**1.** Given class SplayTree definition:

class SplayTree {  
    struct Node {  
        int val;  
        Node\* pLeft;  
        Node\* pRight;  
        Node\* pParent;  
        Node(int val = 0, Node\* l = nullptr, Node\* r = nullptr, Node\* par = nullptr) : val(val), pLeft(l), pRight(r), pParent(par) { }  
    };  
    Node\* root;

    // print the tree structure for local testing  
    void printBinaryTree(string prefix, const Node\* root, bool isLeft, bool hasRightSibling) {  
        if (!root && isLeft && hasRightSibling) {  
            cout << prefix << "├──\n";  
        }  
        if (!root) return;  
        cout << prefix;  
        if (isLeft && hasRightSibling)   
            cout << "├──";  
        else   
            cout << "└──";  
        cout << root->val << '\n';  
        printBinaryTree(prefix + (isLeft && hasRightSibling ? "|  " : "   "), root->pLeft, true, root->pRight);  
        printBinaryTree(prefix + (isLeft && hasRightSibling ? "|  " : "   "), root->pRight, false, root->pRight);  
    }

    void printPreorder(Node\* p) {  
        if (!p) {  
            return;  
        }  
        cout << p->val << ' ';  
        printPreorder(p->pLeft);  
        printPreorder(p->pRight);  
    }  
public:  
    SplayTree() {  
        root = nullptr;  
    }

~SplayTree() {

// Ignore deleting all nodes in the tree

}

    void printBinaryTree() {  
        printBinaryTree("", root, false, false);  
    }

    void printPreorder() {  
        printPreorder(root);  
        cout << "\n";  
    }

    void splay(Node\* p) {  
        // To Do  
    }

    void insert(int val) {  
        // To Do  
    }  
};

Implement the following method:

1. void splay(Node\* p): bottom-up splaying a Node

When a splay operation is performed on Node p, it will be moved to the root. To perform a splay operation we carry out a sequence of splay steps, each of which moves p closer to the root.

The three types of splay steps are:

- Zig step

Diagram

Description automatically generated

- Zig-zig step:

Diagram

Description automatically generated

- Zig-zag step:

Diagram

Description automatically generated

Note: there are also zag, zag-zag and zag-zig step but we don't show them here

2. void insert(int val):

To insert a value val into a splay tree:

+   Insert val as with a normal binary search tree.

+   When the new value is inserted, a splay operation is performed. As a result, the newly inserted node becomes the root of the tree.

Note: In a splay tree, the values the in left subtree <= root's value <= the values in the right subtree. In this exercise, when inserting a duplicate value, you have to insert it to the right subtree to pass the testcases.

Constraint of testcases:

+   number of operation <= 10^4

+   1 <= val <= 10^5

**For example:**

| **Test** | **Input** | **Result** |
| --- | --- | --- |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  }  // print preorder traversal of the tree  tree.printPreorder();  // print structure of the tree  tree.printBinaryTree(); | 6  insert 50  insert 70  insert 30  insert 80  insert 100  insert 90 | 90 80 30 70 50 100  └──90  ├──80  | └──30  | ├──  | └──70  | └──50  └──100 |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  }  // print preorder traversal of the tree  tree.printPreorder();  // print structure of the tree  tree.printBinaryTree(); | 6  insert 95  insert 200  insert 80  insert 100  insert 200  insert 95 | 95 95 80 200 100 200  └──95  ├──95  | └──80  └──200  └──100  ├──  └──200 |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  }  // print preorder traversal of the tree  tree.printPreorder();  // print structure of the tree | 10  insert 52924  insert 40475  insert 23735  insert 13421  insert 94276  insert 18118  insert 30303  insert 16132  insert 67089  insert 75811 | 75811 67089 16132 13421 30303 18118 23735 52924 40475 94276 |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  }  // print preorder traversal of the tree  tree.printPreorder();  // print structure of the tree | 20  insert 10  insert 2  insert 1  insert 3  insert 5  insert 4  insert 5  insert 7  insert 5  insert 3  insert 3  insert 3  insert 3  insert 7  insert 8  insert 3  insert 5  insert 4  insert 2  insert 2 | 2 2 2 1 3 3 3 3 3 4 3 4 5 5 5 5 7 7 8 10 |

