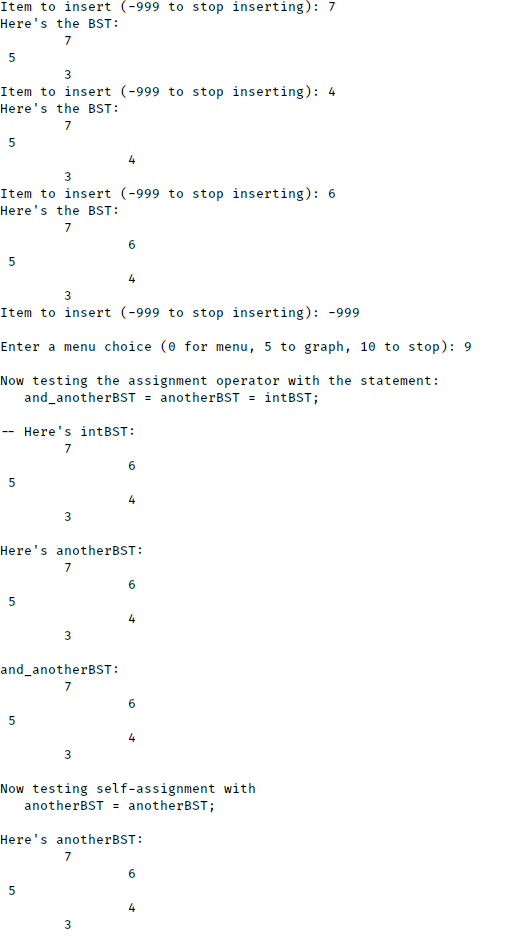
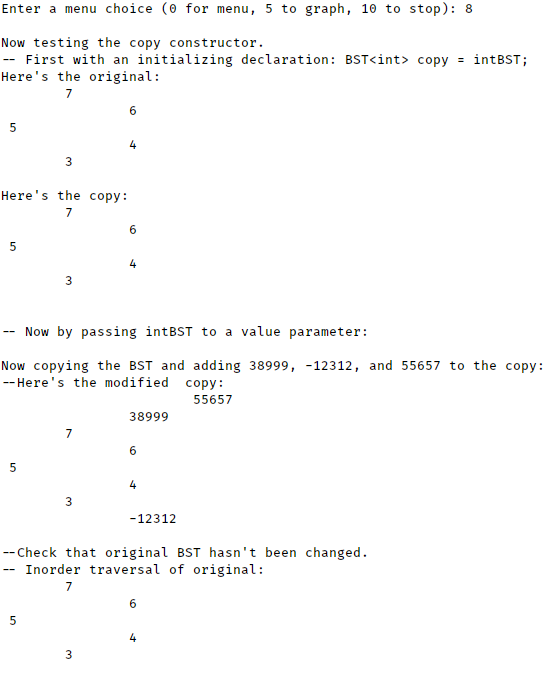
Part 1:





/\*--- treetester.cpp -------------------------------------------------

Program for testing class template BST.

Written by: Larry R. Nyhoff

Written for: Lab Manual for ADTs, Data Structures, and Problem

Solving with C++, 2E

Lab #10.1

Add your name here and other info requested by your instructor.

---------------------------------------------------------------------\*/

#include <iostream>

using namespace std;

#include "BST.h"

//\*---- PART 3 ----

// makeCopy() is a function with a

void **makeCopy**(BST<int> aBST) // BST value parameter

{ // to test the copy constructor

cout << "\nNow copying the BST and adding 38999,"

" -12312, and 55657 to the copy:\n";

aBST.insert(38999);

aBST.insert(-12312);

aBST.insert(55657);

cout << "--Here's the modified copy: \n";

aBST.graph(cout);

}

//---- END PART 3 ----\*/

int **main**() {

const char MENU[] =

"MENU CHOICES\n"

"0. Display the menu\n"

"1. Check if BST is empty\n"

"2. Insert some elements into the BST\n"

"3. Search for an element\n"

"4. Delete some elements from the BST\n"

"5. Graphical representation of BST (sideways)\n"

"6. Inorder traversal\n"

"61. Preorder traversal\n"

"62. Postorder traversal\n"

"7. Check destructor\n"

"8. Check copy constructor\n"

"9. Check assignment operator\n"

"10. Quit the program\n";

// Testing Constructor and empty()

BST<int> intBST; // test the class constructor

cout << "Constructing empty BST\n";

cout << "BST " << (intBST.empty() ? "is" : "is not") << " empty\n";

// Test Other Operations

cout << MENU << endl;

int choice;

do {

cout << "\nEnter a menu choice (0 for menu, 5 to graph, 10 to stop): ";

cin >> choice;

switch (choice) {

case 0: // Display menu

cout << MENU << endl;

break;

case 1: // Checking empty

cout << "BST " << (intBST.empty() ? "is" : "is not") << " empty\n";

break;

case 2: // Insert elements

cout << "\nNow insert a bunch of integers into the BST."

