1. Given class **BinaryTree**, you need to finish methods **getHeight()**, **preOrder()**, **inOrder()**,**postOrder()**.

#include <iostream>  
#include <string>  
#include <algorithm>  
#include <sstream>  
using namespace std;

template<class K, class V>  
class BinaryTree  
{  
public:  
    class Node;  
private:  
    Node\* root;  
public:  
    BinaryTree() : root(nullptr) {}  
    ~BinaryTree()  
    {  
        // You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.  
    }  
    class Node  
    {  
    private:  
        K key;  
        V value;  
        Node\* pLeft, \* pRight;  
        friend class BinaryTree<K, V>;  
    public:  
        Node(K key, V value) : key(key), value(value), pLeft(NULL), pRight(NULL) {}  
        ~Node() {}  
    };  
    void addNode(string posFromRoot, K key, V value)  
    {  
        if (posFromRoot == "")  
        {  
            this->root = new Node(key, value);  
            return;  
        }  
        Node\* walker = this->root;  
        int l = posFromRoot.length();  
        for (int i = 0; i < l - 1; i++)  
        {  
            if (!walker)  
                return;  
            if (posFromRoot[i] == 'L')  
                walker = walker->pLeft;  
            if (posFromRoot[i] == 'R')  
                walker = walker->pRight;  
        }  
        if (posFromRoot[l - 1] == 'L')  
            walker->pLeft = new Node(key, value);  
        if (posFromRoot[l - 1] == 'R')  
            walker->pRight = new Node(key, value);  
    }  
    // STUDENT ANSWER BEGIN   
          
    // STUDENT ANSWER END  
};

**For example:**

| **Test** | **Result** |
| --- | --- |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("", 2, 4); // Add to root  binaryTree.addNode("L", 3, 6); // Add to root's left node  binaryTree.addNode("R", 5, 9); // Add to root's right node  cout << binaryTree.getHeight() << endl;  cout << binaryTree.preOrder() << endl;  cout << binaryTree.inOrder() << endl;  cout << binaryTree.postOrder() << endl; | 2  4 6 9  6 4 9  6 9 4 |

|  |
| --- |
| // STUDENT ANSWER BEGIN  // You can define other functions here to help you.  int height (Node\* r) {  if (r == NULL)  return 0;  else {  /\* compute the height of each subtree \*/  int lheight = height (r -> pLeft);  int rheight = height (r -> pRight);  /\* use the larger one \*/  if (lheight > rheight)  return (lheight + 1);  else return (rheight + 1);  }  }  int getHeight () {  // TODO: return height of the binary tree.  return height (this -> root);  }  string preOrder (Node\* r, string &s) {  // TODO: return the sequence of values of nodes in pre-order.  if (r == NULL)  return "";  /\* first print data of node \*/  s += to\_string (r -> value);  s += " ";  /\* then recur on left sutree \*/  preOrder (r -> pLeft, s);  /\* now recur on right subtree \*/  preOrder (r -> pRight, s);  return s;  }  string inOrder (Node\* r, string &a) {  // TODO: return the sequence of values of nodes in in-order.  if (r == NULL)  return "";  inOrder (r -> pLeft, a);  /\* first print data of node \*/  a += to\_string (r -> value);  a += " ";  /\* then recur on left sutree \*/  /\* now recur on right subtree \*/  inOrder (r -> pRight, a);  return a;  }  string postOrder (Node\* r, string &s) {  // TODO: return the sequence of values of nodes in post-order.  if (r == NULL)  return "";  /\* first print data of node \*/  /\* then recur on left sutree \*/  postOrder (r -> pLeft, s);  /\* now recur on right subtree \*/  postOrder (r -> pRight, s);  s += to\_string (r -> value);  s += " ";  return s;  }  string preOrder () {  // TODO: return the sequence of values of nodes in pre-order.  string s;  return preOrder (this -> root, s);  }  string inOrder () {  // TODO: return the sequence of values of nodes in in-order.  string s;  return inOrder (this -> root, s);  }  string postOrder () {  // TODO: return the sequence of values of nodes in post-order.  string s;  return postOrder (this -> root, s);  }  // STUDENT ANSWER END |

2. Given a Binary tree, the task is to count the number of nodes with two children

#include<iostream>

#include<string>

using namespace std;

template<class K, class V>

class BinaryTree

{

public:

class Node;

private:

Node \*root;

public:

BinaryTree() : root(nullptr) {}

~BinaryTree()

{

// You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.