|  |
| --- |
| // write your helper functions here  void rightRotate (Node \*P) {  Node \*T = P -> pLeft;  Node \*B = T -> pRight;  Node \*D = P -> pParent;  if (D) {  if (D -> pRight == P) D -> pRight = T;  else D -> pLeft = T;  }  if (B) B -> pParent = P;  T -> pParent = D;  T -> pRight = P;    P -> pParent = T;  P -> pLeft = B;  }  void leftRotate (Node \*P) {  Node \*T = P -> pRight;  Node \*B = T -> pLeft;  Node \*D = P -> pParent;  if (D) {  if (D -> pRight == P) D -> pRight = T;  else D -> pLeft = T;  }  if (B) B -> pParent = P;  T -> pParent = D;  T -> pLeft = P;    P -> pParent = T;  P -> pRight = B;  }  void splay (Node\* p) {  while (true) {  Node\* pParent = p -> pParent;  if (!pParent) break;  Node\* pp = pParent -> pParent;  if (!pp) { //Zig  if (pParent -> pLeft == p)  rightRotate (pParent);  else  leftRotate (pParent);  break;  }  if (pp -> pLeft == pParent) {  if (pParent -> pLeft == p) { //ZigZig  rightRotate (pp);  rightRotate (pParent);  }  else { //ZigZag  leftRotate (pParent);  rightRotate (pp);  }  }  else {  if (pParent -> pLeft == p) { //ZigZag  rightRotate (pParent);  leftRotate (pp);  }  else { //ZigZig  leftRotate (pp);  leftRotate (pParent);  }  }  }  root = p;  }  void insert (int val) {  if (!root) {  root = (Node\*)malloc(sizeof(Node));  root -> pLeft = NULL;  root -> pRight = NULL;  root -> pParent = NULL;  root -> val = val;  return;  }  Node\* P = root;  while (true) {  if (val < (P -> val)) {  if (P -> pLeft) P = P -> pLeft;  else {  P -> pLeft = (Node\*)malloc(sizeof(Node));  P -> pLeft -> pParent = P;  P -> pLeft -> pRight = NULL;  P -> pLeft -> pLeft = NULL;  P -> pLeft -> val = val;  P = P -> pLeft;  break;  }  }  else {  if (P -> pRight) P = P -> pRight;  else {  P -> pRight = (Node\*)malloc(sizeof(Node));  P -> pRight -> pParent = P;  P -> pRight -> pRight = NULL;  P -> pRight -> pLeft = NULL;  P -> pRight -> val = val;  P = P -> pRight;  break;  }  }  }  splay (P);  } |

**2.** Given class SplayTree definition:

class SplayTree {  
    struct Node {  
        int val;  
        Node\* pLeft;  
        Node\* pRight;  
        Node\* pParent;  
        Node(int val = 0, Node\* l = nullptr, Node\* r = nullptr, Node\* par = nullptr) : val(val), pLeft(l), pRight(r), pParent(par) { }  
    };

    Node\* root;

    // print the tree structure for local testing  
    void printBinaryTree(string prefix, const Node\* root, bool isLeft, bool hasRightSibling) {  
        if (!root && isLeft && hasRightSibling) {  
            cout << prefix << "├──\n";  
        }  
        if (!root) return;  
        cout << prefix;  
        if (isLeft && hasRightSibling)   
            cout << "├──";  
        else   
            cout << "└──";  
        cout << root->val << '\n';  
        printBinaryTree(prefix + (isLeft && hasRightSibling ? "|  " : "   "), root->pLeft, true, root->pRight);  
        printBinaryTree(prefix + (isLeft && hasRightSibling ? "|  " : "   "), root->pRight, false, root->pRight);  
    }

    void printPreorder(Node\* p) {  
        if (!p) {  
            return;  
        }  
        cout << p->val << ' ';  
        printPreorder(p->pLeft);  
        printPreorder(p->pRight);  
    }

public:  
    SplayTree() {  
        root = nullptr;  
    }

    ~SplayTree() {  
        // Ignore deleting all nodes in the tree  
    }

    void printBinaryTree() {  
        printBinaryTree("", root, false, false);  
    }

    void printPreorder() {  
        printPreorder(root);  
        cout << "\n";  
    }

    void splay(Node\* p);

    void insert(int val);

    bool search(int val) {  
        // To Do  
    }  
};

Method splay and insert are already implemented.

You have to implement the following method:

bool search(int val): search for the value val in the tree.

The search operation in splay tree do the same thing as BST search. In addition, it also splays the node containing the value to the root.