"\nTry items not in the BST and some that are in it:\n";

int number;

for (;;) {

cout << "Item to insert (-999 to stop inserting): ";

cin >> number;

if (number == -999)

break;

intBST.insert(number);

cout << "Here's the BST:\n";

intBST.graph(cout);

}

break;

case 3: // Searching)

cout << "\n\nNow testing the search() operation."

"\nTry both items in the BST and some not in it:\n";

for (;;) {

cout << "Item to find (-999 to stop searching): ";

cin >> number;

if (number == -999)

break;

cout << (intBST.search(number) ? "Found" : "Not found") << endl;

}

break;

case 4: // Deleting elements

cout << "\nNow testing the remove() operation."

"\nTry both items in the BST and some not in it:\n";

for (;;) {

cout << "Item to delete (-999 to stop deleting): ";

cin >> number;

if (number == -999)

break;

intBST.remove(number);

cout << "Here's the BST:\n";

intBST.graph(cout);

}

break;

case 5: // Graphical representation

cout << "Here's the BST (sidewise):\n";

intBST.graph(cout);

break;

case 6: // Inorder traversal

cout << "\nInorder Traversal of BST: \n";

intBST.inorder(cout);

cout << endl;

break;

default:

cerr << "BAD CHOICE -- TRY AGAIN\n";

break;

case 10: // quit menu

break;

/\* ---- PART 1 ----

case 61: // Test preorder traversal

cout << "\nPreorder Traversal of BST: \n";

intBST.preorder(cout);

break;

---- END PART 1 ----\*/

/\* ---- PART 1 ----

case 62: // Test postorder traversal

cout << "\nPostorder Traversal of BST: \n";

intBST.postorder(cout);

break;

---- END PART 2 ----\*/

//\* ---- PART 2 ----

case 7: // Testing the Destructor

cout << "\nNow testing the destructor. Remember to add an\n"

"output statement to your destructor to indicate \n"

"when it is called.\n";

{

BST<int> doomedBST;

doomedBST.insert(6);

doomedBST.insert(9);

doomedBST.insert(5);

doomedBST.insert(1);

doomedBST.insert(3);

doomedBST.insert(7);

cout << "\nHere's a BST:\n";

doomedBST.graph(cout);

cout << "\n\nLifetime of this BST is over -- now destroy it.\n";

}

break;

//---- END PART 2 ----\*/

//\* ---- PART 3 ----

case 8: // Testing the Copy Constructor

{

cout << "\nNow testing the copy constructor.\n";

cout << "-- First with an initializing declaration: "

"BST<int> copy = intBST;\n";

BST<int> copy = intBST;

cout << "Here's the original:\n";

intBST.graph(cout);

cout << "\nHere's the copy:\n";

copy.graph(cout);

}

cout << "\n\n-- Now by passing intBST to a value parameter:\n";

makeCopy(intBST);

cout << "\n--Check that original BST hasn't been changed.\n"

"-- Inorder traversal of original:\n";

intBST.graph(cout);

cout << endl;

break;

// ---- END PART 3 ----\*/

//\* ---- PART 4 ----

case 9: // Testing the Assignment Operator

cout << "\nNow testing the assignment operator with the "

"statement:\n and\_anotherBST = anotherBST = intBST;\n";

BST<int> anotherBST, and\_anotherBST;

;

and\_anotherBST = anotherBST = intBST;

cout << "\n-- Here's intBST:\n";

intBST.graph(cout);

cout << "\nHere's anotherBST:\n";

anotherBST.graph(cout);

cout << "\nand\_anotherBST:\n";

and\_anotherBST.graph(cout);

cout << "\nNow testing self-assignment with"

"\n anotherBST = anotherBST;\n";

anotherBST = anotherBST;

cout << "\nHere's anotherBST:\n";

anotherBST.graph(cout);

cout << endl;

break;

