**C++ Workshop – 150018**

**Homework Assignment #9**

**Templates and Trees**

Important Notice: This exercise must be submitted by the exam date on: 22/06/2021.

Question 1

1. Add the following methods to the class **Tree** that was discussed in class:
   * **int** **height**() – returns the height of the tree (-1 for empty tree, 0 for tree with just a root node, etc.)
   * **void** **reflect**() – swaps the children of each node to make a mirror image of the original tree.
   * **void** **breadthScan** () – scans the tree nodes horizontally and prints the node values level by level (starting from the root) from left to right. This method should use a queue. The abstract **Queue** class should be used, as well as one of the derived classes (vector or list) that inherit from **Queue** and implement its interface.
2. Add the following methods to the class **SearchTree** that inherits from **Tree**:
   * **void remove(T val)** – removes from the tree the node whose value is val.
   * **T successor(T val)** – returns the value of the succesor of the node whose value is val. If val is not found the method should throw an exception no successor
   * **void deleteDuplicates()** – deletes all duplicates in tree.
3. Comments:

* In the derived class **SearchTree**, the The **root** data member should be used as follows: **Tree<T>::root** and the declaration of a pointer to Node as **typename Tree<T>::Node\***
* The classes **Tree** and **SearchTree** appear as appendices after question 2. You will of course need to add to these classes what is required by the question.
* The **Queue** and **QueueVector** classes are presented as appendices (after question 2) Please note, these classes are presented as non-abstract classes - they must be redefined to be abstract classes for the question.
* For the method **remove**, three cases must be considered:

1. vertex that is a leaf (that is, has no children)
2. vertex with an only child
3. vertex with two children

* The methods on trees should be implemented recursively.

1. Add the classes you wrote to the following main program and test for correctness:

#include <iostream>

using namespace std;

#include "SearchTree.h"

enum { END, ADD, SEARCH, REMOVE, BREADTHSCAN, HEIGHT, SUCCESSOR, DELETEDUP, REFLECT };

int main()

{

SearchTree<int> T1;

cout << "enter 10 numbers:\n";

int x, y, z;

for (int i = 0; i < 10; i++)

{

cin >> x;

T1.add(x);

}

cout << "inorder: ";

T1.inOrder();

cout << "\nenter 0-8:\n";

cin >> x;

while (x != END)

{

switch (x)

{

case ADD: cout << "enter a number: ";

cin >> y;

T1.add(y);

cout << "after adding " << y << ": ";

T1.inOrder();

cout << endl;

break;

case SEARCH: cout << "enter a number: ";

cin >> y;

if (T1.search(y))

cout << "exist" << endl;

else

cout << "no exist" << endl;

break;

case REMOVE:cout << "enter a number: ";

cin >> y;

try

{

T1.remove(y);

cout << "after removing " << y << ": ";

T1.inOrder();

cout << endl;

}

catch (const char\* str)

{

cout << str << endl;

}

break;

case BREADTHSCAN: cout << "breadth scan: ";

T1.breadthScan();

cout << endl;

break;

case HEIGHT:cout << "height of tree: " << T1.height() << endl;

break;

case SUCCESSOR:cout << "enter a number: ";

cin >> y;

try

{

z = T1.successor(y);

cout << "successor of " << y << " is: " << z << endl;

}

catch (const char\* str)

{

cout << str << endl;

}

break;

case DELETEDUP: cout << "after delete duplicate: ";

T1.deleteDuplicates();

T1.inOrder();

cout << endl;

break;

case REFLECT:T1.reflect();

cout << "reflected tree: ";

T1.inOrder();

T1.reflect();

cout << endl;

break;

}

cout << "enter 0-8:\n";

cin >> x;

}

return 0;

}

Question 2

Write a class to represent books in a library. The class should contain the following data members:

* 1. book number (int)
  2. author name (string)
  3. book name (string)

Add to the class (at least) the following methods:

1. constructor: which has a default value for its data members (0 or empty string)
2. operator <, >, <=, >=, ==, !=
   * relational operators: comparison is done lexicographically first by the author’s name, then by the book’s name (if necessary), then numerically by the book’s number (if necessary). For operators == and != comparison is done on all fields.
   * input/output operators (<<, >>): reads/prints in the following order: book number, author’s name, and book’s name.

Using the classes defined in question 1, write a program that manages the list of books in a libaray. **To do this, use the classes defined in question 1.**

At each iteration, the program displays a menu and reads in a request (a-e) from the user to be executed. The program terminates when an e (end of program) is read. The menu options are:

1. add a book to the library
2. remove a book from the library
3. search for a book in the library
4. print (in order defined by relational operators above) all the books in the library
5. exit application

During the program execution, the following output messages only should be used:

enter a-e:

enter a book

exist (result of search if book is in the system)

not exist (result of search or delete if book is not in the system)

error (on illegal input request)

Here is an example for the program output:

enter a-e:  
a  
enter a book:  
2 b b  
enter a-e:  
a  
enter a book:  
5 e e  
enter a-e:  
a  
enter a book:  
1 a a  
enter a-e:  
a  
enter a book:  
4 d d  
enter a-e:  
a  
enter a book:  
7 g g  
enter a-e:  
a  
enter a book:  
3 c c  
enter a-e:  
b  
enter a book:  
5 e e  
enter a-e:  
d  
1 a a  
2 b b  
3 c c  
4 d d  
7 g g  
enter a-e:  
e

