ASSIGNMENT-3

TREAP

Roll-No-214101018

Node structure-

Left	Key	Pointer	Right

- Left- left pointer used for present left child of node
- Right- right pointer used for point right child of node
- Key – store the key value of node and tree is ordered based on key value of node, as a typically binary search tree.
- Priority- store the priority which decide the heap order of tree.
- 0 means highest priority

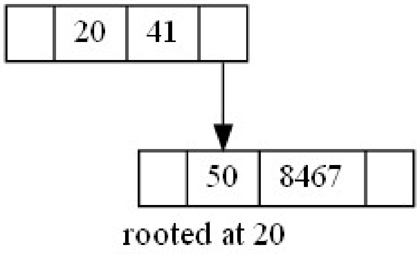
Insert(K) –

Insertion in Treap, we insert an element in tree based on key value and priority of element-

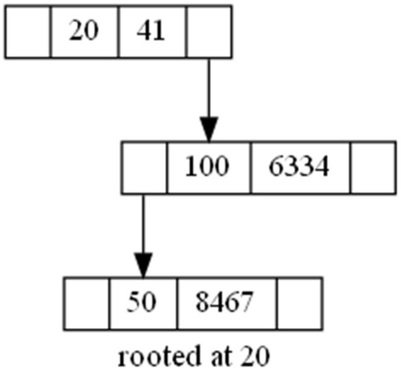
Steps-

1. Find Element in Tree if element is present then then throw exception that element is already present.
2. Assign a random unique priority to node.
3. Insert priority in map so that we can remember the used priority.
4. If element is not present in treap then find the correct position according to BST property and insert.
5. Rearrange the trap based on priority starting from newly inserted key
6. Compare priority with parent node
7. If priority of parent is less, then rearranging the tree
8. For re-arranging tree we de rotation (L-Rotation and R-Rotation)
    a. If priority of parent node is less, then left child node then R-Rotation occur.
    b. If priority of parent node is less, then right child node then L-Rotation occur.