}

class Node

{

private:

K key;

V value;

Node \*pLeft, \*pRight;

friend class BinaryTree<K, V>;

public:

Node(K key, V value) : key(key), value(value), pLeft(NULL), pRight(NULL) {}

~Node() {}

};

void addNode(string posFromRoot, K key, V value)

{

if(posFromRoot == "")

{

this->root = new Node(key, value);

return;

}

Node\* walker = this->root;

int l = posFromRoot.length();

for (int i = 0; i < l-1; i++)

{

if (!walker)

return;

if (posFromRoot[i] == 'L')

walker = walker->pLeft;

if (posFromRoot[i] == 'R')

walker = walker->pRight;

}

if(posFromRoot[l-1] == 'L')

walker->pLeft = new Node(key, value);

if(posFromRoot[l-1] == 'R')

walker->pRight = new Node(key, value);

}

// STUDENT ANSWER BEGIN

// STUDENT ANSWER END

};

You can define other functions to help you.

**For example:**

| **Test** | **Result** |
| --- | --- |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("",2, 4); // Add to root  binaryTree.addNode("L",3, 6); // Add to root's left node  binaryTree.addNode("R",5, 9); // Add to root's right node  cout << binaryTree.countTwoChildrenNode(); | 1 |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("",2, 4);  cout << binaryTree.countTwoChildrenNode(); | 0 |

|  |
| --- |
| // STUDENT ANSWER BEGIN  // You can define other functions here to help you.  int countTwoChildrenNode (Node\* r, int &n) {  if (r == NULL) return 0;  if (r -> pLeft != NULL && r -> pRight != NULL) n = n + 1;  countTwoChildrenNode (r -> pLeft, n);  countTwoChildrenNode (r -> pRight, n);  return n;  }  int countTwoChildrenNode() {  int n = 0;  return countTwoChildrenNode (this -> root, n);  }  // STUDENT ANSWER END |

3. In this question, you have to perform add **and delete on binary search tree**. Note that:

- When deleting a node which still have 2 children, **take the inorder successor** (smallest node of the right sub tree of that node) to replace it.

- When adding a node which has the same value as parent node, add it in the**left sub tree**.

Your task is to implement two functions: add and deleteNode. You could define one or more functions to achieve this task.

#include <iostream>  
#include <string>  
#include <sstream>  
**using** **namespace** std;  
#define SEPARATOR "#<ab@17943918#@>#"  
**template**<**class** T>  
**class** BinarySearchTree  
{  
**public**:  
    **class** Node;  
**private**:  
    Node\* root;  
**public**:  
    BinarySearchTree() : root(**nullptr**) {}  
    ~BinarySearchTree()  
    {  
        // You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.  
    }

    //Helping function

    **void** add(T value){  
        **//TODO**  
    }  
     
    **void** deleteNode(T value){  
        //TODO  
    }  
    string inOrderRec(Node\* root) {  
        stringstream ss;  
        **if** (root != **nullptr**) {  
            ss << inOrderRec(root->pLeft);  
            ss << root->value << " ";  
            ss << inOrderRec(root->pRight);  
        }  
        **return** ss.str();  
    }  
      
    string inOrder(){  
        **return** inOrderRec(**this**->root);  
    }  
      
    **class** Node  
    {  
    **private**:  
        T value;  
        Node\* pLeft, \* pRight;  
        **friend** **class** BinarySearchTree<T>;  
    **public**:  
        Node(T value) : value(value), pLeft(**NULL**), pRight(**NULL**) {}  
        ~Node() {}  
    };  
};