+  If the search is successful, the node that is found will become the new root and the function return true.

+  Else, the last accessed node will be splayed and become the new root and the function return false.

Constraints of the testcases:

+    number of operation <= 10^4

+    1 <= val <= 10^5

For example:

| **Test** | **Input** | **Result** |
| --- | --- | --- |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  else if (op == "search")  cout << (tree.search(val) ? "found" : "not found") << '\n';  else if (op == "print")  tree.printPreorder();  }  tree.printBinaryTree(); | 8  insert 95  insert 200  insert 80  search 100  insert 55  insert 100  search 95  print 0 | not found  found  95 55 80 100 200  └──95  ├──55  | ├──  | └──80  └──100  ├──  └──200 |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  else if (op == "search")  cout << (tree.search(val) ? "found" : "not found") << '\n';  else if (op == "print")  tree.printPreorder();  }  tree.printBinaryTree(); | 5  insert 100  insert 200  insert 300  insert 200  search 250 | not found  └──300  └──200  └──200  └──100 |

|  |
| --- |
| // Write your helper functions here  Node\* Search (int val) {  if (!root) return NULL;  Node \*P = root;  while (P) {  if (P -> val == val)  break;  if (val < (P -> val)) {  if (P -> pLeft)  P = P -> pLeft;  else  break;  }  else {  if (P -> pRight)  P = P -> pRight;  else  break;  }  }  splay (P);  if (P -> val == val) return P;  else return NULL;  }  bool search (int val) {  if (Search (val)) return true;  else return false;  } |

**3.** Given class SplayTree definition:

class SplayTree {  
    struct Node {  
        int val;  
        Node\* pLeft;  
        Node\* pRight;  
        Node\* pParent;  
        Node(int val = 0, Node\* l = nullptr, Node\* r = nullptr, Node\* par = nullptr) : val(val), pLeft(l), pRight(r), pParent(par) { }  
    };  
    Node\* root;  
  
    // print the tree structure for local testing  
    void printBinaryTree(string prefix, const Node\* root, bool isLeft, bool hasRightSibling) {  
        if (!root && isLeft && hasRightSibling) {  
            cout << prefix << "├──\n";  
        }  
        if (!root) return;  
        cout << prefix;  
        if (isLeft && hasRightSibling)   
            cout << "├──";  
        else   
            cout << "└──";  
        cout << root->val << '\n';  
        printBinaryTree(prefix + (isLeft && hasRightSibling ? "|  " : "   "), root->pLeft, true, root->pRight);  
        printBinaryTree(prefix + (isLeft && hasRightSibling ? "|  " : "   "), root->pRight, false, root->pRight);  
    }

    void printPreorder(Node\* p) {  
        if (!p) {  
            return;  
        }  
        cout << p->val << ' ';  
        printPreorder(p->pLeft);  
        printPreorder(p->pRight);  
    }  
public:  
    SplayTree() {  
        root = nullptr;  
    }

    ~SplayTree() {  
        // Ignore deleting all nodes in the tree  
    }

    void printBinaryTree() {  
        printBinaryTree("", root, false, false);  
    }

    void printPreorder() {  
        printPreorder(root);  
        cout << "\n";  
    }

    void splay(Node\* p);

    void insert(int val);

    bool search(int val);

    Node\* remove(int val) {  
        // To Do  
    }  
};

The methods splay, insert and search are already implemented.

Implement the following method:

Node\* remove(int val): remove the first Node with value equal to val from the tree and return it.

To perform remove operation on splay tree:

1. If root is NULL, return the root

2. Search for the first node containing the given value val and splay it. If val is present, the found node will become the root. Else the last accessed leaf node becomes the root.

3. If new root's value is not equal to val, return NULL as val is not present.

4. Else the value val is present, we remove root from the tree by the following steps:

    4.1 Split the tree into two tree: tree1 = root's left subtree and tree2 = root's right subtree

    4.2 If tree1 is NULL, tree2 is the new root

    4.3 Else, splay the leaf node with the largest value in tree1. tree1 will be a left skewed binary tree. Make tree2 the right subtree of tree1. tree1 becomes the new root

    4.4 Return the removed node.

![Diagram

Description automatically generated]()

Constraints of the testcases:

+ number of operations <= 10^4

+ 1 <= val <= 10^5

**For example:**

| **Test** | **Input** | **Result** |
| --- | --- | --- |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  else if (op == "remove")  cout << (tree.remove(val) != nullptr ? "removed" : "not found") << '\n';  else if (op == "search")  cout << (tree.search(val) ? "found" : "not found") << '\n';  else if (op == "print")  tree.printPreorder();  }  tree.printBinaryTree(); | 7  insert 100  insert 300  insert 200  insert 50  insert 250  remove 200  print 0 | removed  100 50 250 300  └──100  ├──50  └──250  ├──  └──300 |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  else if (op == "remove")  cout << (tree.remove(val) != nullptr ? "removed" : "not found") << '\n';  else if (op == "search")  cout << (tree.search(val) ? "found" : "not found") << '\n';  else if (op == "print")  tree.printPreorder();  }  tree.printBinaryTree(); | 7  insert 900  insert 1400  insert 100  insert 800  insert 750  remove 500  print 0 | not found  100 750 800 900 1400  └──100  ├──  └──750  ├──  └──800  ├──  └──900  ├──  └──1400 |
| SplayTree tree;  int query;  cin >> query;  for(int i = 0; i < query; i++) {  string op;  int val;  cin >> op >> val;  if (op == "insert")  tree.insert(val);  else if (op == "remove")  cout << (tree.remove(val) != nullptr ? "removed" : "not found") << '\n';  else if (op == "search")  cout << (tree.search(val) ? "found" : "not found") << '\n';  else if (op == "print")  tree.printPreorder();  } | 24  insert 8  insert 2  insert 6  remove 15  print 0  insert 7  insert 1  insert 5  insert 15  insert 11  insert 14  remove 2  remove 14  remove 7  print 0  insert 6  insert 6  insert 8  insert 12  print 0  insert 4  insert 6  remove 8  print 0 | not found  8 6 2  removed  removed  removed  6 5 1 8 11 15  12 8 6 6 6 5 1 8 11 15  removed  6 4 1 6 6 5 6 12 8 11 15 |

|  |
| --- |
| // Write your helper functions here  Node\* findMax (Node\* node) {  while (node -> pRight != NULL) node = node -> pRight;  return node;  }  Node\* remove (int val) {  if (root == NULL) return NULL;  search (val);  if (root -> val != val) return NULL;  Node\* tree1 = root -> pLeft;  Node\* tree2 = root -> pRight;  if (tree1 == NULL) {  Node\* del = root;  root = tree2;  root -> pParent = NULL;  return del;  }  else if (tree2 == NULL) {  Node\* del = root;  root = tree1;  root -> pParent = NULL;  return del;  }  else {  Node\* del = this -> root;  Node\* max = findMax (this -> root -> pLeft);  del = this -> root;  this -> root = this -> root -> pLeft;  this -> root -> pParent = NULL;  del -> pLeft = NULL;  splay (max);  this -> root -> pRight = del -> pRight;  del -> pRight -> pParent = this -> root;  del -> pRight = NULL;  return del;  }  } |

**4.** In this exercise, you are introduced to 2 inner class **Entry** and **Node** of BTree. You should review your slides to see their structures.

In the template declaration before class BTree, K is a type for key variable, D is a type for data variable, M is the degree of BTree.

Method **toString()** of class Entry is already implemented to print as:

**<key,data>**

Your task is to implement method **toString()** of class Node, the return string is of form as:

**[(n)E1E2...En]**

* n is the number of entries of node
* E1, E2, ..., En are the string represents each entry respectively, which is the output of **toString()** of **Entry**.

Please refer to example for print format

#include <iostream>  
#include <sstream>  
#include <string>  
  
using namespace std;  
  
template <class K, class D, int *M*> // K: key, D: data, M: degree of BTree  
class BTree {  
 /// Convention: Left sub-tree < Root's key <= Right sub-tree  
  
public:  
 class Entry;  
 class Node;  
  
private:  
 Node \*root;  
  
public:  
 BTree() : root(0) {};  
 ~BTree() {}  
  
 ///////////////////////////////////////////////////  
 /// CLASS `Entry` ///  
 ///////////////////////////////////////////////////  
public:  
 class Entry {  
 private:  
 K key;  
 D data;  
 Node \*rightPtr;  
  
 friend class BTree<K, D, *M*>;  
  
 public:  
 Entry(K key = K{}, D value = D{}) : key(key), data(value), rightPtr(0) {}  
 ~Entry() {}  
  
 string toString() {  
 stringstream ss;  
 ss << "<"  
 << this->key << ","  
 << this->data  
 << ">";  
 return ss.str();  
 }  
  