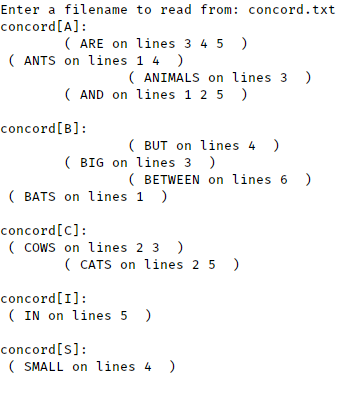
//---- END PART 4 ----\*/

} // switch

} while (choice != 10);

} // end main()

Part 2:



/// \file Project-10.1.cpp

/// \brief Concordance creator

/// \author Johnathan Lee for CSCI 1107, Lab 14

/// \date Due 05/04/18

#include <fstream>

#include <iostream>

#include <sstream>

#include <string>

#include "BST.h"

#include "Token.h"

using namespace std;

typedef BST<Token>

AlphabetBST[26]; ///>! An array of BSTs, 1 member per letter.

/// \brief Gets a file based on user input.

/// \param file The ifstream to open.

/// \pre User must enter a valid filename/path in cin.

/// \post file is initialized/opened to the specified file.

void **getFile**(ifstream& file);

/// \brief Tells whether ch is one of [A-Z] or [a-z]

/// (uppercase or lowercase alphabetic)

bool **isAlphabetic**(char ch);

/// \brief Generates a concordance (an alphabetical listing of all distinct

/// words in a piece of text.)

/// \param storage An array of 26 BSTs which store tokens in which to store the

/// concordance.

/// \param inputFile The file to read from.

/// \post storage contains a concordance, indexed by the first letter of each

/// word.

/// \note All non-alphabetic (see isAlphabetic) characters will be removed. All

/// alphabetic characters will be made uppercase. (See std::toupper)

void **generateConcordance**(AlphabetBST& storage, ifstream& inputFile);

/// \brief Displays a concordance.

/// \param concord The pre-generated concordance to display.

/// \post concord has been displayed to cout in the following format:

/// concord[<UPPERCASE CHARACTER INDEX>]:

/// <CONCORDANCE FOR THAT LETTER GRAPHED>

/// ...

/// \note See BST::graph and Token::display for more.

void **displayConcordance**(AlphabetBST& concord);

/// \brief Strips any non-alphabetic characters from str and makes it all

/// UPPERCASE.

/// \param str The string to operate on

/// \post Examples of transformations: "hello"->"HELLO", "h3llo"->"HLLO"

void **stripNonAlpha**(string& str);

//\*/

int **main**() {

AlphabetBST concord;

ifstream inputFile;

getFile(*inputFile*);

generateConcordance(*concord*, *inputFile*);

displayConcordance(*concord*);

return 0;

}

//\*/

void **getFile**(ifstream& file) {

// Default name for rapid testing: Just comment out the 2 lines after it.

string fileName = "concord.txt";

//\*

cout << "Enter a filename to read from: ";

cin >> fileName;

//\*/

file.open(fileName);

if (!file.good()) {

cout << "Error opening " << fileName << " , terminating...\n";

exit(1);

}

}

bool **isAlphabetic**(char ch) {

return ((ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z'));

}

void **generateConcordance**(AlphabetBST& storage, ifstream& inputFile) {

string lineBuf;

unsigned lineNum = 1;

// For each line

while (getline(inputFile, *lineBuf*)) {

stringstream line;

line << lineBuf;

// For each word on the line

string wordBuf;

while (line >> wordBuf) {

stripNonAlpha(*wordBuf*);

// Note for future: Don't do this and then try to test with a string

// like "2". Just don't. Especially before you implement the non-alpha

// character stripper.

storage[wordBuf[0] - 'A'].insert(Token(wordBuf, lineNum));

}

lineNum++;

}

}

void **displayConcordance**(AlphabetBST& concord) {

// Automatically displays in ascending order since that's how they were

// inserted.

for (char i = 'A'; i < 'Z'; i++) {

if (!concord[i - 'A'].empty())

cout << "concord[" << i << "]:\n" << concord[i - 'A'] << endl;

}

}

void **stripNonAlpha**(string& str) {

for (unsigned i = 0; i < str.size(); i++) {

char ch = str[i];

if (isAlphabetic(ch)) {

str[i] = toupper(ch);

} else {

// If this character isn't [A-Z] or [a-z]

str.erase(i, 1); // Remove it

i--; // And make sure we're still going over all the stuff

// properly (Since erase will move us).

}

}

}

/\*--- BST.h ----------------------------------------------------------------

This header file contains the class template BST.