**For example:**

| **Test** | **Result** |
| --- | --- |
| BinarySearchTree<int> bst;  bst.add(9);  bst.add(2);  bst.add(10);  bst.deleteNode(9);  cout << bst.inOrder(); | 2 10 |
| BinarySearchTree<int> bst;  bst.add(9);  bst.add(2);  bst.add(10);  bst.add(8);  cout << bst.inOrder()<<endl;  bst.add(11);  bst.deleteNode(9);  cout << bst.inOrder(); | 2 8 9 10  2 8 10 11 |

|  |
| --- |
| //Helping functions  void add (T value) {  //TODO  if (root == NULL) root = new Node (value);  else {  Node\* pWalk = root;  Node\* parent = pWalk;  while (pWalk != NULL) {  parent = pWalk;  if (parent -> value >= value) pWalk = pWalk -> pLeft;  else pWalk = pWalk->pRight;  }  if (parent -> value >= value) parent -> pLeft = new Node (value);  else parent -> pRight = new Node (value);  }  }  Node\* findSmallest (Node\* root) {  Node\* current = root;  /\* loop down to find the leftmost leaf \*/  while (current -> pLeft != NULL) {  current = current->pLeft;  }  return current;  }  Node\* DeleteItems (Node\* root, T data, int& t) {  t++;  if (root == NULL) return root;  else if (root -> value > data) root -> pLeft = DeleteItems (root -> pLeft, data, t);  else if (root -> value < data) root -> pRight = DeleteItems (root -> pRight, data, t);  else {  if (root -> pLeft == NULL) {  Node\* temp = root -> pRight;  delete root;  if (t == 1) {this -> root = temp;};  return temp;  }  else if (root -> pRight == NULL) {  Node\* temp = root -> pLeft;  delete root;  if (t == 1) {this -> root = temp;};  return temp;  }  else {  Node\* temp = findSmallest (root -> pRight);  root -> value = temp -> value;  root -> pRight = DeleteItems (root -> pRight, temp -> value, t);  }  }  return root;  }  void deleteNode (T value) {  //TODO  int t = 0;  DeleteItems (root, value, t);  } |

4. Given class **BinaryTree**, you need to finish methods **isBST()** to determine if a tree is a binary search tree (BST)

#include <iostream>  
#include <string>  
#include <algorithm>  
#include <sstream>  
using namespace std;

template<class K, class V>  
class BinaryTree  
{  
public:  
    class Node;  
private:  
    Node\* root;  
public:  
    BinaryTree() : root(nullptr) {}  
    ~BinaryTree()  
    {  
        // You have to delete all Nodes in BinaryTree. However in this task, you can ignore it.  
    }  
    class Node  
    {  
    private:  
        K key;  
        V value;  
        Node\* pLeft, \* pRight;  
        friend class BinaryTree<K, V>;  
    public:  
        Node(K key, V value) : key(key), value(value), pLeft(NULL), pRight(NULL) {}  
        ~Node() {}  
    };  
    void addNode(string posFromRoot, K key, V value)  
    {  
        if (posFromRoot == "")  
        {  
            this->root = new Node(key, value);  
            return;  
        }  
        Node\* walker = this->root;  
        int l = posFromRoot.length();  
        for (int i = 0; i < l - 1; i++)  
        {  
            if (!walker)  
                return;  
            if (posFromRoot[i] == 'L')  
                walker = walker->pLeft;  
            if (posFromRoot[i] == 'R')  
                walker = walker->pRight;  
        }  
        if (posFromRoot[l - 1] == 'L')  
            walker->pLeft = new Node(key, value);  
        if (posFromRoot[l - 1] == 'R')  
            walker->pRight = new Node(key, value);  
    }

    // STUDENT ANSWER BEGIN

// STUDENT ANSWER END  
};

**For example:**

| **Test** | **Result** |
| --- | --- |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("", 2, 4); // Add to root  binaryTree.addNode("L", 3, 2); // Add to root's left node  binaryTree.addNode("R", 5, 9); // Add to root's right node  cout << binaryTree.isBST(); | 1 |
| BinaryTree<int, int> binaryTree;  binaryTree.addNode("", 2, 4);    cout << binaryTree.isBST(); | 1 |

|  |
| --- |
| // STUDENT ANSWER BEGIN  // You can define other functions here to help you.  bool isBST (Node\* r) {  // TODO: return true if a tree is a BST; otherwise, return false.  if (r == NULL) return 1;  if ((r -> pLeft != NULL && r -> value <= r -> pLeft -> value) || (r -> pRight != NULL && r -> value >= r -> pRight -> value)) return 0;  if (!isBST (r -> pLeft) || !isBST (r -> pRight)) return 0;  return 1;  }  bool isBST () {  // TODO: return true if a tree is a BST; otherwise, return false.  return isBST (root);  }  // STUDENT ANSWER END |

5. QuickSort is one of the fastest sorting algorithms for sorting large data. When implemented well, it can be about two or three times faster than its main competitors, such as MergeSort and HeapSort. When the number of elements is below some threshold (min\_size), switch to insertion sort - non-recursive sorting algorithm that performs fewer swaps, comparisons or other operations on such small arrays.