 };  
  
 ///////////////////////////////////////////////////  
 /// CLASS `Node` ///  
 ///////////////////////////////////////////////////  
public:  
 class Node {  
 private:  
 Node \*firstPtr;  
 int numEntries;  
 Entry entries[*M* - 1];  
  
 friend class BTree<K, D, *M*>;  
  
 public:  
 Node() : firstPtr(0), numEntries(0) {};  
 ~Node() { }  
  
 bool isFull() {  
 return (numEntries >= *M* - 1);  
 }  
  
 /// BEGIN STUDENT CODE  
 string toString() {  
 stringstream ss;  
 // Fill your code here  
 return ss.str();  
 }  
 /// END STUDENT CODE  
 };

**/////////////////////////////////////////////////////////////  
 ///** CLASS `BTree`: method run sample test **///  
 /////////////////////////////////////////////////////////////**

void testPrintNode(K\* keys, D\* data, int size) {  
 Node node;  
  
 for (int idx = 0; idx < size; idx++) {  
 node.entries[idx].key = keys[idx];  
 node.entries[idx].data = data[idx];  
 }  
 node.numEntries = size;  
 cout << node.toString() << endl;  
 }

};

**For example:**

| **Test** | **Result** |
| --- | --- |
| int keys[] = {3, 5, 7};  int data[] = {33, 55, 77};  int size = sizeof(keys) / sizeof(int);  BTree<int, int, 5>().testPrintNode(keys, data, size); | [(3)<3,33><5,55><7,77>] |

|  |
| --- |
| /// BEGIN STUDENT CODE  string toString() {  stringstream ss;  ss << "["  << "("  << this -> numEntries  << ")";  for (int i = 0; i < this -> numEntries; i++) {  ss << this->entries[i].toString();  }  ss << "]";  return ss.str();  }  /// END STUDENT CODE |

**5.** In this exercise, you need to complete method **toStringPreOrder()** of class BTree, the method prints each node of tree when using pre-order traversal strategy. When printing the information of a node, you should to use method **toString()** of class **Node**. This method is hidden in the below initial code, if you want to run the code in your IDE, you need to use your implementation in question BTree-1.

The test code use method insert() but it is already implemented. You don't need to worry about it on this exercise. You may need it to run this exercise in your IDE, in that case, you can take a look at BTree-4.

You can provide your implementation in this exercise directly, using PreCheck button to see the result in some examples without any penalty. This exercise implementation is also needed for exercise BTree-4.

#include <iostream>  
#include <sstream>  
#include <string>  
  
using namespace std;  
  
template <class K, class D, int *M*> // K: key, D: data, M: degree of BTree  
class BTree {  
 /// Convention: Left sub-tree < Root's key <= Right sub-tree  
  
public:  
 class Entry;  
  
 class Node;  
  
private:  
 Node \*root;  
  
public:  
 BTree() : root(0) {};  
  
 ~BTree() {}  
 ///////////////////////////////////////////////////  
 /// CLASS `Entry` ///  
 ///////////////////////////////////////////////////  
public:  
 class Entry {  
 private:  
 K key;  
 D data;  
 Node \*rightPtr;  
  
 friend class BTree<K, D, *M*>;  
  
 public:  
 Entry(K key = K{}, D value = D{}) : key(key), data(value), rightPtr(0) {}  
 ~Entry() {}  
 string toString() {  
 stringstream ss;  
 // Fill your own code here  
 ss << "<"  
 << this->key << ","  
 << this->data  
 << ">";  
 return ss.str();  
 }  
 };  
  
 ///////////////////////////////////////////////////  
 /// CLASS `Node` ///  
 ///////////////////////////////////////////////////  
public:  
 class Node {  
 private:  
 Node \*firstPtr;  
 int numEntries;  
 Entry entries[*M* - 1];  
  
 friend class BTree<K, D, *M*>;  
  
 public:  
 Node() : firstPtr(0), numEntries(0) {};  
  
 ~Node() {  
 delete firstPtr;  
 }  
 string toString() {  
 stringstream ss;  
 // Hidden code  
 return ss.str();  
 }  
 bool isFull() {  
 return (numEntries >= *M* - 1);  
 }  
 };  
  
