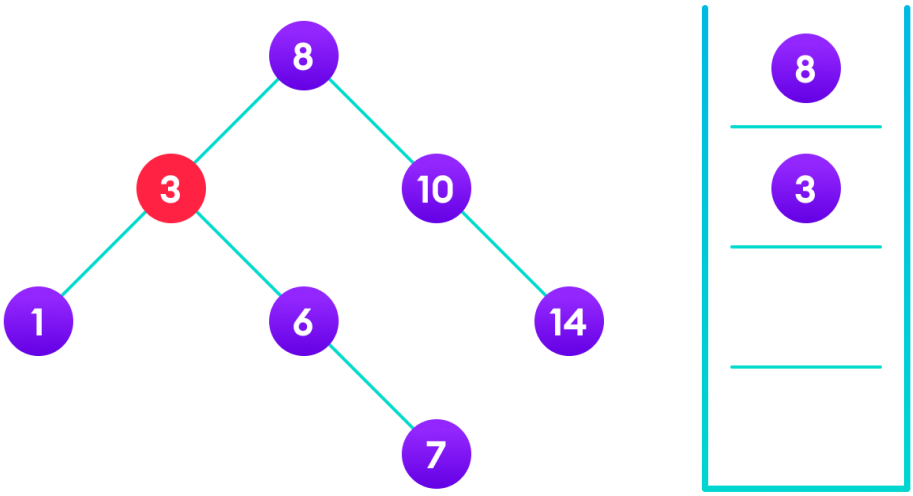
  Algorithm:

$$If\ node\ ==\ NULL$$
$$\quad return\ createNode(data)$$
$$if\ (data\ <\ node{->}\ data)$$
$$\quad node{->}\ left\ =\ insert(node{->}\ left, data);$$
$$else\ if\ (data\ >\ node{->}\ data)$$
$$\quad node{->}\ right\ =\ insert(node{->}\ right, data);$$
$$return\ node;$$

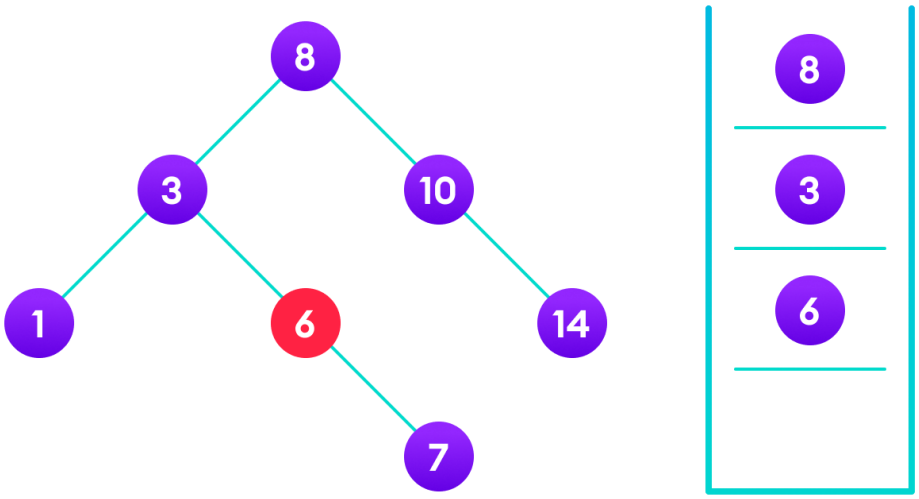
  The algorithm isn't as simple as it looks. Let's try to visualize how we add a number to an existing BST.



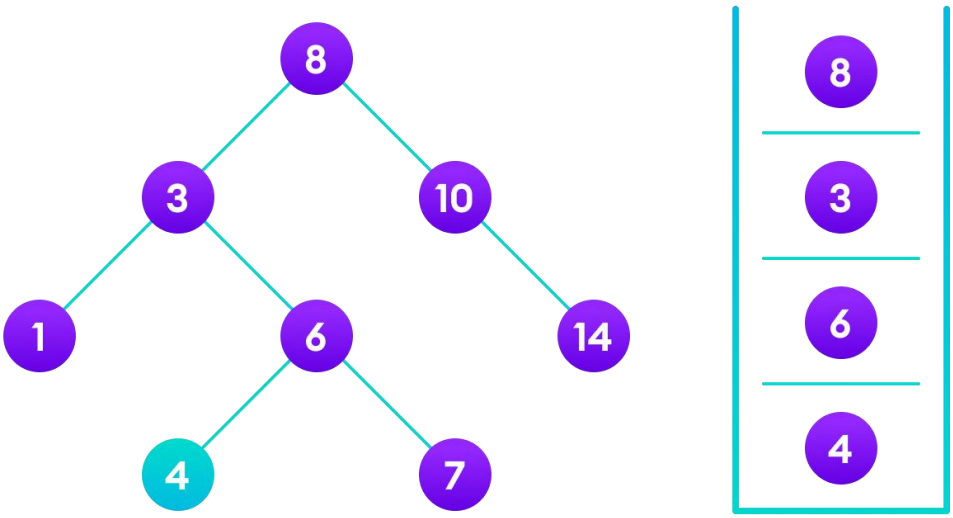
4<8 so, transverse through the left child of 8



