```
#include<iostream>
using namespace std;

class Node
{
public:
    int data;
    Node* left;
    Node* right;
    Node* parent;
    Node(int d)
    {
        data = d;
        left = NULL;
        right = NULL;
        parent = NULL;
    }
};
class Heap
{
private:
    Node* root;
    Node* last;
public:
    Heap()
    {
        root = NULL;
        last = NULL;
    }
    Node* getRoot()
    {
        return root;
    }
    void createTree()
    {
        Node* n1 = new Node(1);
        Node* n2 = new Node(2);
        Node* n3 = new Node(3);
        Node* n4 = new Node(4);
        Node* n5 = new Node(5);

        root = n1;
        //1->2,3,  2->4,5

        n1->left = n2;
        n1->right = n3;
        n2->parent = n1;
        n3->parent = n1;

        n2->left = n4;
        n2->right = n5;
        n4->parent = n2;
        n5->parent = n2;

        last = n5;
    }
```

```
    void insert(int x)
    {
        Node* newNode = new Node(x);
        if(root==NULL)
        {
            root = newNode;
            last = newNode;
        }
        else if(last==root)
        {
            root->left = newNode;
            last = newNode;
            last->parent = root;
        }
        else
        {
            if(isLeftNode(last))
            {
                last->parent->right = newNode;
                newNode->parent = last->parent;
                last = newNode;
            }
            else if(isRightNode(last))
            {
                Node* temp = last;

                while(!(temp==root || isLeftNode(temp)) )
                {
                    temp = temp->parent;
                }
                if(temp==root)
                {
                    while(temp->left!=NULL)
                    {
                        temp = temp->left;
                    }
                }
                if(isLeftNode(temp))
                {
                    temp = temp->parent->right;
                }
                // temp is now referring to parent of new last node.
last node will be attached to the left of this node.
                temp->left = newNode;
                newNode->parent = temp;
                last = newNode;
            }
        }
        HeapifyUp(last);
    }
//Node* sibling(Node* nd)
//    {
//        if(isRightNode(nd))
//            return nd->parent->
//        else(isLeftNode(nd))
//            return
//    }
```

```
    bool isHeapNode(Node* target)
    {
        if(isLeafNode(target))
            return true;
        else
            return (target->left->data <= target->data) && (target-
>right->data <= target->data);
    }
    bool isLeftNode(Node* nd)
    {
        return nd == nd->parent->left;
    }
    bool isRightNode(Node* nd)
    {
        return nd == nd->parent->right;
    }
    bool isLeafNode(Node* nd)
    {
        return nd->left==NULL && nd->right==NULL;
    }
    bool isParentHeapNode(Node* target)
    {
        return (target->parent->data < target->data);
    }
    void swapData(Node* nd1, Node* nd2)
    {
        int temp = nd1->data;
        nd1->data = nd2->data;
        nd2->data = temp;
    }
    int deleteRoot()
    {
        if(root==NULL) // empty tree
            return -1;
        else if(root==last) // only 1 node in the tree
        {
            // simply save the root data (the only node in the tree)
and update root/last pointers to NULL
            int deletedRootData = root->data;
            delete root;
            root = NULL;
            last = NULL;
            return deletedRootData;
        }
        else
        {
//            swapNode(root, last);
            int deletedRootData = root->data;
            swapData(root, last); // swap root and last node data
//            root->data = last->data; // replace root data with the
data of the
            updateLast();
            HeapifyDown(root);
            return deletedRootData;
        }
    }
    void HeapifyDown(Node* target)
```

```
    {
        if(target==NULL)
            return;
        else if(!isHeapNode(target))
        {
            Node* smaller = min_child(target);
            swapData(target, smaller);
            HeapifyDown(smaller);
        }
    }
    Node* min_child(Node* nd)
    {
        if(nd->left < nd->right)
            return nd->left;
        else
            return nd->right;
    }
    // This function will delete the old last node, and update the
last pointer accordingly
    void updateLast()
    {
        if(isRightNode(last)) // if current last node is a right node
        {
            Node* temp = last;
            last = last->parent->left;
            last->parent->right = NULL;
            delete temp; // this is necessary to deallocate the memory
assigned to last, as it was allocated dynamically/
        }
        else //if the last node is a left child of some node
        {
            Node* last_ancestor = last->parent;
            while(!(isRightNode(last_ancestor) ||
last_ancestor==root))
            {
                last_ancestor = last_ancestor->parent;
            }
            if(isRightNode(last_ancestor))
            {
                // get sibling of the ancestor. As ancestor is a right
node, so its sibling will be left node of its parent.
                Node* ancestor_sibling = last_ancestor->parent->left;
                Node* temp = ancestor_sibling;
                while(temp->right!=NULL) // after this loop temp will
contain the extreme right node of ancestor_sibling
                    temp = temp->right;
                // now delete the old last node and update new last to
temp
                Node* temp2 = last; // save the old last node
                last->parent->left = NULL;
                delete temp2; // delete the memory for the old last
node
                last = temp; // update the last node            }
            }
            else
            {
                Node* temp = root;
```

```
                while(temp->right!=NULL) // after this loop temp will
contain the extreme right node of root
                    temp = temp->right;
                // now delete the old last and update last to temp
                Node* temp2 = last; // save the old last node
                last->parent->left = NULL;
                delete temp2; // delete the memory for the old last
node
                last = temp; // update the last node
            }
        }
    }

    void HeapifyUp(Node* target)
    {
        if(target==root)
            return;
        if(isParentHeapNode(target))
            return;
//        swapNodes(target, target->parent);
        HeapifyUp(target->parent);
    }

    void traverse_inorder(Node* node1)
    {
        if(node1!=NULL)
        {
            cout<<node1->data<<endl;
            traverse_inorder(node1->left);
            traverse_inorder(node1->right);
        }
    }
};
int main()
{
    Heap tree1;
//    tree1.createTree();
    tree1.insert(10);
    tree1.insert(3);
//    tree1.insert(4);
//    tree1.insert(9);
//    tree1.insert(1);
//    tree1.insert(11);
    tree1.traverse_inorder(tree1.getRoot());

    int x = tree1.deleteRoot();
    cout<<x<<endl<<endl;
    x = tree1.deleteRoot();
//    cout<<x<<endl<<endl;
//    x = tree1.deleteRoot();
//    cout<<x<<endl<<endl;
//    x = tree1.deleteRoot();
//    cout<<x<<endl<<endl;


    return 0;
}
```