 /////////////////////////////////////////////////////////////  
 /// CLASS `BTree`: method implementation ///  
 /////////////////////////////////////////////////////////////  
public:  
 /// BEGIN STUDENT CODE  
 /// ==========================================================  
 /// toStringPreOrder()  
   
 /// END STUDENT CODE  
};

**For example:**

| **Test** | **Result** |
| --- | --- |
| int keys[] = {78, 21, 14};  int size = sizeof(keys) / sizeof(int);  BTree<int, int, 5> bTree;  for (int idx = 0; idx < size; idx++) {  bTree.insert(keys[idx], idx);  }  cout << bTree.toStringPreOrder(); | [(3)<14,2><21,1><78,0>] |
| int keys[] = {78, 21, 14, 7, 10};  int size = sizeof(keys) / sizeof(int);  BTree<int, int, 3> bTree;  for (int idx = 0; idx < size; idx++) {  bTree.insert(keys[idx], idx);  }  cout << bTree.toStringPreOrder(); | [(2)<10,4><21,1>] [(1)<7,3>] [(1)<14,2>] [(1)<78,0>] |

|  |
| --- |
| // You may need some helping functions here  void BTreeTraversal (Node \*root, string &str) {  int scanCount = 0;  Node \*ptr = root -> firstPtr;  str += root -> toString();  str += " ";  while (scanCount <= root -> numEntries) {  if (ptr != NULL) {  BTreeTraversal (ptr, str);  }  if (scanCount <= root->numEntries) {  ptr = root -> entries [scanCount].rightPtr;  }  scanCount = scanCount + 1;  }  return;  }  string toStringPreOrder() {  string str = "";  BTreeTraversal (this -> root, str);  return str;  }  /// END STUDENT CODE |

**6.** In this exercise, you need to complete method **search** for BTree. Method declaration as

**bool search(const K& key, Node\*& outNodePtr, int& outEntryIdx);**

When implementing, you need to assign appropriate value to**outNodePtr**, **outEntryIdx**which will be used for checking your code.

Method **toString()** of class Node is hidden. If you want to see the result in your IDE, you need to provide your previous implementation of these methods.

The test code use method insert() but it is already implemented. You don't need to worry about it on this exercise. You may need it to run this exercise in your IDE, in that case, you can take a look at BTree-4.

You can provide your implementation in this exercise directly, using PreCheck button to see the result in some examples without any penalty.

#include <iostream>  
#include <sstream>  
#include <string>  
  
using namespace std;  
  
template <class K, class D, int *M*> // K: key, D: data, M: order  
class BTree {  
 /// Convention: Left sub-tree < Root's key <= Right sub-tree  
  
public:  
 class Entry;  
 class Node;  
  
private:  
 Node\* root;  
  
public:  
 BTree() : root(0) { }  
 ~BTree() { }  
  
 ///////////////////////////////////////////////////  
 /// CLASS `Entry` ///  
 ///////////////////////////////////////////////////  
public:  
 class Entry {  
 private:  
 K key;  
 D data; // accept all type  
 Node\* rightPtr;  
 friend class BTree<K, D, *M*>;  
  
 public:  
 Entry(K key = K{}, D value = D{}) : key(key), data(value), rightPtr(0) {}  
 ~Entry() { }

string toString() {

stringstream ss;

ss << "<"

<< this->key << ","

<< this->data

<< ">";

return ss.str();

}

};

///////////////////////////////////////////////////

/// CLASS `Node` ///

///////////////////////////////////////////////////

public:

class Node {

private:

Node\* firstPtr;

int numEntries;

Entry entries[M-1];

friend class BTree<K, D, M>;

public:

Node() : firstPtr(0), numEntries(0) {};

~Node() { }

string toString() {

// hidden code

return ss.str();

}

bool isFull() {

return (numEntries >= M-1);

}

};

/////////////////////////////////////////////////////////////

/// CLASS `BTree`: method implementation ///

/////////////////////////////////////////////////////////////

public:

string toStringPreOrder();

void insert(const K& key, const D& data);

/// BEGIN STUDENT CODE

/// ==================================================

/// search(key, outNodePtr, outEntryIdx)

bool search(const K& key, Node\*& outNodePtr, int& outEntryIdx) {

}

/// END STUDENT CODE

};

