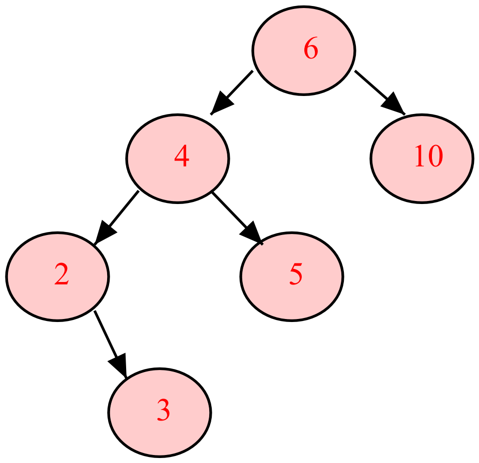
Binary Search Tree

A tree is a connected, acyclic, unidirectional graph. It emulates a tree structure with a set of linked nodes. The topmost node in a tree is called the root node, node of a tree that has child nodes is called an internal node or inner node and the bottom most nodes are called a leaf node. The root is a node that has no parent; it can have only child nodes. Leaves, on the other hand, have no children. The simplest form of a tree is a Binary Tree which has a root node and two subtrees (which is a portion of a tree data structure that can be viewed as a complete tree in itself) - left and right.

A Binary Search Tree is a binary tree in which if a node has value N, all values in its left sub-tree are less than N, and all values in its right sub-tree are greater than N. This property called binary search tree property holds for each and every node in the tree.



Basic operations of Binary Search tree are:

1. Finding a node with a specific data value.
2. Finding a node with minimum or maximum data value.
3. Insertion and deletion of a node.
4. Three (preorder, postorder and inorder) operations for depth-first traversal of a tree
    a. Preorder: Visit a node before traversing it's subtrees
    b. Inorder: Visit a node after traversing it's left subtree but before traversing it's right subtree
    c. Postorder: Visit a node after traversing all of it's subtrees

Binary search tree has three properties: **data** stands for the value of the node, ***left** is the left subtree of a node and ***right** (a pointer of type struct treeNode) is the right subtree of a node.

Algorithm:

Tree's node (treeNode) structure is defined with fields data, *left (pointer to the structure treeNode basically denotes the left subtree of a node) and *right (pointer to the structure treeNode basically denotes the right subtree of a node).

First initialize the value of the root (pointer to the structure treeNode) with NULL.

```
treeNode *root = NULL
```

Functions:

1. Insert function - This function takes the root node of the tree and the value (data) of the node to be inserted as arguments, and returns the pointer to the root node after the new node has been inserted.

   If the node is equal to NULL (i.e. it is an empty tree)
   - Initialize a temp (pointer to the structure treeNode) node and store the data in temp -> data.
   - Initialize temp -> left & temp -> right with NULL (empty left and right subtree).
   - Return temp.

   Otherwise, if data > data value of the node (node->data), then insert the new node at the right subtree.

   ```
   node->right = Insert(node->right , data)
   ```

   Else, if data < data value of the node (node->data), then insert the new node at the left subtree.

   ```
   node->left = Insert(node->left , data)
   ```

   Else there is nothing to do as the data is already in the tree. Return the root node.

2. Delete function - This function takes the root node of the tree and the value (data) of the node to be deleted as arguments, and returns the pointer to the root node after replacing the deleted node.

   If the node is equal to NULL (i.e. it is an empty tree). So, return the same node without deletion.

   Else, if data > data value of the node (node->data), then move to the right subtree.

   ```
   node->right = Delete(node->right , data)
   ```

   Else, if data < data value of the node (node->data), then move to the left subtree.

   ```
   node->left = Delete(node->left , data)
   ```

   Else, now We can delete this node and replace with either minimum element in the right sub tree or maximum element in the left subtree
   - If the node to be deleted has both left and right subtree replace it with the minimum element in the right sub tree, with the help of a temporary node. Find the minimum element in the right subtree using FindMin function and store it in a temporary node. Now, replace the value of the node to be deleted by the value of the temporary node. After replacing, we have to delete the node that replaces the deleted node.
   - Else, if there is only one or zero children then we can connect its parent

to its child or directly remove it from the tree. After all this free the temporary node used.