4>3 so, transverse through the right child of 8



4<6 so, transverse through the left child of 6



Insert 4 as a left child of 6

We have attached the node, but we still have to exit from the function without doing any damage to the rest of the tree. This is where the return node; at the end comes in handy. In the case of NULL, the newly created node is returned and attached to the parent node, otherwise the same node is returned without any change as we go up until we return to the root.

This makes sure that as we move back up the tree, the other node connections aren't changed.

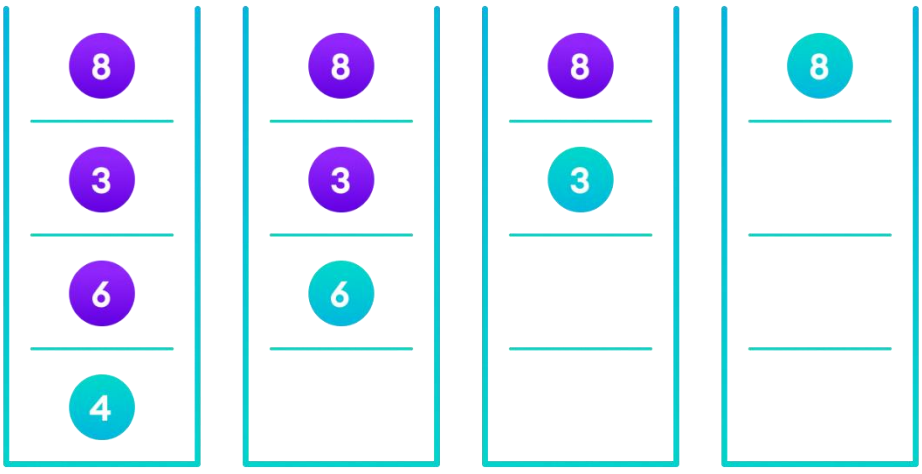
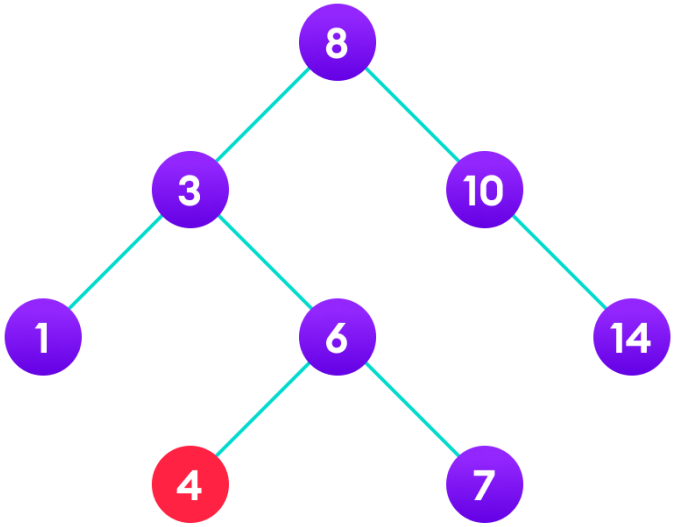
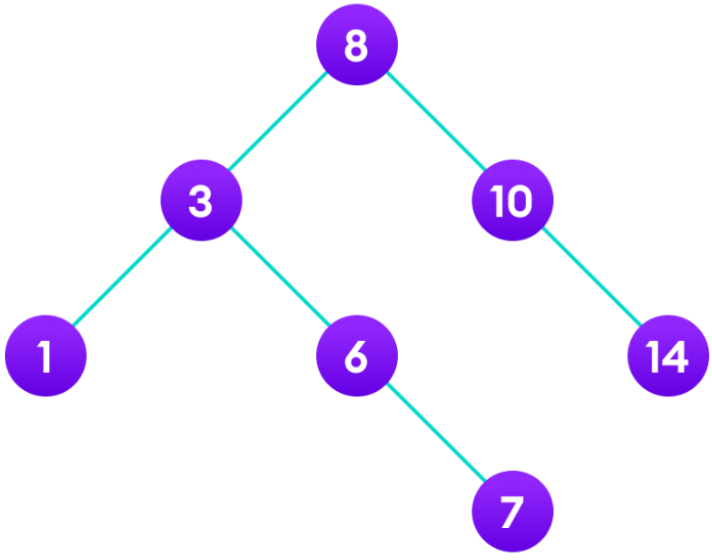