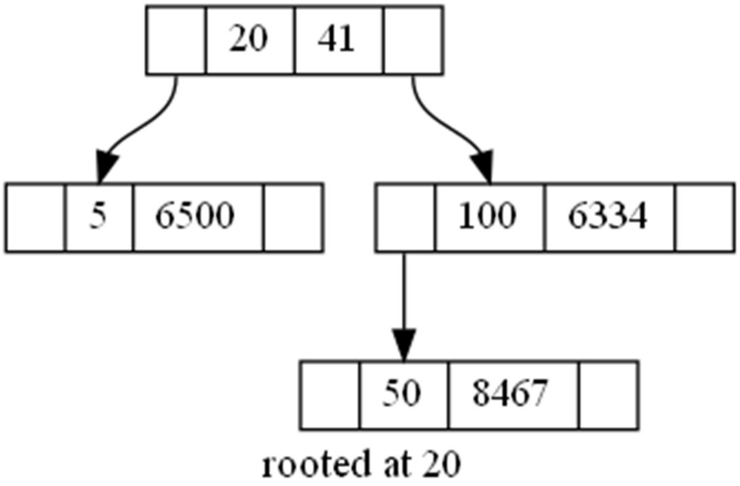
Insert 20,50



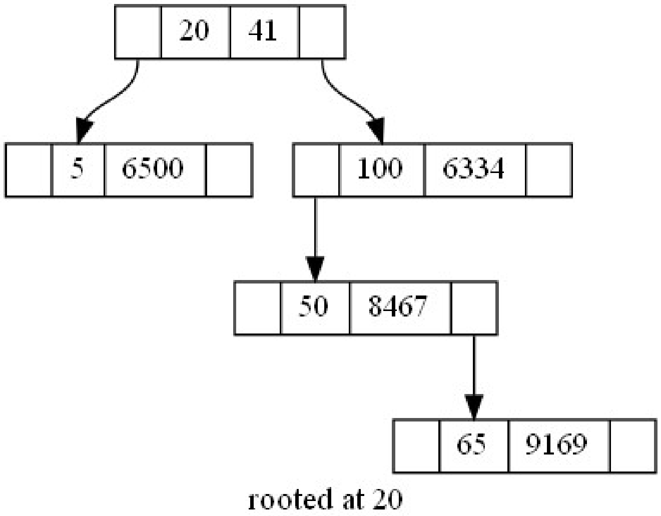
Insert 100



Insert 5

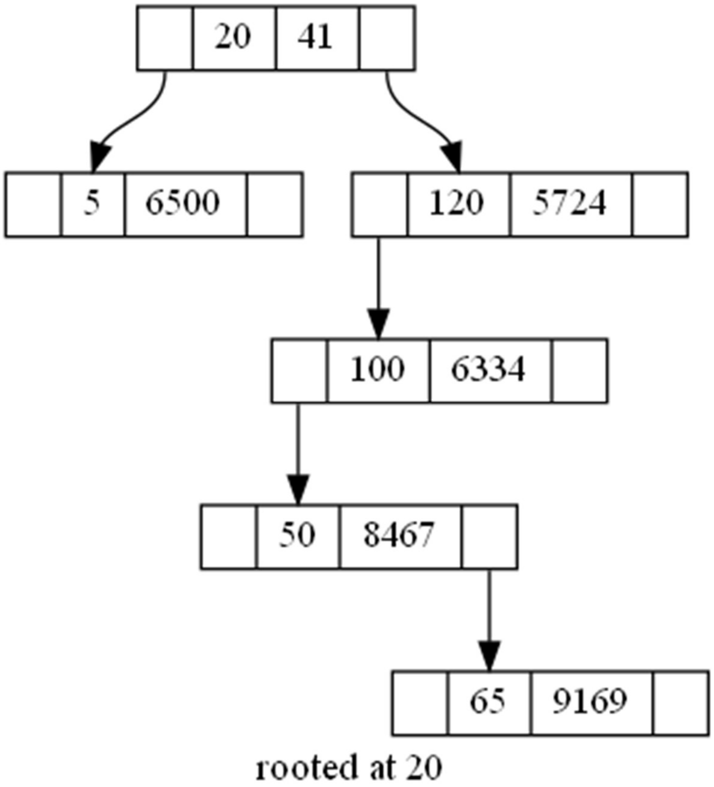


Insert 65



Insert 120

Here Left rotation will happen

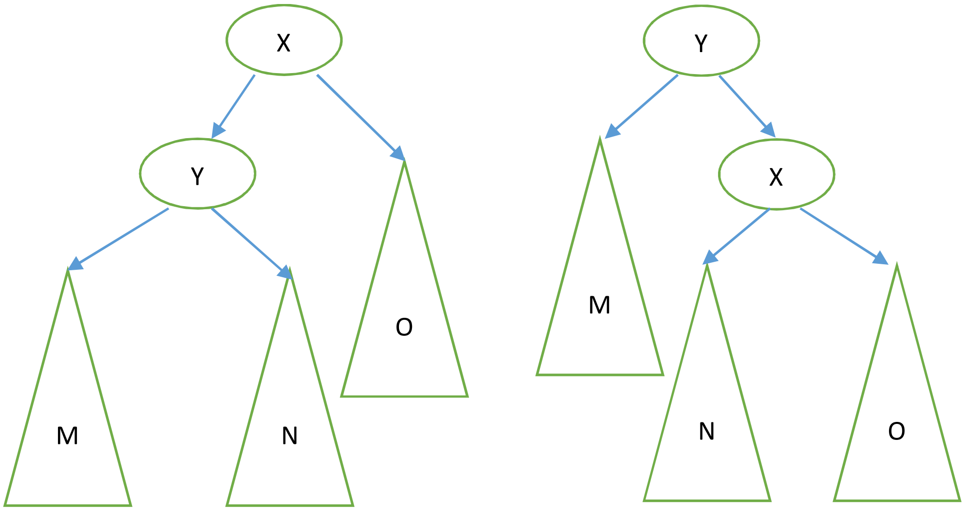


R-Rotation

Right rotation Happen when child node has higher priority than parent node. And child node is on left side of parent node.