Return the root node.

3. FindMin function - This function takes the root node of the tree as an argument, and returns the minimum node of the tree.

   If node is equal to NULL, then there is no element in the tree. So, return NULL (Minimum not found).
   Else, if there is a left subtree of the node, go to the left sub tree to find the minimum element. FindMin(node->left).
   Else, return the same node as it will be the minimum node if it doesn't have a left subtree.

4. FindMax function - This function takes the root node of the tree as an argument, and returns the maximum node of the tree.

   If node is equal to NULL, then there is no element in the tree. So, return NULL (Maximum not found).
   Else, if there is a right subtree of the node, go to the right sub tree to find the maximum element. FindMax(node->right).
   Else, return the same node as it will be the maximum node if it doesn't have a right subtree.

5. Find function - This function takes the root node of the tree and the value (data) of the node to be deleted as arguments, and returns the found node of the tree.

   If node is equal to NULL, then there is no element in the tree. So, return NULL (Element not found).
   If there is a right subtree of the node, go to the right sub tree to find the required element. Find(node->right, data).
   Else, if there is a left subtree of the node, go to the left sub tree to find the minimum element. Find(node->left).
   Else, the node is found in either the left or right subtree. Return the node.
   The returned node is stored in a temporary node. If that temporary node = NULL, then the node is not found else it is found.

6. PrintInorder function – This function takes the root node of the tree as an argument and visits all the nodes in the tree in in-order and prints the value of the nodes as they are visited.

   If node is equal to NULL, then there is no element in the tree. Return to the main function.
   Visit the left subtree of the node first, then print the value of that node and lastly visit the right subtree of that node.
   - PrintInorder(node->left)
   - Print node->data
   - PrintInorder(node->right)

7. PrintPreorder function – This function takes the root node of the tree as an argument and visits all the nodes in the tree in pre-order and prints the value of the nodes as they are visited.

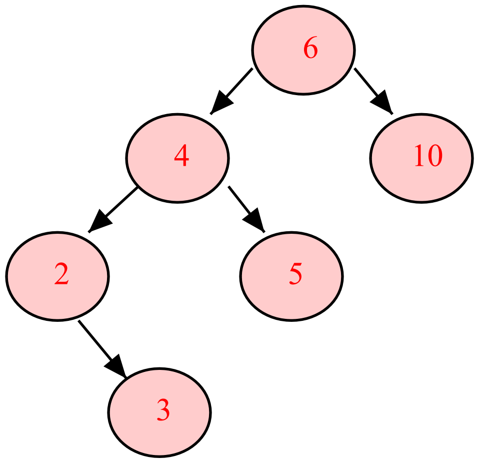
   If node is equal to NULL, then there is no element in the tree. Return to the main function.
   Print the value of the node first, then visit the left subtree of that node and lastly visit the right subtree of that node.
   - Print node->data

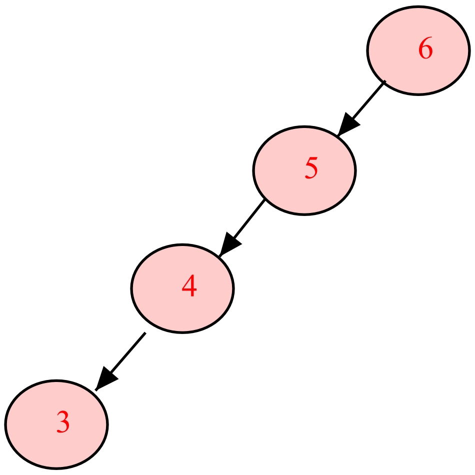
- PrintPreorder(node->left)
- PrintPreorder(node->right)

8. PrintPostorder function – This function takes the root node of the tree as an argument and visits all the nodes in the tree in post-order and prints the value of the nodes as they are visited.
    If node is equal to NULL, then there is no element in the tree. Return to the main function.
    Visit the left subtree of the node first, then visit the right subtree of that node and lastly print the value of that node.
    - PrintPostorder(node->left)
    - PrintPostorder(node->right)
    - Print node->data

Performance:

1. Basic operations takes time proportional to the height of the tree – O(h). Where,
    h = Θ(lgn) for a balanced and for an average tree built by adding the nodes randomly. Where, n is the total number of nodes.
    h = Θ(n) for an balanced tree that resembles a linked list (a linear chain of connected nodes). Where, n is the total number of nodes.





