```
#include<iostream>
#include<string>
#include<cstdlib>

using namespace std;



#include <iostream>
using namespace std;

struct BSTNode /// BinarySearchTreeNode
{
   int value;
   BSTNode* left = NULL;
   BSTNode* right = NULL;
};

class BST // BinarySearchTree
{
   private:
      BSTNode* root ;
   public:
   BST() //constructor
   {
      root = NULL;
   }
   void insert(int value) //insert algorithm that links towards the helper functions
   {
      root = insert(root, value);
   }
   BSTNode* insert(BSTNode *node, int value)
```

```
  {
    if(node == NULL) //when the node is empty, create a new one with the left and right pointers
    {
      node = new BSTNode;
      node->value = value;
      node->left = NULL;
      node->right = NULL;
    }
    else //otherwise use recursion to insert the value depending on whether it is bigger or smaller
    {
      if(value < node -> value)
        node->left = insert(node->left, value);
      else
        node->right = insert(node->right, value);
    }
    return node;
  }
  ///----------------------------------------
  void remove(int value)
  {
    if(root != NULL) //if the root is not empty
    {
      if(root->value == value) //if the root's value is equal,
      {
        if(root->left == NULL && root->right == NULL) //set the root equal to null if there is nothing
on the left/right node
        {
          root = NULL;
        }
        else if(root->left == NULL) //if the left root is empty, set the right node to be the root
        {
```

```
        root = root->right;
    }
    else //otherwise allocate the most suitable node
    {
        BSTNode *p = root->right;
        while(p->left != NULL)
        {
            p = p->left;
        }
        p->left = root->left;
        root = root->right;
    }
    return;
}
BSTNode *node = root; //set the selected node as the root
BSTNode *parent = NULL; //change the parent into a null
bool found = false;
while(node != NULL && !found) //if the value is not null or found, keep traversing
{
    if(node->value == value) //stop the loop when the value is found
    {
        found = true;
        break;
    }
    parent = node; //traverse the tree
    if(value < node->value)
        node = node->left;
    else
        node = node->right;
}
if (found) //when the value is found..
```

```
{
   //locate the appropriate node to relink into the parent
    if(node->left == NULL && node->right == NULL)
    {

      if(node->left == NULL && node->right == NULL)
         parent->left = NULL;
      else
         parent->right = NULL;
    }
    else if(node->left == NULL) //when the left node is empty
    {
      //allocate a new parent node if there is a left
      if(parent->left == node)
         parent->left = node->right;
      else
         parent->right = node->right;
    }
    else if(node->right == NULL) //when the right node is empty
    {
      //allocate a new parent node if there is a right
      if(parent->left == node)
         parent->left = node->left;
      else
         parent->right = node->left;
    }
    else
    {
       BSTNode *p = node->right;
       while(p->left != NULL)
         p = p->left;
```

```
            p->left = node->left;
            if(parent->left == node)
              parent->left = node->right;
            else
              parent->right = node->right;
          }
          delete node;
        }
      }
    }
    /// ------------------------------------------
    BSTNode* finMin() const //function to find the minimum
    {
      if(root == NULL)
        return NULL;
      else
      {
        BSTNode *curr = root;
        while(curr->left != NULL)
          curr = curr->left;
        return curr;
      }
    }
    /// ------------------------------------------
    BSTNode* finMax() const //function to find the maximum
    {
      if(root == NULL)
        return NULL;
      else
      {
        BSTNode *curr = root;
```

```
        while(curr->right != NULL)
            curr = curr->right;
        return curr;
    }

}
/// ----------------------------------------
void preOrderTraversal() const //function to traverse preorderly
{
    cout << "preOrderTraversal: ";
    preOrderTraversal(root);
    cout << endl;
}
void preOrderTraversal(BSTNode* node) const //traversing with the node
{
    if (node != NULL)
    {
        cout << node->value << " ";
        preOrderTraversal(node->left);
        preOrderTraversal(node->right);
    }
}
/// ----------------------------------------
void inOrderTraversal() const //inorder traversal
{
    cout << "inOrderTraversal: ";
    inOrderTraversal(root);
    cout << endl;
}
void inOrderTraversal(BSTNode* node) const
{
```

```
        if (node != NULL)
        {
            inOrderTraversal(node->left);
            cout << node->value << " ";
            inOrderTraversal(node->right);
        }
    }
    /// ----------------------------------------
    void postOrderTraversal() const //post order traveresal
    {
        cout << "postOrderTraversal: ";
        postOrderTraversal(root);
        cout << endl;
    }
    void postOrderTraversal(BSTNode* node) const
    {
        if (node != NULL)
        {
            postOrderTraversal(node->left);
            postOrderTraversal(node->right);
            cout << node->value << " ";
        }
    }
};

const int SIZE =15;

int main()
{

    BST bst;
```

```
    int values[SIZE] = {5, 2, 12, -4, 3, 9, 21, -7, 19, 25, -8, -6, -4, 3, 12};

    for (int i = 0; i < SIZE; i++)

    bst.insert(values[i]);


    bst.preOrderTraversal(); /// should be 5 2 -4 -7 -8 -6 3 12 9 21 19 25

    bst.inOrderTraversal(); /// should be -8 -7 -6 -4 2 3 5 9 12 19 21 25

    bst.postOrderTraversal(); /// should be -8 -6 7 -4 3 2 9 19 25 21 12 5


    bst.remove(3); /// Node 3 has 0 children --> delete the node and make it NULL;

    bst.remove(-4); /// Node -4 has 1 children --> Link parent to child --> delete the node and make it NULL;

    bst.remove(12); /// Node 12 has 2 children --> findMin for the right subtree --> swap value -> delete


    bst.preOrderTraversal(); /// should be 5 2 -7 -8 -6 19 9 21 25

    bst.inOrderTraversal(); /// should be -8 -7 -6 2 5 9 19 21 25

    bst.postOrderTraversal(); /// should be -8 -6 7 2 9 25 21 19 5


    return 0;

}
```