Steps –

- If Y node have priority greater then X and Y is on left side of node, then R-Rotation will occur.
- Right child of Y become left child of X
- X become right child of Y
- Y become root



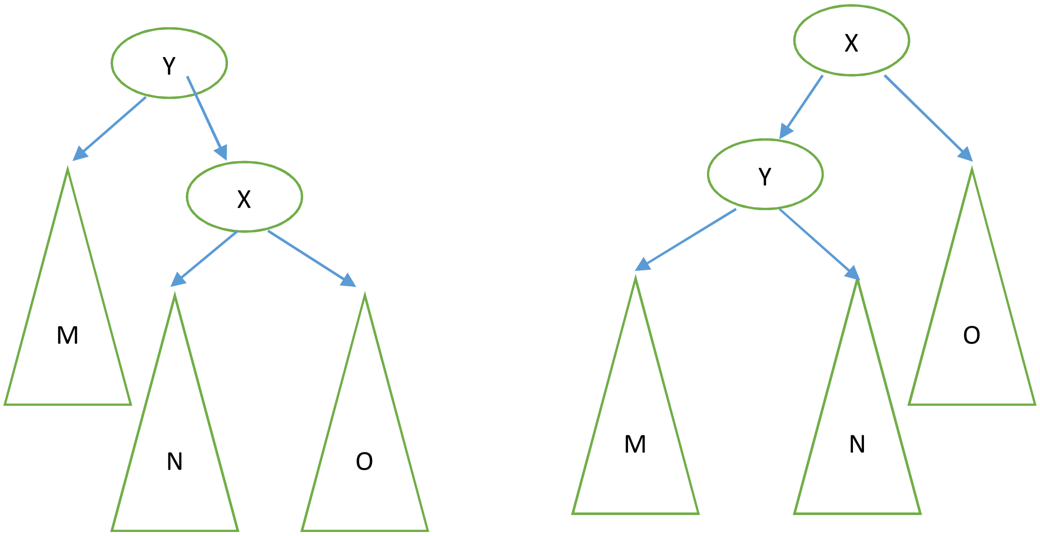
R-Rotation

L-Rotation()

left rotation Happen when child node has higher priority than parent node. And child node is on right side of parent node.

Steps –

- If X node have priority greater then Y and X is on right side of node, then L-Rotation will occur.
- Left child of X become right child of Y
- Y become left child of X
- X become root



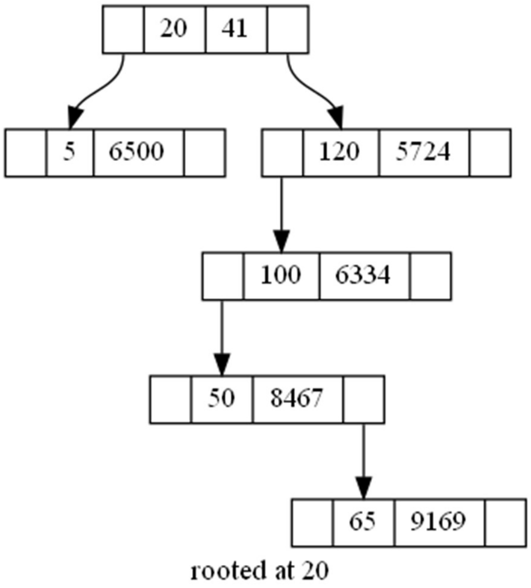
Search (k) -

Treap follow BST property so for searching an element we start from root and compare the key value of root to search element if key value match then Print that Element is present in tree otherwise if search key is less then root key then search element in left subtree otherwise search element in right subtree, if we find the end point of the tree then we print Element is not present in Tree.

Steps –

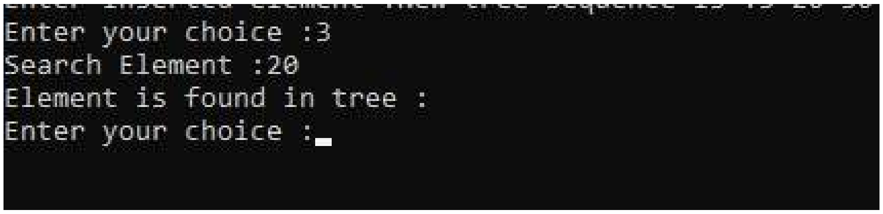
1. Start from root
2. Compare the key value to search key
3. If match return Pointer of node

4. If not match check element in subtree
5. If search key < key of node, then find in left subtree otherwise in right subtree
6. If node is NULL return NULL (means element is not present).