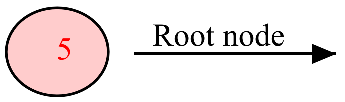
Performance - O(lgn)

Performance - O(n)

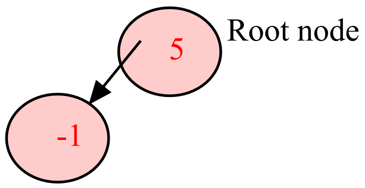
Example:

Start with *root = NULL (Initializes a tree node)

Insert(root, 5)

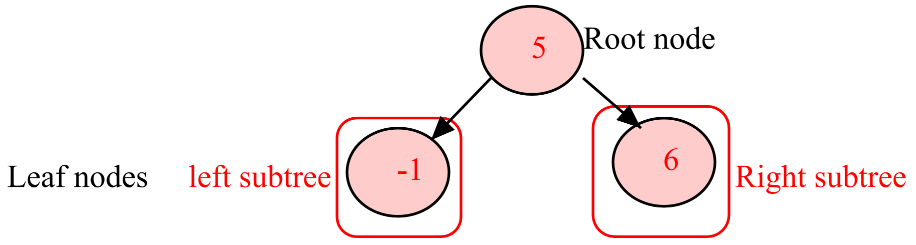


Insert(root, -1): Since, -1 < 5 go to the left child as node->left = NULL -1 can be inserted.

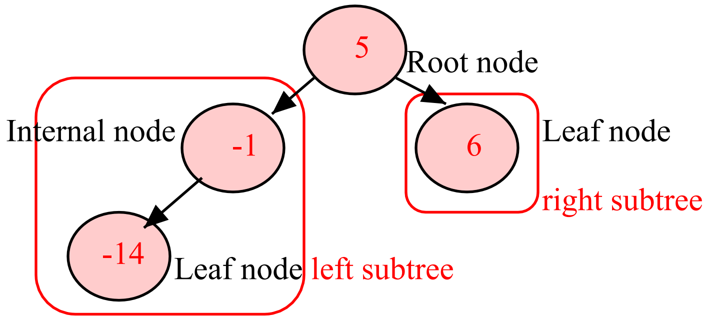


Leaf node (Left Subtree)

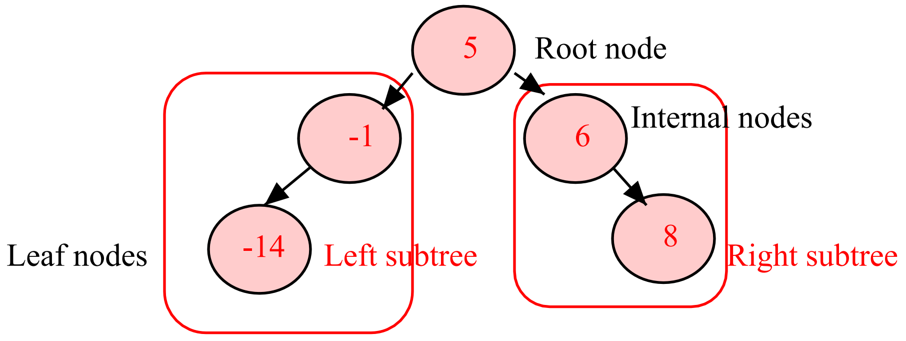
Insert(root, -6): Since, 6 > 5 go to the right child as node->right = NULL 6 can be inserted here.