Basic operations:

Constructor: Constructs an empty BST

empty: Checks if a BST is empty

search: Search a BST for an item

insert: Inserts a value into a BST

remove: Removes a value from a BST

graph: Output a grapical representation of a BST

inorder: Inorder traversal of a BST -- output the data values

Private utility helper operations:

search2: Used by delete

inorderAux: Used by inorder

graphAux: Used by graph

Operations tp be added:

preorder and postorder traversals

destructor

copy constructor

assignment operator

level-by-level traversal (in project)

level finder (in project)

Written by: Larry R. Nyhoff

Written for: Lab Manual for ADTs, Data Structures, and Problem

Solving with C++, 2E

Lab #10.1

Johnathan Lee CSCI 1107

Lab 14 Due 05/04/18

---------------------------------------------------------------------------\*/

#include <iostream>

#include <new>

#include <vector>

#include "Token.h"

#include "circq.h"

#ifndef BINARY\_SEARCH\_TREE

#define BINARY\_SEARCH\_TREE

template <typename T>

class BST {

public:

/\*\*\*\*\* Function Members \*\*\*\*\*/

**BST**();

/\*------------------------------------------------------------------------

Construct a BST object.

Precondition: None.

Postcondition: An empty BST has been constructed.

-----------------------------------------------------------------------\*/

bool **empty**() const;

/\*------------------------------------------------------------------------

Check if BST is empty.

Precondition: None.

Postcondition: Returns true if BST is empty and false otherwise.

-----------------------------------------------------------------------\*/

bool **search**(const T& item) const;

/\*------------------------------------------------------------------------

Search the BST for item.

Precondition: None.

Postcondition: Returns true if item found, and false otherwise.

-----------------------------------------------------------------------\*/

void **insert**(const T& item);

/\*------------------------------------------------------------------------

Insert item into BST.

Precondition: None.

Postcondition: BST has been modified with item inserted at proper

position to maintain BST property.

------------------------------------------------------------------------\*/

void remove(const T& item);

/\*------------------------------------------------------------------------

Remove item from BST.

Precondition: None.

Postcondition: BST has been modified with item removed (if present);

BST property is maintained.

Note: remove uses private auxiliary function search2() to locate

the node containing item and its parent.

------------------------------------------------------------------------\*/

void **graph**(ostream& out) const;

/\*------------------------------------------------------------------------

Graphic output of BST.

Precondition: ostream out is open.

Postcondition: Graphical representation of BST has been output to out.

Note: graph() uses private auxiliary function graphAux().

------------------------------------------------------------------------\*/

void **inorder**(ostream& out) const;

/\*------------------------------------------------------------------------

Inorder traversal of BST.

Precondition: ostream out is open.

Postcondition: BST has been inorder traversed and values in nodes

have been output to out.

Note: inorder uses private auxiliary function inorderAux().

------------------------------------------------------------------------\*/

//--- ADD PROTOTYPES OF preorder() AND postorder() HERE

//--- ADD PROTOTYPE OF DESTRUCTOR HERE

~**BST**();

/\*------------------------------------------------------------------------

\* Destructor

\*

\* Preconditions: None.

\* Postconditions: Any nodes have had selfDestruct called on them, destroying

\* them.

\*/

//--- ADD PROTOTYPE OF COPY CONSTRUCTOR HERE

**BST**(const BST& bst);

//--- ADD PROTOTYPE OF ASSIGNMENT OPERATOR HERE

BST& *operator*=(const BST& rhs);

/\*------------------------------------------------------------------------

\* Assignment operator

\*

\* Preconditions: None.

\* Postconditions: This BST is now an exact clone of rhs.

\*/

//--- ADD PROTOTYPE OF LEVEL-BY-LEVEL TRAVERSAL HERE

void **levelByLevel**() const;

/\*------------------------------------------------------------------------

\* levelByLevel

\* Goes through the BST and prints out node contents 1 level at a time

\*

\* Preconditions: None.

\* Postconditions: Printed in the following format:

\* Level <LEVEL>:

\* <COUT ALL NODES IN THIS LEVEL>

\* Level <LEVEL+1>:

\* ...

\*/

//--- ADD PROTOTYPE OF LEVEL FINDER HERE

unsigned **level**(const T& val);

/\*------------------------------------------------------------------------

\* level

\* Gets the level of an element.

\*

\* Preconditions: None.

\* Postconditions: If val is somewhere in the BST, returns the level it was

\* found on. If it wasn't, the maximum value for unsigned is returned as a

\* sentinel.

\*/

private:

/\*\*\*\*\* Node class \*\*\*\*\*/

class BinNode {

public:

T data;

BinNode\* left;

BinNode\* right;

// BinNode constructors

// Default -- data part is default DataType value; both links are null.