Image showing the importance of returning the root element at the end so that the elements don't lose their position during the upward recursion step.

- **Delete Operation:** There are three cases for deleting a node from a binary search tree.
    - *Case I* - In the first case, the node to be deleted is the leaf node. In such a case, simply delete the node from the tree.

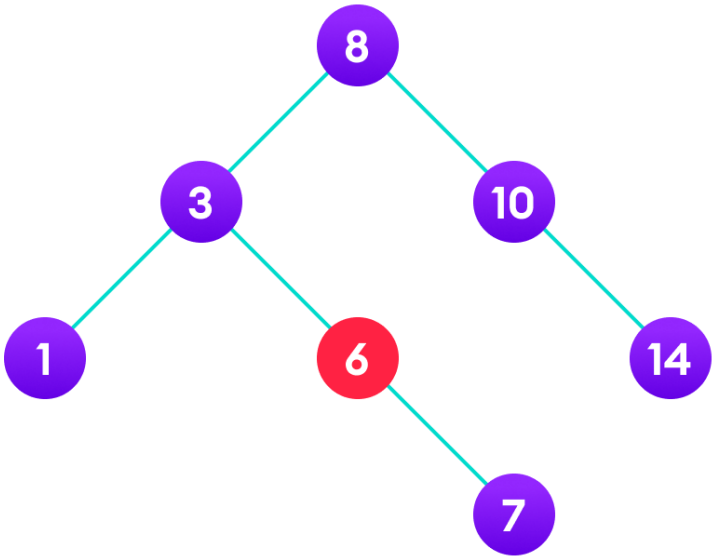


4 is to be deleted

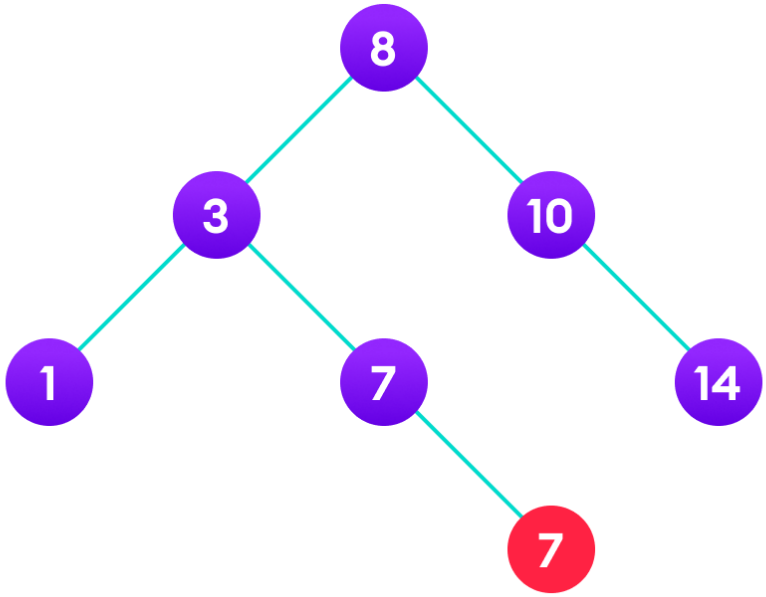


After deletion

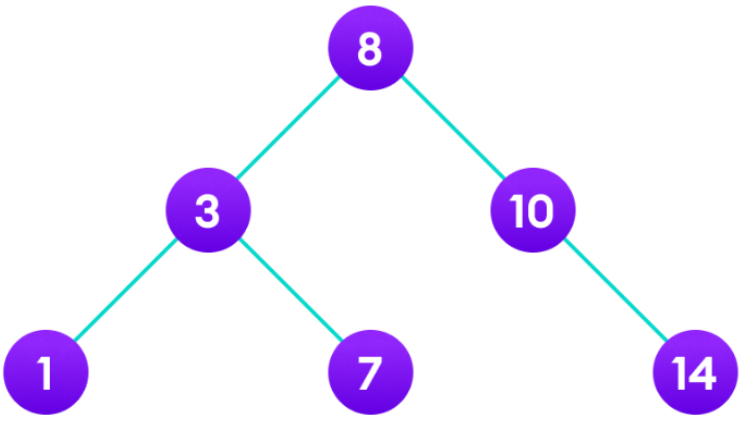
    - *Case II* - In the second case, the node to be deleted lies has a single child node. In such a case follow the steps below:
        - Replace that node with its child node.
        - Remove the child node from its original position.