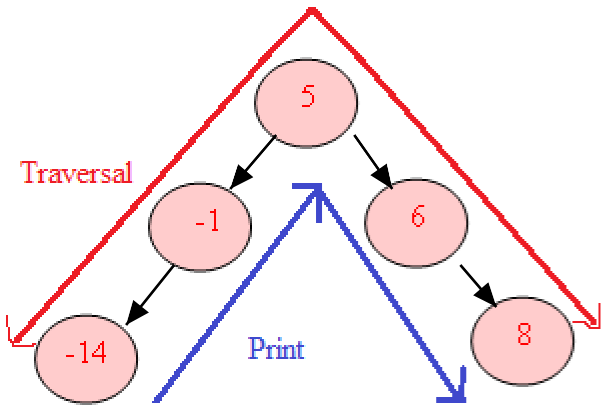
Insert(root, -14): Since, -14 < 5 go to the left child as node->left is not equal to NULL -14 cannot be inserted here. Now, -14 < -1 go to the left child as node->left = NULL -14 can be inserted here.



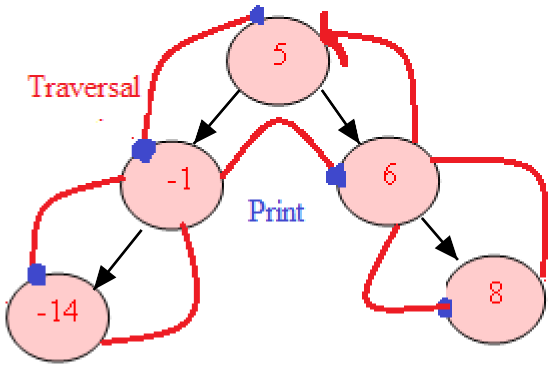
Insert(root, 8): Since, 8 > 5 go to the right child as node->right is not equal to NULL 8 cannot be inserted here. Now, 8 > 6 go to the right child as node->right = NULL 8 can be inserted here.