**BinNode**() : left(NULL), right(NULL) {

}

// Explicit Value -- data part contains item; both links are null.

**BinNode**(T item) : data(item), left(NULL), right(NULL) {

}

**BinNode**(const BinNode& other) {

data = other.data;

left = other.left;

right = other.right;

}

void **selfDestruct**();

/\*--------------------------------------------------------------------------

\* selfDestruct

\* Traverses this node's subtree in LRP order, freeing all nodes.

\* Precondition: This node exists.

\* Postcondition: This node is history.

\*/

void **copyTo**(BinNode\* node) const;

/\*------------------------------------------------------------------------

\* copyTo

\* Recursively copies this subtree to another subtree.

\* Pre: this node exists.

\* Post: This entire subtree was replicated at node

\*/

}; // end of class BinNode declaration

typedef BinNode\* BinNodePointer;

/\*\*\*\*\* Private Function Members \*\*\*\*\*/

void **search2**(const T& item, bool& found, BinNodePointer& locptr,

BinNodePointer& parent) const;

/\*------------------------------------------------------------------------

Locate a node containing item and its parent.

Precondition: None.

Postcondition: locptr points to node containing item or is null if

not found, and parent points to its parent.#include <iostream>

------------------------------------------------------------------------\*/

void **inorderAux**(ostream& out, BinNodePointer subtreePtr) const;

/\*------------------------------------------------------------------------

Inorder traversal auxiliary function.

Precondition: ostream out is open; subtreePtr points to a subtree

of this BST.

Postcondition: Subtree with root pointed to by subtreePtr has been

output to out.

------------------------------------------------------------------------\*/

void **graphAux**(ostream& out, int indent, BinNodePointer subtreeRoot) const;

/\*------------------------------------------------------------------------

Graph auxiliary function.

Precondition: ostream out is open; subtreePtr points to a subtree

of this BST.

Postcondition: Graphical representation of subtree with root pointed

to by subtreePtr has been output to out, indented indent spaces.

------------------------------------------------------------------------\*/

/\*\*\*\*\* Data Members \*\*\*\*\*/

BinNodePointer myRoot;

}; // end of class template declaration

//--- Definition of constructor

template <typename DataType>

inline BST<DataType>::BST() : myRoot(NULL) {

}

//--- Definition of empty()

template <typename DataType>

inline bool BST<DataType>::empty() const {

return myRoot == NULL;

}

//--- Definition of search()

template <typename DataType>

bool BST<DataType>::search(const DataType& item) const {

BST<DataType>::BinNodePointer locptr = myRoot;

bool found = false;

while (!found && locptr != NULL) {

if (item < locptr->data) // descend left

locptr = locptr->left;

else if (locptr->data < item) // descend right

locptr = locptr->right;

else // item found

found = true;

}

return found;

}

//--- Definition of insert()

template <typename DataType>

inline void BST<DataType>::insert(const DataType& item) {

BST<DataType>::BinNodePointer locptr = myRoot, // search pointer

parent = NULL; // pointer to parent of current node

bool found = false; // indicates if item already in BST

while (!found && locptr != NULL) {

parent = locptr;

if (item < locptr->data) // descend left

locptr = locptr->left;

else if (locptr->data < item) // descend right

locptr = locptr->right;

else // item found

found = true;

}

if (!found) { // construct node containing item

locptr = new (nothrow) BST<DataType>::BinNode(item);

if (locptr == NULL) {

cerr << "\*\*\* Out of memory -- terminating program \*\*\*\n";

exit(1);

}

if (parent == NULL) // empty tree

myRoot = locptr;

else if (item < parent->data) // insert to left of parent

parent->left = locptr;

else // insert to right of parent

parent->right = locptr;

} else

cout << "Item already in the tree\n";

}

//--- Definition of remove()

template <typename DataType>

void BST<DataType>::remove(const DataType& item) {

bool found; // signals if item is found

BST<DataType>::BinNodePointer x, // points to node to be deleted

parent; // " " parent of x and xSucc

search2(item, *found*, *x*, *parent*);

if (!found) {

cout << "Item not in the BST\n";

return;

}

// else

if (x->left != NULL && x->right != NULL) { // node has 2 children

// Find x's inorder successor and its parent

BST<DataType>::BinNodePointer xSucc = x->right;

parent = x;

while (xSucc->left != NULL) // descend left

{

parent = xSucc;

xSucc = xSucc->left;

}

// Move contents of xSucc to x and change x

// to point to successor, which will be removed.