6 is to be deleted

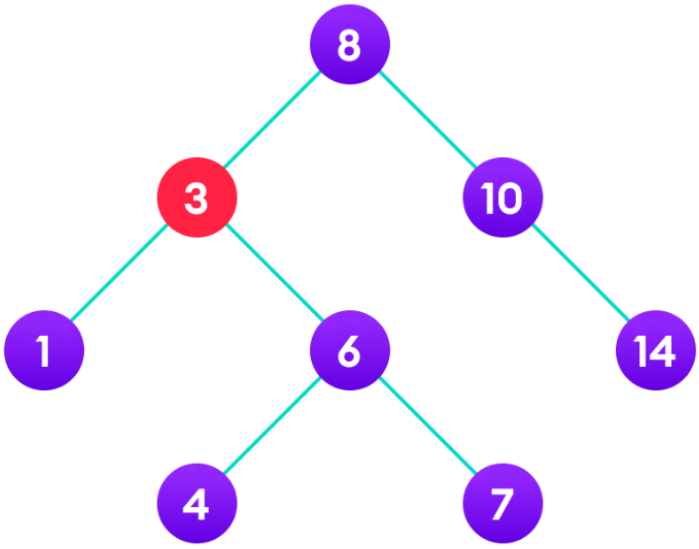


copy the value of its child to the node and delete the child

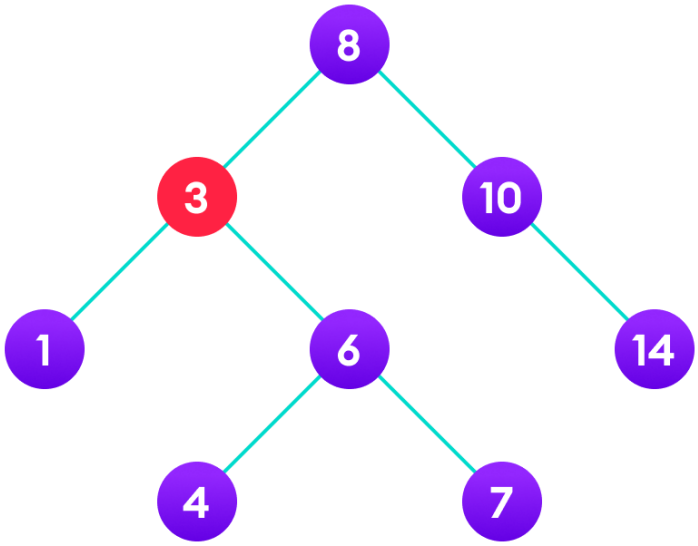


Final tree

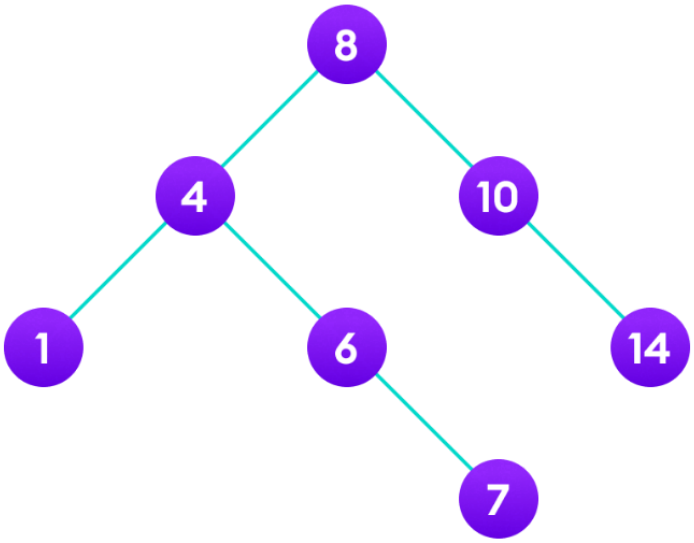
- *Case III* - In the third case, the node to be deleted has two children. In such a case follow the steps below:
  - Get the inorder successor of that node.
  - Replace the node with the inorder successor.
  - Remove the inorder successor from its original position.