Implement static methods **hybridQuickSort** in class Sorting to sort an array in ascending order. If you do insertion sort, please print "Insertion sort: array" (using printArray) first. If you do quick sort, please print "Quick sort: array (using printArray)" first. Please read example carefully to know exactly what we print.

To match the test cases, please note:

* Using first element as pivot
* Recursively call the hybridQuickSort function to the left of the pivot first, then recursively to the right of the pivot
* Do insertion sort if sub array size smaller than min\_size

#include <iostream>

using namespace std;

template <class T>

class Sorting

{

private:

static T \*Partition(T \*start, T \*end);

public:

static void printArray(T \*start, T \*end)

{

int size = end - start;

for (int i = 0; i < size - 1; i++)

cout << start[i] << ", ";

cout << start[size - 1];

cout << endl;

}

static void insertionSort(T \*start, T \*end);

static void hybridQuickSort(T \*start, T \*end, int min\_size);

};

**For example:**

| **Test** | **Result** |
| --- | --- |
| int array[] = {1, 2, 6, 4, 7, 8, 5, 3};  int min\_size = 4;  Sorting<int>::hybridQuickSort(&array[0], &array[8], min\_size); | Quick sort: 1, 2, 6, 4, 7, 8, 5, 3  Quick sort: 2, 6, 4, 7, 8, 5, 3  Quick sort: 6, 4, 7, 8, 5, 3  Insertion sort: 5, 4, 3  Insertion sort: 8, 7 |
| int array[] = {2, 6, 4, 12, 23, 1, 0, -12};  int min\_size = 4;  Sorting<int>::hybridQuickSort(&array[0], &array[8], min\_size); | Quick sort: 2, 6, 4, 12, 23, 1, 0, -12  Insertion sort: 1, -12, 0  Quick sort: 23, 12, 4, 6  Insertion sort: 6, 12, 4 |
| int array[] = {30, 7, 20, 0, -13, 1, 19, 72};  int min\_size = 3;  Sorting<int>::hybridQuickSort(&array[0], &array[8], min\_size); | Quick sort: 30, 7, 20, 0, -13, 1, 19, 72  Quick sort: 19, 7, 20, 0, -13, 1  Quick sort: -13, 7, 1, 0  Quick sort: 7, 1, 0  Insertion sort: 0, 1  Insertion sort: 20  Insertion sort: 72 |

|  |
| --- |
| template <class T>  T \*Sorting <T>::Partition (T \*start, T \*end) {  T p = start [0]; // pivot  int i = 0;  int j = end - start;  do {  do {  i++;  }  while (start [i] <= p);  do {  j--;  }  while (start [j] > p);  swap (start [i], start [j]);  }  while (i<j);  swap (start [i], start [j]);  swap (start [0], start [j]);  return start + j;  }  template <class T>  void Sorting <T>::insertionSort (T \*start, T \*end) {  for (int i = 0; i < end - start; i++) {  char min = start [i];  int id = i;  for (int j = i; j < end - start; j++) {  if (start[j] < min) {  id = j;  min = start[j];  }  }  int temp = start[i];  start[i] = start[id];  start[id] = temp;  }  }  template <class T>  void Sorting <T>::hybridQuickSort (T \*start, T \*end, int min\_size)  {  if (end - start == 0) return;  int size = end - start;  if (size < min\_size) {  cout << "Insertion sort: ";  printArray (start, end);  insertionSort (start, end);  }  else {  cout << "Quick sort: ";  printArray (start, end);  T \*pivot = Partition (start, end);  hybridQuickSort (start, pivot, min\_size);  hybridQuickSort (pivot + 1, end, min\_size);  }  } |

6. Implement static methods **Merge**and **MergeSort** in class Sorting to sort an array in ascending order. The Merge method has already been defined a call to method printArray so you do not have to call this method again to print your array.