x->data = xSucc->data;

x = xSucc;

} // end if node has 2 children

// Now proceed with case where node has 0 or 2 child

BST<DataType>::BinNodePointer subtree =

x->left; // pointer to a subtree of x

if (subtree == NULL)

subtree = x->right;

if (parent == NULL) // root being removed

myRoot = subtree;

else if (parent->left == x) // left child of parent

parent->left = subtree;

else // right child of parent

parent->right = subtree;

delete x;

}

//--- Definition of graph()

template <typename DataType>

inline void BST<DataType>::graph(ostream& out) const {

graphAux(*out*, 0, myRoot);

}

//--- Definition of search2()

template <typename DataType>

void BST<DataType>::search2(const DataType& item, bool& found,

BinNodePointer& locptr,

BinNodePointer& parent) const {

locptr = myRoot;

parent = NULL;

found = false;

while (!found && locptr != NULL) {

if (item < locptr->data) // descend left

{

parent = locptr;

locptr = locptr->left;

} else if (locptr->data < item) // descend right

{

parent = locptr;

locptr = locptr->right;

} else // item found

found = true;

}

}

//--- Definition of graphAux()

#include <iomanip>

template <typename DataType>

void BST<DataType>::graphAux(ostream& out, int indent,

BinNodePointer subtreeRoot) const {

if (subtreeRoot != NULL) {

graphAux(*out*, indent + 8, subtreeRoot->right);

out << setw(indent) << " " << subtreeRoot->data << endl;

graphAux(*out*, indent + 8, subtreeRoot->left);

}

}

//--- Definition of inorder()

template <typename DataType>

inline void BST<DataType>::inorder(ostream& out) const {

inorderAux(*out*, myRoot);

}

//--- Definition of inorderAux()

template <typename DataType>

void BST<DataType>::inorderAux(ostream& out, BinNodePointer subtreeRoot) const {

if (subtreeRoot != NULL) {

inorderAux(*out*, subtreeRoot->left); // L operation

out << subtreeRoot->data << " "; // V operation

inorderAux(*out*, subtreeRoot->right); // R operation

}

}

//--- PUT DEFINITIONS OF THE ADDED OPERATIONS HERE

template <typename T>

BST<T>::~**BST**() {

if (myRoot)

myRoot->selfDestruct();

myRoot = NULL;

}

template <typename T>

BST<T>::BST(const BST& bst) {

*this*->~BST();

if (bst.myRoot) {

myRoot = new BinNode();

bst.myRoot->copyTo(myRoot);

}

}

template <typename T>

BST<T>& BST<T>::operator=(const BST& rhs) {

if (this != &rhs) {

*this*->~BST();

if (rhs.myRoot) {

myRoot = new BinNode();

rhs.myRoot->copyTo(myRoot);

}

}

return \*this;

}

template <typename T>

void BST<T>::levelByLevel() const {

bool endReached = false;

unsigned curLevel = 0;

CircularQ<BinNode\*> curLevelPtrs, curLevelBuff;

curLevelPtrs.enqueue(myRoot);

while (!endReached && !curLevelPtrs.empty()) {

endReached = true;

cout << "\nLevel " << curLevel++ << ":\n";

while (!curLevelPtrs.empty()) {

BinNode\* cur = curLevelPtrs.front();

if (cur) {

endReached = false;

cout << cur->data;

// Make sure we don't enqueue useless data.

if (cur->left)

curLevelBuff.enqueue(cur->left);

if (cur->right)

curLevelBuff.enqueue(cur->right);

}

curLevelPtrs.dequeue();

}

curLevelPtrs = curLevelBuff; // Swap them.

curLevelBuff.~CircularQ(); // Clear out the buffer for next level.

}

}

template <typename T>

unsigned BST<T>::level(const T& val) {

unsigned finalLevel = -1;

BinNode\* cur = myRoot;

while (cur) {

finalLevel++;

if (cur->data == val)

return finalLevel;

else if (val > cur->data)

cur = cur->right;

else if (val < cur->data)

cur = cur->left;

}

}

// For destructor

template <typename T>

void BST<T>::BinNode::selfDestruct() {

if (left)

left->selfDestruct();

if (right)

right->selfDestruct();

left = right = NULL;

delete this;

}

template <typename T>

void BST<T>::BinNode::copyTo(BST::BinNode\* node) const {

node->data = *this*->data;

if (*this*->left) {

if (node->left == NULL)

node->left = new BinNode(left->data);

left->copyTo(node->left);

}

if (*this*->right) {

if (node->right == NULL)

node->right = new BinNode(right->data);

right->copyTo(node->right);

}

}

// Specialized for Project-10.1

template <typename T>

ostream& *operator*<<(ostream& out, const BST<T>& bst) {

bst.graph(out);

return out;

}

#endif

/// \file circq.h

/// \brief Declares and defines CircularQ data type and associated operators.