3 is to be deleted



Copy the value of the inorder successor (4) to the node



Deletion the inorder successor

Code:

```
class BinarySearchTree {
 class Node {
  int key;
  Node left, right;

  public Node(int item) {
   key = item;
   left = right = null;
  }
 }

 Node root;

 BinarySearchTree() {
  root = null;
 }

 void insert(int key) {
  root = insertKey(root, key);
 }

 // Insert key in the tree
 Node insertKey(Node root, int key) {
  // Return a new node if the tree is empty
```

```
 if (root == null) {
  root = new Node(key);
  return root;
 }

 // Traverse to the right place and insert the node
 if (key < root.key)
  root.left = insertKey(root.left, key);
 else if (key > root.key)
  root.right = insertKey(root.right, key);

 return root;
}

void inorder() {
 inorderRec(root);
}

// Inorder Traversal
void inorderRec(Node root) {
 if (root != null) {
  inorderRec(root.left);
  System.out.print(root.key + " -> ");
  inorderRec(root.right);
 }
}

void deleteKey(int key) {
 root = deleteRec(root, key);
}

Node deleteRec(Node root, int key) {
 // Return if the tree is empty
 if (root == null)
  return root;

 // Find the node to be deleted
 if (key < root.key)
  root.left = deleteRec(root.left, key);
```

```
  else if (key > root.key)
   root.right = deleteRec(root.right, key);
  else {
   // If the node is with only one child or no child
   if (root.left == null)
    return root.right;
   else if (root.right == null)
    return root.left;

   // If the node has two children
   // Place the inorder successor in position of the node to be deleted
   root.key = minValue(root.right);

   // Delete the inorder successor
   root.right = deleteRec(root.right, root.key);
  }

  return root;
 }

 // Find the inorder successor
 int minValue(Node root) {
  int minv = root.key;
  while (root.left != null) {
   minv = root.left.key;
   root = root.left;
  }
  return minv;
 }

 // Driver Program to test above functions
 public static void main(String[] args) {
  BinarySearchTree tree = new BinarySearchTree();

  tree.insert(8);
  tree.insert(3);
  tree.insert(1);
  tree.insert(6);
  tree.insert(7);
```

```
    tree.insert(10);
    tree.insert(14);
    tree.insert(4);

    System.out.print("Inorder traversal: ");
    tree.inorder();

    System.out.println("\n\nAfter deleting 10");
    tree.deleteKey(10);
    System.out.print("Inorder traversal: ");
    tree.inorder();
  }
}
```

Binary Search Tree Complexities:

Operation	Best Case Complexity	Average Case Complexity	Worst Case Complexity
Search	$O(log\ n)$	$O(log\ n)$	$O(n)$
Insertion	$O(log\ n)$	$O(log\ n)$	$O(n)$
Deletion	$O(log\ n)$	$O(log\ n)$	$O(n)$

Problem Statements

Construct Tree from given Inorder and Preorder traversals

Explanation:

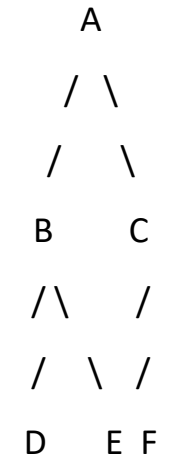
Let us consider the below traversals:

Inorder sequence: $D\ B\ E\ A\ F\ C$

Preorder sequence: $A\ B\ D\ E\ C\ F$

In a Preorder sequence, the leftmost element is the root of the tree. So, we know ‘A’ is the root for given sequences. By searching ‘A’ in the Inorder sequence, we can find out all elements on the left side of ‘A’ is in the left subtree, and elements on right in the right subtree. So, we know the below structure now.

We recursively follow the above steps and get the following tree.

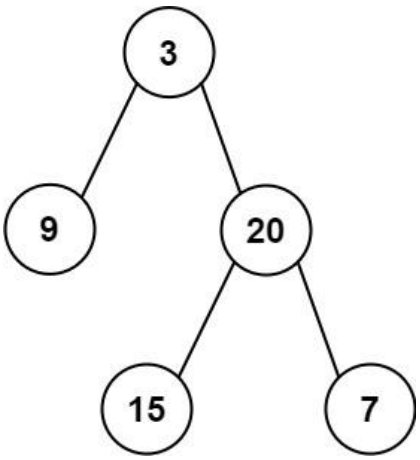


Problem Statement:

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

Examples:

Example-1:



Input: preorder = $[3,9,20,15,7]$, inorder = $[9,3,15,20,7]$

Output: $[3,9,20,null,null,15,7]$

Example 2:

Input: preorder = $[-1]$, inorder = $[-1]$

Output: $[-1]$

AVL Tree

Explanation:

AVL tree is a self-balancing binary search tree in which each node maintains extra information called a balance factor whose value is either -1, 0 or +1.