**For example:**

| **Test** | **Result** |
| --- | --- |
| int keys[] = {78, 21, 14};  int size = sizeof(keys) / sizeof(int);  BTree<int, int, 5> bTree;  for (int idx = 0; idx < size; idx++) {  bTree.insert(keys[idx], idx + 5);  }  BTree<int, int, 5>::Node\* outNode;  int outEntryIdx;  if (bTree.search(11, outNode, outEntryIdx)) {  cout << "FOUND" << endl;  cout << outNode->toString() << endl;  cout << "Index: " << outEntryIdx;  }  else {  cout << "NOT FOUND";  } | NOT FOUND |
| int keys[] = {78, 21, 14};  int size = sizeof(keys) / sizeof(int);  BTree<int, int, 5> bTree;  for (int idx = 0; idx < size; idx++) {  bTree.insert(keys[idx], idx + 5);  }  BTree<int, int, 5>::Node\* outNode;  int outEntryIdx;  if (bTree.search(21, outNode, outEntryIdx)) {  cout << "FOUND" << endl;  cout << outNode->toString() << endl;  cout << "Index: " << outEntryIdx;  }  else {  cout << "NOT FOUND";  } | FOUND  [(3)<14,7><21,6><78,5>]  Index: 1 |

|  |
| --- |
| /// BEGIN STUDENT CODE  /// ======================================================  // You may need some helping functions here  bool BTreeSearch (Node \*root, K target, Node \*&node, int &entryNo) {  if (root == NULL) return 0;  bool found = 0;  if (target < root -> entries [0].key)  return BTreeSearch (root -> firstPtr, target, node, entryNo);  else {  entryNo = root -> numEntries - 1;  while (target < root -> entries [entryNo].key) {  entryNo = entryNo - 1;  if (entryNo < 0)  break;  }  if (entryNo < 0)  found = 0;  else if (target == root -> entries [entryNo].key) {  found = true;  node = root;  }  else  return BTreeSearch (root -> entries [entryNo].rightPtr, target, node, entryNo);  }  return found;  }  bool search (const K &key, Node \*&outNodePtr, int &outEntryIdx) {  return BTreeSearch (root, key, outNodePtr, outEntryIdx);  }  /// END STUDENT CODE |

**7.** In this exercise, you have to implement method insert of class BTree, the declaration of method is

**void insert(const K& key, const D& data);**

The degrees of BTree is **M**, which is declared with template.

Recall that an entry have 3 fields:

* key: type K
* data: type D
* rightPtr: type Node\*

You should read the slides for the algorithm of method **insert()**. The difference is our method takes key, data and specifies order in **key**, not **data.key** as in the slides.

Method **toString()** of class Node is hidden, and method toStringPreOrder() is also need to print the results. If you want to see the result in your IDE, you need to provide your previous implementation of these methods.

#include <iostream>  
#include <sstream>  
#include <string>  
  
using namespace std;  
  
template <class K, class D, int *M*> // K: key, D: data, M: degree of BTree  
class BTree {  
 /// Convention: Left sub-tree < Root's key <= Right sub-tree  
  
public:  
 class Entry;  
  
 class Node;  
  
private:  
 Node \*root;  
  
public:  
 BTree() : root(0) {};  
  
 ~BTree() {}  
 ///////////////////////////////////////////////////  
 /// CLASS `Entry` ///  
 ///////////////////////////////////////////////////  
public:  
 class Entry {  
 private:  
 K key;  
 D data;  
 Node \*rightPtr;  
  
 friend class BTree<K, D, *M*>;  
  
 public:  
 Entry(K key = K{}, D value = D{}) : key(key), data(value), rightPtr(0) {}  
 ~Entry() {}  
 string toString() {  
 stringstream ss;  
 ss << "<"

<< this->key << ","

<< this->data

<< ">";  
 return ss.str();  
 }  
 };  
  
 ///////////////////////////////////////////////////  
 /// CLASS `Node` ///  
 ///////////////////////////////////////////////////  
public:  
 class Node {  
 private:  
 Node \*firstPtr;  
 int numEntries;  
 Entry entries[*M* - 1];  
  
 friend class BTree<K, D, *M*>;  
  
 public:  
 Node() : firstPtr(0), numEntries(0) {};  
  
 ~Node() { }  
 string toString() {  
 stringstream ss;  
 // Fill your own code here  
 return ss.str();  
 }  
 bool isFull() {  
 return (numEntries >= *M* - 1);  
 }  
 };  
  
 /////////////////////////////////////////////////////////////  
 /// CLASS `BTree`: method implementation ///  
 /////////////////////////////////////////////////////////////  
public:  
 /// toStringPreOrder()  
 string toStringPreOrder();  
 /// BEGIN STUDENT CODE  
 void insert(const K& key, const D& data) {