///

/// \author Johnathan Lee for CSCI 1107, Lab 10

/// \date Due 04/03/18

///

/// \note Modified for Lab 14 by Johnathan Lee:

/// - Removed circq.cpp (All definitions now in this file)

/// - Turned to template version.

/// - Replaced QueueElement with typename T.

/// - Added enqueueIfUnique

/// - Changed header comment style to fit with the rest of the docs.

#ifndef CIRCQ\_H

#define CIRCQ\_H

#include <iostream>

using namespace std;

/// \class CircularQ

/// \brief A Queue implemented with a circular linked list.

/// \note Uses sizeof(QueueElement) + sizeof(POINTER) per element.

template <typename T>

class CircularQ {

public:

/// \brief Default (Empty) constructor

/// \param None

/// \post \*this is initialized with absolutely no elements.

**CircularQ**();

/// \brief Copy constructor:

/// \param orig The CircularQ to copy elements from

/// \post All elements from orig are now ALSO contained in \*this.

/// \note This does NOT transfer ownership, it is a deep copy, orig is

/// unchanged.

**CircularQ**(const CircularQ& orig);

/// \brief Destructor

/// \post All elements in this queue are deleted

~**CircularQ**();

/// \brief Assignment

/// \param rhs The queue to copy elements from.

/// \post All elements are deep copied to \*this. See copy constructor.

const CircularQ& *operator*=(const CircularQ& rhs);

/// \brief Check if the queue is empty.

/// \return Whether there are no elements in the queue

bool **empty**() const;

/// \brief Check if the queue contains a certain element.

/// \returns Whether the queue contains the val.

/// \note type T MUST have operator== overloaded.

bool **contains**(const T& val) const;

/// \brief Add a value to the end of the queue.

/// \param value The value to add into the queue.

/// \post value is now at the end of the queue.

void **enqueue**(const T& value);

/// \brief Adds a value to the end of the queue ONLY IF it's not already in

/// the queue.

/// \param value The value to add.

/// \post If value isn't in the queue, it has been enqueued. If it is,

/// nothing has changed.

/// \note See this->contains for type restrictions.

void **enqueueIfUnique**(const T& value);

/// \brief Outputs \*this

/// \param out The stream to write to

/// \post The contents of this list are written to out, space delimited.

void **display**(ostream& out) const;

/// \brief Get the front of the queue.

/// \returns The first element in the queue. (I.E been there the longest).

T **front**() const;

/// \brief Remove the element at the front of the queue.

/// \post Current this->front() is no longer in memory.

void **dequeue**();

private:

/// \class Node

/// \brief Internal storage for data elements. Not relevent outside

/// CircularQueue class internals.

class Node {

public:

/// \brief Constructor

/// \param value The data value to assign to data

/// \param link The Node \*this should link to.

**Node**(T value, Node\* link = NULL) : data(value), next(link) {

}

T data; ///>! Our data element

Node\* next; ///>! The next link in the chain

};

typedef Node\* NodePointer;

NodePointer myBack; ///>! Our view into the list. Use myBack->next for front

};

/// \brief Output stream operator for CircularQ

/// \param out The stream to write to.

/// \param q The queue to display

/// \post Writes all elements in q to out. See CircularQ::display for more.

template <typename T>

ostream& *operator*<<(ostream& out, const CircularQ<T>& q);

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

// CLASS FUNCTION DEFS

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

template <typename T>

CircularQ<T>::CircularQ() {

myBack = NULL;

}

// Since it's just 2 statements, should really be inlined...

template <typename T>

CircularQ<T>::CircularQ(const CircularQ<T>& orig) {

myBack = NULL;

// Since it's the same code and the (this == &rhs) still works...

\*this = orig;

}

template <typename T>

CircularQ<T>::~**CircularQ**() {

if (myBack) {

Node\* cur = myBack->next;

myBack->next = NULL;

while (cur != myBack) {

Node\* old = cur;

cur = cur->next;

delete old;

}

delete myBack; // Putting out here to avoid problems with size() == 1.