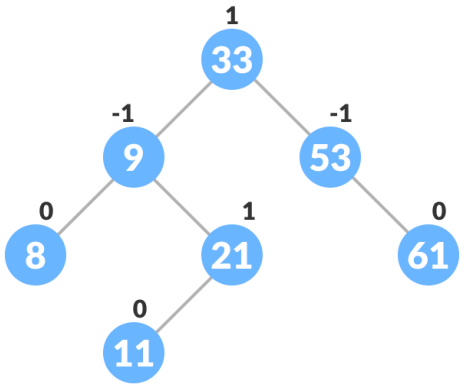
AVL tree got its name after its inventor Georgy Adelson-Velsky and Landis.

Balance Factor: Balance factor of a node in an AVL tree is the difference between the height of the left subtree and that of the right subtree of that node.

$$\begin{aligned} &Balance\ Factor \\ &\quad = (Height\ of\ Left\ Subtree \\ &\quad - Height\ of\ Right\ Subtree)\ or\ (Height\ of\ Right\ Subtree \\ &\quad - Height\ of\ Left\ Subtree) \end{aligned}$$

The self-balancing property of an AVL tree is maintained by the balance factor.

An example of a balanced AVL tree is:



Here are some key points about AVL trees:

- If there are n nodes in AVL tree, minimum height of AVL tree is $floor(log_2 n)$.
- If there are n nodes in AVL tree, maximum height can't exceed $1.44 * log_2 n$.
- If height of AVL tree is h, maximum number of nodes can be $2^{h+1} - 1$.
- Minimum number of nodes in a tree with height h can be represented as:
  $N(h) = N(h-1) + N(h-2) + 1 \; for \; n > 2 \; where \; N(0) = 1 \; and \; N(1) = 2.$
- The complexity of searching, inserting and deletion in AVL tree is $O(log\ n)$.

Problem Statement:

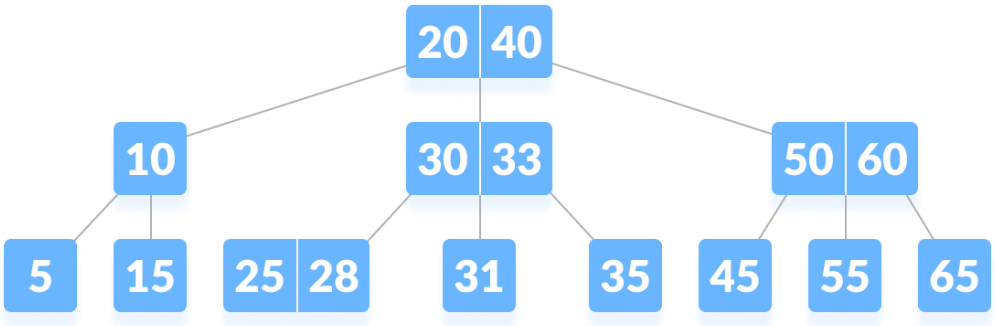
Implement insertion and deletion of node in an AVL tree.

B-tree

Explanation:

B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children. It is a generalized form of the binary search tree.

It is also known as a height-balanced m-way tree.



Why do you need a B-tree data structure?

The need for B-tree arose with the rise in the need for lesser time in accessing the physical storage media like a hard disk. The secondary storage devices are slower with a larger capacity. There was a need for such types of data structures that minimize the disk accesses.

Other data structures such as a binary search tree, avl tree, red-black tree, etc can store only one key in one node. If you have to store a large number of keys, then the height of such trees becomes very large, and the access time increases.

However, B-tree can store many keys in a single node and can have multiple child nodes. This decreases the height significantly allowing faster disk accesses.

B-tree Properties

- For each node x, the keys are stored in increasing order.
- In each node, there is a boolean value $x.leaf$ which is true if x is a leaf.
- If n is the order of the tree, each internal node can contain at most $n - 1$ keys along with a pointer to each child.
- Each node except root can have at most n children and at least $n/2$ children.
- All leaves have the same depth (i.e., height-h of the tree).
- The root has at least 2 children and contains a minimum of 1 key.
- If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2, h \geq logt\ (n+1)/2$.