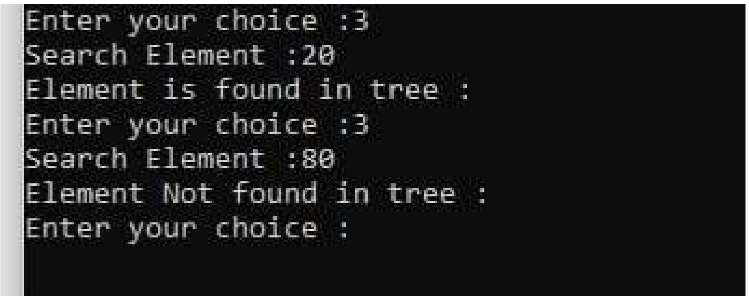
Search Tree

Search 20



```
Enter your choice :3
Search Element :20
Element is found in tree :
Enter your choice :
```

Search – 80



```
Enter your choice :3
Search Element :20
Element is found in tree :
Enter your choice :3
Search Element :80
Element Not found in tree :
Enter your choice :
```

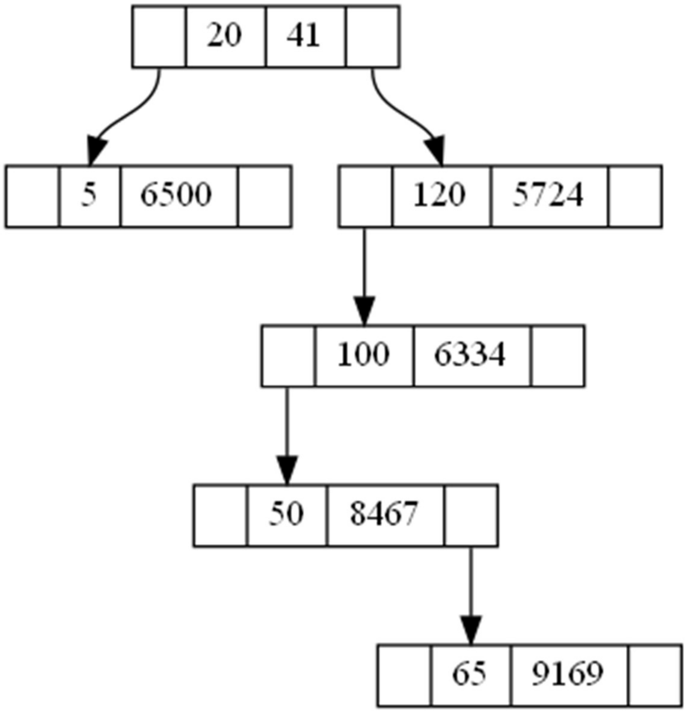
Delete(k) –

Steps –

1. Search the element in tree if element is not present in tree then throw the error element is not present in tree.
2. If element is present in tree, then we find the deleted node of tree.
    a. If deleted element have two subtrees, then find the successor element of node and swap the key value of successor node to deleted node.
    b. New deleted element is successor node of tree.
3. We maintain a stack which contain the elements which is ancestor node of deleted node.
4. Now deleted node have only one child or no child
5. If deleted node have no child, then we pop a node from stack and if deleted node is left then make left as NULL or right as NULL.
6. If deleted node have only one node.
    a. If deleted node is in left subtree, then left child of parent =child of deleted node
    b. If deleted node is in right subtree, then right child of parent =child of deleted node
7. We pop pointers of ancestor from stack and rearrange the tree
    a. If priority of left child is greater than parent node then right rotation will happen
    b. If priority of right child is greater than parent node then left rotation will happen .