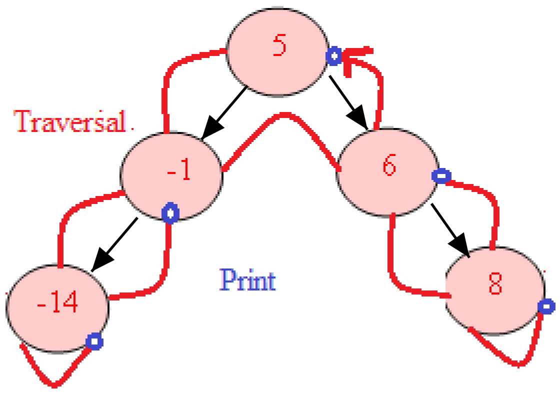
PrintInorder(root) : -14 -1 5 6 8



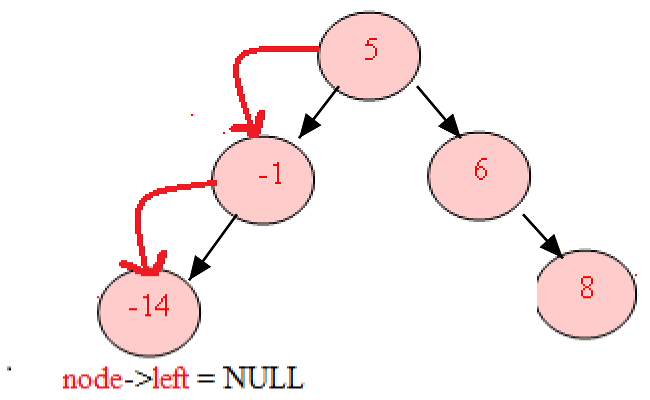
PrintPreorder(root) : 5 -1 -14 6 8



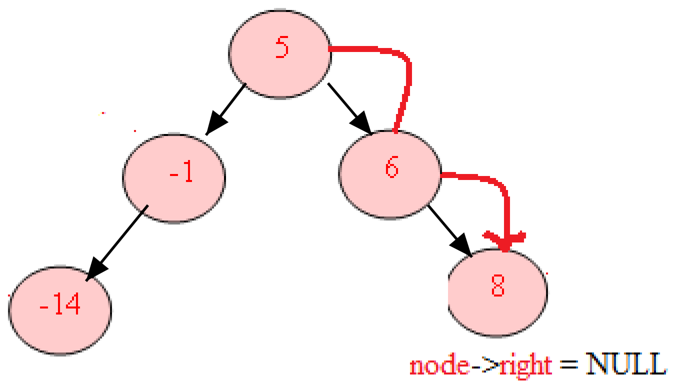
PrintPostorder(root) : -14 -1 6 8 5



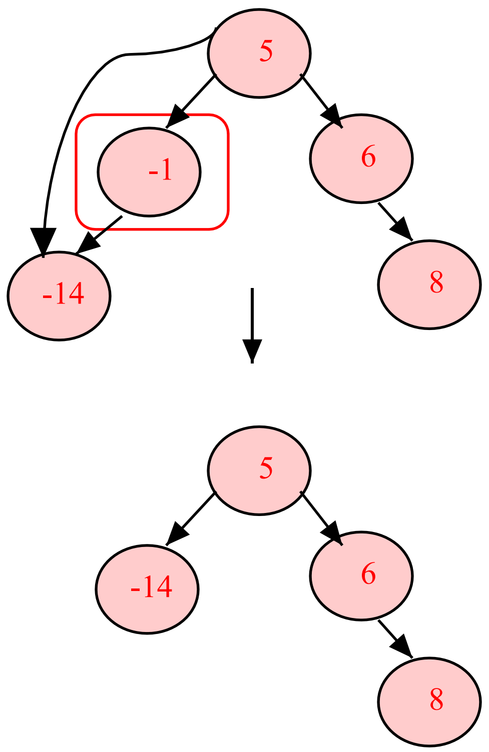
FindMin(root): -14 (branch left until node->left = NULL)



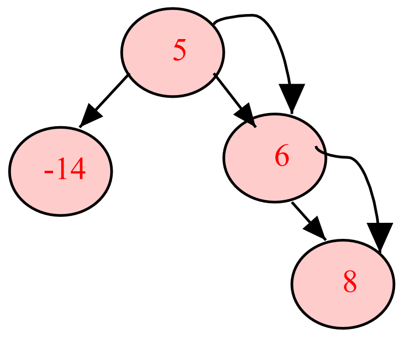
FindMax(root): 8 (branch right until node->right = NULL)



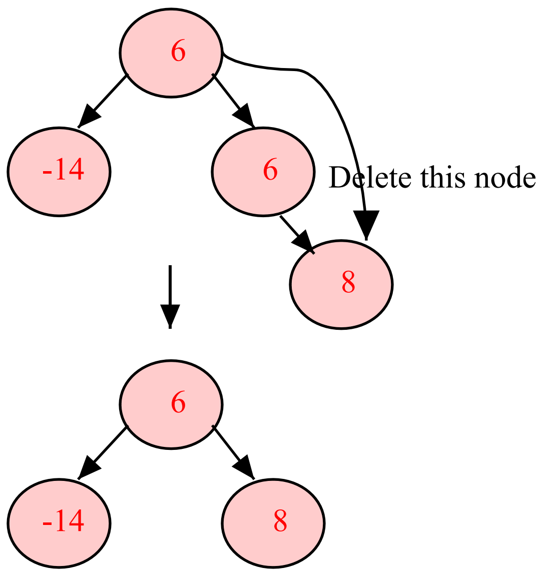
Delete(root,-1): Since, -1 < 5 go to the left child as node->left = -1 and it has only one child(left), remove -1 and connect 5(parent of -1) to-14(child of -1).



Find(root,8): Element is found.



Delete(root,5): Since, it has both children(left and right), replace data value of the root node(5) by 6(minimum value of the right subtree) and delete that node.


6
-14
6
Delete this node
8
6
-14
8