Problem Statement:

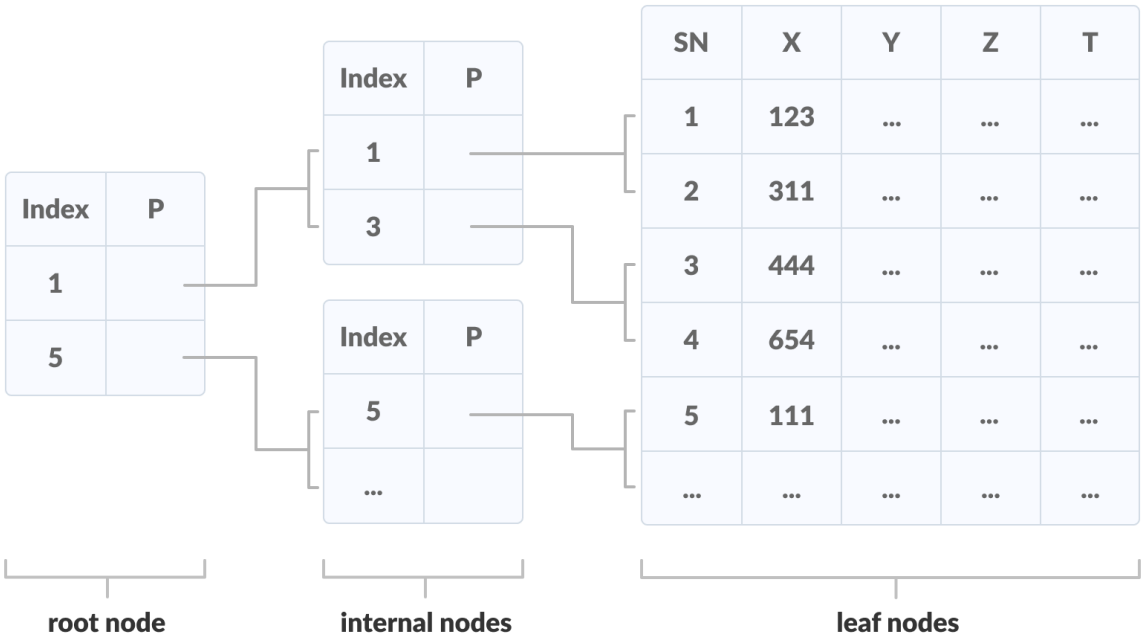
Implement searching an element in a B-tree.

B+ Tree

Explanation:

A B+ tree is an advanced form of a self-balancing tree in which all the values are present in the leaf level.

An important concept to be understood before learning B+ tree is multilevel indexing. In multilevel indexing, the index of indices is created as in figure below. It makes accessing the data easier and faster.



Properties of a B+ Tree

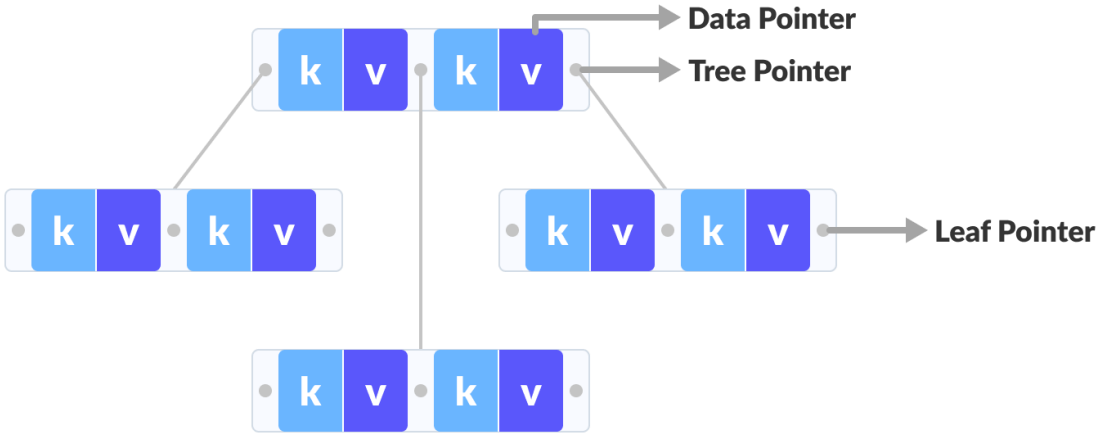
1. All leaves are at the same level.
2. The root has at least two children.
3. Each node except root can have a maximum of m children and at least $m/2$ children.
4. Each node can contain a maximum of $m - 1$ keys and a minimum of $\lceil m/2 \rceil - 1$ keys.