}  
 /// END STUDENT CODE  
};

**For example:**

| **Test** | **Result** |
| --- | --- |
| BTree<int, int, 5> bTree;  int keys[] = {78};  int size = sizeof(keys) / sizeof(int);  for (int idx = 0; idx < size; idx++) {  bTree.insert(keys[idx], idx);  }  cout << bTree.toStringPreOrder() << endl; | [(1)<78,0>] |
| BTree<int, int, 3> bTree;  int keys[] = {9, 21, 14};  int size = sizeof(keys) / sizeof(int);  for (int idx = 0; idx < size; idx++) {  bTree.insert(keys[idx], idx);  }  cout << bTree.toStringPreOrder() << endl; | [(1)<14,2>] [(1)<9,0>] [(1)<21,1>] |

|  |
| --- |
| /// BEGIN STUDENT CODE  /// =====================================================  /// Proposal methods from slides  /// Feel free to use them,  /// or delete them and implement your own helping methods  void insertEntry (Node \*node, int entryNdx, const Entry &newEntry) {  int shifter = node -> numEntries;  while (shifter > entryNdx + 1) {  node -> entries [shifter] = node -> entries [shifter - 1];  shifter = shifter - 1;  }  node -> entries [shifter] = newEntry;  return;  }  void splitNode (Node \*node, int entryNdx, Entry &upEntry) {  int minEntries = (M/2);  Node \*rightPtr = new Node();  int fromNdx = 0;  // Build right subtree node  if (entryNdx <= minEntries - 1)  fromNdx = minEntries;  else  fromNdx = minEntries + 1;  rightPtr -> numEntries = node -> numEntries - fromNdx;  int toNdx = 0;  while (fromNdx < node -> numEntries) {  rightPtr -> entries [toNdx] = node -> entries [fromNdx];  fromNdx = fromNdx + 1;  toNdx = toNdx + 1;  }  node -> numEntries = node -> numEntries - rightPtr -> numEntries;  if (entryNdx <= minEntries - 1) {  insertEntry (node, entryNdx, upEntry);  }  else {  insertEntry (rightPtr, entryNdx - minEntries - 1, upEntry);  node -> numEntries = node -> numEntries - 1;  rightPtr -> numEntries = rightPtr -> numEntries + 1;  }  // Build entry for parent  int medianNdx = minEntries;  upEntry.data = node -> entries [medianNdx].data;  upEntry.key = node -> entries [medianNdx].key;  upEntry.rightPtr = rightPtr;  rightPtr -> firstPtr = node -> entries [medianNdx].rightPtr;  return;  }  int searchNode (Node \*nodePtr, K target) {  int walker = 0;  if (target < nodePtr -> entries [0].key)  walker = -1;  else {  walker = nodePtr -> numEntries - 1;  while (target < nodePtr -> entries [walker].key) {  walker = walker - 1;  if (walker < 0)  return walker;  }  }  return walker;  }  bool insertNode (Node \*&root, const K &key, const D &data, Entry &upEntry) {  bool taller = 0;  if (root == NULL) {  upEntry.key = key;  upEntry.data = data;  upEntry.rightPtr = NULL;  taller = 1;  }  else {  Node\* subTree = root -> firstPtr;  int entryNdx = searchNode (root, key);  if (entryNdx >= 0)  subTree = root -> entries [entryNdx].rightPtr;  else  subTree = root -> firstPtr;  taller = insertNode (subTree, key, data, upEntry);  if (taller) {  if (root -> isFull()) {  splitNode (root, entryNdx, upEntry);  taller = 1;  }  else {  insertEntry (root, entryNdx, upEntry);  taller = 0;  root -> numEntries = root -> numEntries + 1;  }  }  }  return taller;  }  Node \*BTreeInsert (Node \*root, const K &key, const D &data) {  Entry upEntry;  bool taller = insertNode (root, key, data, upEntry);  if (taller) {  Node\* newPtr = new Node();  newPtr -> entries[0] = upEntry;  newPtr -> firstPtr = root;  newPtr -> numEntries = 1;  root = newPtr;  }  return root;  }  /// TODO: Implement following method  void insert (const K &key, const D &data) {  this -> root = BTreeInsert (this -> root, key, data);  }  /// END STUDENT CODE |