#ifndef SORTING\_H  
#define SORTING\_H  
#include <iostream>  
**using** **namespace** std;  
**template** <**class** T>  
**class** Sorting {  
**public**:  
    /\* Function to print an array \*/  
    **static** **void** printArray(T \*start, T \*end)  
    {  
        **long** size = end - start + 1;  
        **for** (**int** i = 0; i < size - 1; i++)  
            cout << start[i] << ", ";  
        cout << start[size - 1];  
        cout << endl;  
    }  
      
    **static** **void** merge(T\* left, T\* middle, T\* right){  
 /\*TODO\*/  
        Sorting::printArray(left, right);  
    }  
    **static** **void** mergeSort(T\* start, T\* end) {

/\*TODO\*/

}  
};  
#endif /\* SORTING\_H \*/

**For example:**

| **Test** | **Result** |
| --- | --- |
| int arr[] = {0,2,4,3,1,4};  Sorting<int>::mergeSort(&arr[0], &arr[5]); | 0, 2  0, 2, 4  1, 3  1, 3, 4  0, 1, 2, 3, 4, 4 |
| int arr[] = {1};  Sorting<int>::mergeSort(&arr[0], &arr[0]); |  |

|  |
| --- |
| static void merge (T\* left, T\* middle, T\* right) {  /\*TODO\*/  int left\_size = middle - left + 1;  int right\_size = right - middle;    int leftArr [left\_size], rightArr [right\_size];  for (int i = 0; i < left\_size; ++i)  leftArr [i] = left [i];  for (int j = 0; j < right\_size; ++j)  rightArr [j] = middle [j + 1];    int leftIdx = 0;  int rightIdx = 0;  int mergedArrIdx = 0;    while (leftIdx < left\_size && rightIdx < right\_size) {  if (leftArr [leftIdx] <= rightArr [rightIdx]) {  left [mergedArrIdx] = leftArr [leftIdx];  ++leftIdx;  }  else {  left [mergedArrIdx] = rightArr [rightIdx];  ++rightIdx;  }  ++mergedArrIdx;  }    while (leftIdx < left\_size) {  left [mergedArrIdx] = leftArr [leftIdx];  ++leftIdx;  ++mergedArrIdx;  }  while (rightIdx < right\_size) {  left [mergedArrIdx] = rightArr [rightIdx];  ++rightIdx;  ++mergedArrIdx;  }    Sorting::printArray (left, right);  }  static void mergeSort (T\* start, T\* end) {  /\*TODO\*/  if (start < end) {  T\* mid = start + (end - start) / 2;  mergeSort (start, mid);  mergeSort (mid + 1, end);  merge (start, mid, end);  }  } |

7. Implement static methods **Partition**and **QuickSort** in class Sorting to sort an array in ascending order.

#ifndef SORTING\_H  
#define SORTING\_H  
#include <sstream>  
#include <iostream>  
#include <type\_traits>  
using namespace std;  
template <class T>  
class Sorting {  
private:  
    static T\* Partition(T\* start, T\* end) ;  
public:  
    static void QuickSort(T\* start, T\* end) ;  
};  
#endif /\* SORTING\_H \*/

You can read the pseudocode of the algorithm used to in method Partition in the below image.

Text, letter

Description automatically generated

**For example:**

| **Test** | **Result** |
| --- | --- |
| int array[] = { 3, 5, 7, 10 ,12, 14, 15, 13, 1, 2, 9, 6, 4, 8, 11, 16, 17, 18, 20, 19 };  cout << "Index of pivots: ";  Sorting<int>::QuickSort(&array[0], &array[20]);  cout << "\n";  cout << "Array after sorting: ";  for (int i : array) cout << i << " "; | Index of pivots: 2 0 0 6 1 0 2 1 0 0 2 1 0 0 0 0 0 0 1 0  Array after sorting: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 |

|  |
| --- |
| static T\* Partition (T\* start, T\* end) {  // TODO: return the pointer which points to the pivot after rearrange the array.  T p = start [0]; // pivot  int i = 0;  int j = end - start;  do {  do {  i++;  }  while (start [i] <= p);  do {  j--;  }  while (start [j] > p);  swap (start [i], start [j]);  }  while (i < j);  swap (start [i], start [j]);  swap (start [0], start [j]);  return start + j;  }  static void QuickSort (T\* start, T\* end) {  // TODO  // In this question, you must print out the index of pivot in subarray after everytime calling method Partition.  if (start < end) {  T\* i = Partition (start, end);  cout << i - start << " ";  QuickSort (start, i);  QuickSort (i + 1, end);  }  } |