Comparison between a B-tree and a B+ Tree

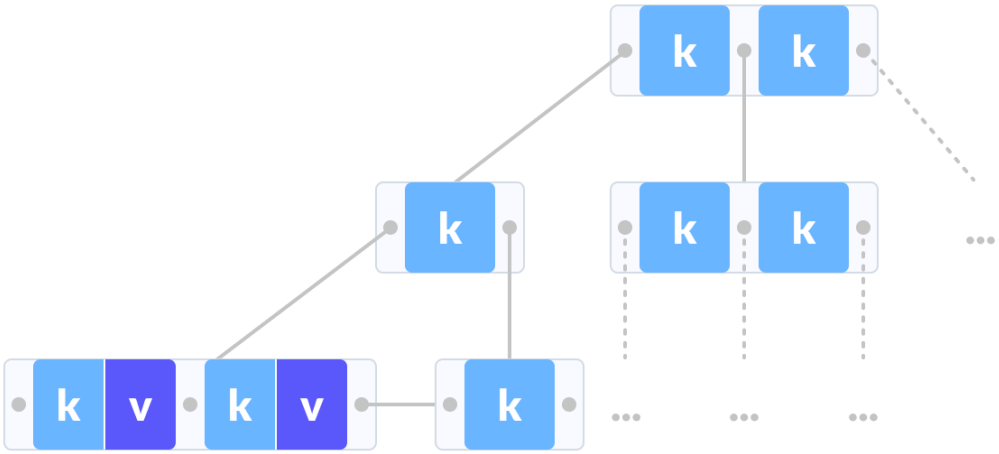
The data pointers are present only at the leaf nodes on a B+ tree whereas the data pointers are present in the internal, leaf or root nodes on a B-tree.

The leaves are not connected with each other on a B-tree whereas they are connected on a B+ tree.

Operations on a B+ tree is faster than on a B-tree.



B-tree



B+ tree

B+ Tree Applications

- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)
- Database indexing

Problem Statement:

Implement searching an element in a B+ tree

Assessment

1) Which of the following is false about a binary search tree?
   a) The left child is always lesser than its parent
   b) The right child is always greater than its parent
   c) The left and right sub-trees should also be binary search trees
   d) In order sequence gives decreasing order of elements

2) What is the speciality about the inorder traversal of a binary search tree?
   a) It traverses in a non-increasing order
   b) It traverses in an increasing order
   c) It traverses in a random fashion
   d) It traverses based on priority of the node

3) What does the following piece of code do?

$$
\begin{aligned}
&public\ void\ func(Tree\ root)\\
&\{\\
&func(root.left());\\
&func(root.right());\\
&System.out.println(root.data());\\
&\}
\end{aligned}
$$

   a) Preorder traversal
   b) Inorder traversal
   c) Postorder traversal
   d) Level order traversal

4) What does the following piece of code do?

$$
\begin{aligned}
&public\ void\ func(Tree\ root)\\
&\{\\
&System.out.println(root.data());\\
&func(root.left());\\
&func(root.right());\\
&\}
\end{aligned}
$$

   a) Preorder traversal
   b) Inorder traversal
   c) Postorder traversal
   d) Level order traversal

5) What are the conditions for an optimal binary search tree and what is its advantage?
a) The tree should not be modified and you should know how often the keys are accessed, it improves the lookup cost
b) You should know the frequency of access of the keys, improves the lookup time
c) The tree can be modified and you should know the number of elements in the tree before hand, it improves the deletion time
d) The tree should be just modified and improves the lookup time

6) The number of edges from the root to the node is called _____ of the tree.
a) Height
b) Depth
c) Length
d) Width

7) The number of edges from the node to the deepest leaf is called _____ of the tree.
a) Height
b) Depth
c) Length
d) Width

8) What is a full binary tree?
a) Each node has exactly zero or two children
b) Each node has exactly two children
c) All the leaves are at the same level
d) Each node has exactly one or two children

9) What is a complete binary tree?
a) Each node has exactly zero or two children
b) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from right to left
c) A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right
d) A tree In which all nodes have degree 2

10) Which of the following is not an advantage of trees?

a) Hierarchical structure

b) Faster search

c) Router algorithms

d) Undo/Redo operations in a notepad