Attachments for Question 1

**Tree Class**

#pragma once

#include <iostream>

using namespace std;

//-----------------------------------

// class Tree (Binary Trees)

// process nodes in Pre/In/Post order

//-----------------------------------

template <class T>

class Tree

{

protected:

//--------------------------------------------------------

// inner class Node

// a single Node from a binary tree

//--------------------------------------------------------

class Node

{

public:

Node\* left;

Node\* right;

T value;

Node(T val) : value(val), left(NULL), right(NULL) {}

Node(T val, Node\* l, Node\* r): value(val), left(l), right(r) {}

}; //end of Node class

//data member of tree

Node\* root;

public:

Tree() { root = NULL; } // initialize tree

~Tree();

int isEmpty() const;

void clear() { clear(root); root = NULL; }

void preOrder() { preOrder(root); }

void inOrder() { inOrder(root); }

void postOrder() { postOrder(root); }

virtual void process(T val) { cout << val << " "; }

virtual void add(T val) = 0;

virtual bool search(T val) = 0;

virtual void remove(T val) = 0;

private:

//private function for not give acsses to user

void clear(Node\* current);

void preOrder(Node\* current);

void inOrder(Node\* current);

void postOrder(Node\* current);

};

//----------------------------------------------------------

// class Tree implementation

//----------------------------------------------------------

template <class T>

Tree<T>::~Tree() // deallocate tree

{

if (root != NULL)

clear(root);

}

template <class T>

void Tree<T>::clear(Node\* current)

{

if (current)

{ // Release memory associated with children

if (current->left)

clear(current->left);

if (current->right)

clear(current->right);

delete current;

}

}

template <class T>

int Tree<T>::isEmpty() const

{

return root == NULL;

}

// preOrder processing of tree rooted at current

template <class T>

void Tree<T>::preOrder(Node\* current)

{ // visit Node, left child, right child

if (current)

{ // process current Node

process(current->value);

// then visit children

preOrder(current->left);

preOrder(current->right);

}

}

// inOrder processing of tree rooted at current

template <class T>

void Tree<T>::inOrder(Node\* current)

{ // visit left child, Node, right child

if (current)

{

inOrder(current->left);

process(current->value);

inOrder(current->right);

}

}

// postOrder processing of tree rooted at current

template <class T>

void Tree<T>::postOrder(Node\* current)

{ // visit left child, right child, node

if (current)

{

postOrder(current->left);

postOrder(current->right);

process(current->value);

}

}

**SearchTree class**

#pragma once

#include "Tree.h"

#include <iostream>

using namespace std;

template<class T>

class SearchTree : public Tree<T>

{

public:

void add(T value);

bool search(T value)

{

return search(Tree<T>::root, value);

}

}

private:

void add(typename Tree<T>::Node\* current, T val);

bool search(typename Tree<T>::Node\* current, T val);

};

template <class T>

void SearchTree<T>::add(T val)

{

// add value to binary search tree

if (!Tree<T>::root)

{

Tree<T>::root = new typename Tree<T>::Node(val);

return;

}

add(Tree<T>::root, val);

}

template <class T>

void SearchTree<T>::add(typename Tree<T>::Node\* current, T val)

{

if (current->value < val)

if (!current->right)

{

current->right = new typename Tree<T>::Node(val);

return;

}

else add(current->right, val);

else

if (!current->left)

{

current->left = new typename Tree<T>::Node(val);

return;

}

else add(current->left, val);

}

template <class T>

bool SearchTree<T>::search(typename Tree<T>::Node\* current, T val)

{

// see if argument value occurs in tree

if (!current)

return false; // not found

if (current->value == val)

return true;

if (current->value < val)

return search(current->right, val);

else

return search(current->left, val);

}

**Queue class**

#pragma once

#include <iostream>

using namespace std;

class Queue

{

public:

virtual ~Queue() {};

virtual void clear() = 0;

virtual void enqueue(int value) = 0;

virtual int dequeue() = 0;

virtual int front() = 0;

virtual bool isEmpty() const = 0;

};

**QueueVector class**

בהצלחה!

#pragma once

#include "Queue.h"

class QueueVector : public Queue

{

public:

QueueVector(int max);

//QueueVector(const QueueVector&);

void clear() override;

int dequeue() override;

void enqueue(int value) override;

int front() override;

bool isEmpty() const override;

private:

int\* data;

int capacity;

int nextSlot;

int firstUse;

};

QueueVector::QueueVector(int size) {

capacity = size + 1;

data = new int[capacity];

clear();

}

void QueueVector::clear() {

nextSlot = 0;

firstUse = 0;

}

int QueueVector::dequeue()

{

if (isEmpty()) throw "Queue is empty\n";

int dataloc = firstUse;

++firstUse %= capacity;

return data[dataloc];

}

void QueueVector::enqueue(int val) {

if ((nextSlot + 1) % capacity == firstUse)

throw "the Queue is full\n";

data[nextSlot] = val;

++nextSlot %= capacity;

}

int QueueVector::front() {

if (isEmpty())

throw "Queue is empty\n";

return data[firstUse];

}

bool QueueVector::isEmpty() const {

return nextSlot == firstUse;

}