Test cases –

This is tree from which we implement delete

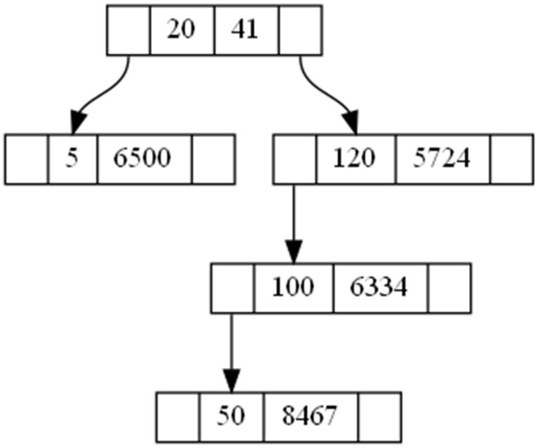


rooted at 20

Delete -

1. Delete 65-

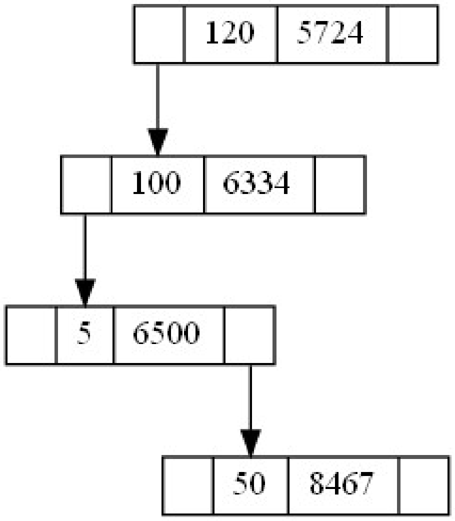
   leaf node deletion



rooted at 20

2. Delete -20

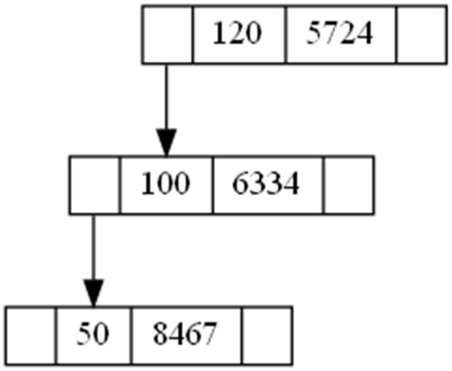
   Delete root element and node which have two children



rooted at 120

3. Delete -5

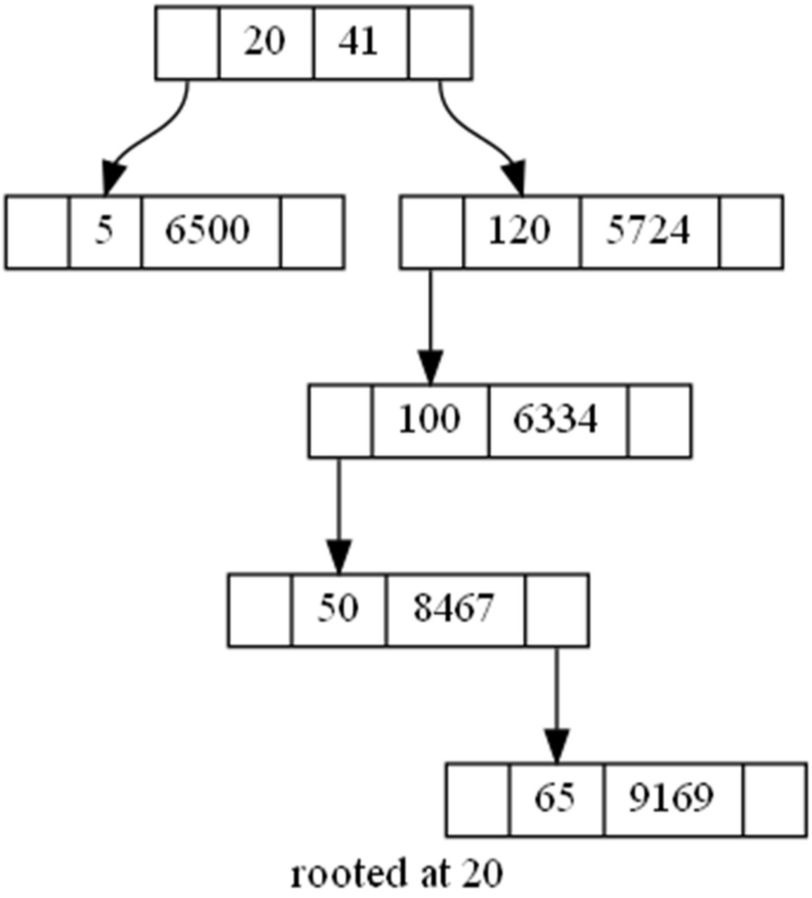
   Delete node which have one child



rooted at 120

Print()

In print function we ask .dot file name for create .dot file .I traverse the tree in inorder traversal and add edge in filename.dot file which is used for create a png image.



--XXX---