}

myBack = NULL; // Accomodate Microsoft

}

template <typename T>

const CircularQ<T>& CircularQ<T>::operator=(const CircularQ<T>& rhs) {

if (this != &rhs) {

*this*->~CircularQ(); // Delete old list

if (rhs.myBack) {

Node\* curRhs = rhs.myBack->next;

// Use our methods - think deeply of simple things.

// (I'm gonna have nightmares about that phrase)

do {

enqueue(curRhs->data);

curRhs = curRhs->next;

} while (curRhs != rhs.myBack->next);

}

}

return \*this;

}

template <typename T>

bool CircularQ<T>::empty() const {

return myBack == NULL; // If myBack is null we have no nodes, so empty.

}

template <typename T>

bool CircularQ<T>::contains(const T& value) const {

bool found = false;

Node\* cur = myBack;

if (myBack) {

// Check myBack first, otherwise the cur!=myBack will skip it.

if (myBack->data == value)

found = cur->data == value;

else // 2+ elements

// While we aren't NULL and we haven't looped and we haven't found

// it

while ((cur = cur->next) && (cur != myBack) && !found)

if (cur->data == value)

found = true;

}

}

template <typename T>

void CircularQ<T>::enqueue(const T& value) {

if (myBack) { // !empty

Node\* newNode = new Node(value, myBack->next);

myBack->next = newNode;

myBack = newNode;

} else {

myBack = new Node(value, myBack);

// Because assigning it in the constructor would assign to NULL.

*//* *You* *don't* *wanna* *know* *how* *long* *that* *one* *took* *me.*

myBack->next = myBack;

}

}

template <typename T>

void CircularQ<T>::enqueueIfUnique(const T& value) {

if (!contains(value))

enqueue(value); // Avoid duplicating code.

}

template <typename T>

void CircularQ<T>::display(ostream& out) const {

if (myBack) {

Node\* cur = myBack->next;

do {

out << cur->data << " ";

cur = cur->next;

} while (cur != myBack->next);

} else {

// To help distinguish between no output and outputting blank queue.

/\*//\*/ out << "EMPTY\_CIRCULAR\_QUEUE ";

}

}

template <typename T>

T CircularQ<T>::front() const {

return myBack->next->data;

}

template <typename T>

void CircularQ<T>::dequeue() {

if (myBack->next == myBack) {

delete myBack;

myBack = NULL;

} else {

Node\* oldFront = myBack->next;

myBack->next = oldFront->next;

delete oldFront;

}

}

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

// HELPER FUNCTION DEFS

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

template <typename T>

ostream& *operator*<<(ostream& out, const CircularQ<T>& q) {

q.display(out);

return out;

}

#endif

/// \file Token.cpp

/// \brief Defines a Token class

/// \author Johnathan Lee

/// \date Due 05/04/18

#ifndef TOKEN\_H

#define TOKEN\_H

#include <string>

#include "circq.h"

/// \class Token

/// \brief A token class. Holds a string and a list of all lines it is seen on.

class Token {

public:

/// \brief Explicit constructor

/// \param tok The token to use

/// \param initialLine The line that will be first in lineNumbers.

/// \note Do not use as implicit constructor. This is just to make the

/// compiler happy.

**Token**(const string& tok = "", unsigned initialLine = -1);

/// \brief Copy constructor

/// \param tok The Token to copy from.

**Token**(const Token& tok);

/// \brief Assignment operator for token

/// \param rhs The token to assign from.

/// \returns a reference to \*this;

Token& *operator*=(const Token& rhs);

/// \brief Print \*this to an ostream

/// \param out The ostream to write to.

/// \post \*this has been written to out in the following format:

/// (##TOKEN## on lines ##EACH LINE NUMBER##)

/// \note See CircularQ->display for each line number.

void **display**(ostream& out) const;

/// \brief Concatenate the line numbers in this Token with those of another.

/// \param rhs The token to take numbers from.

/// \returns A token containing all the line numbers from \*this, followed by

/// all the line numbers from rhs. (Skipping duplicate lines)

/// \note Pass by value to avoid making another temp variable.

Token *operator*+(Token rhs);

/// \brief Less than operator

/// \param rhs Other token to compare

/// \returns Whether this->token < rhs.token.

bool *operator*<(const Token& rhs) const;

/// \brief Greater than operator

/// \param rhs Other token to compare

/// \returns Whether this->token > rhs.token.

bool *operator*>(const Token& rhs) const;

/// \brief Equality operator

/// \param rhs Other token to compare

/// \returns Whether this->token == rhs.token.

bool *operator*==(const Token& rhs) const;

/// \brief Gets the token from this instance.

/// \returns A const ref to the token.

/// \note const ref to avoid copying large strings.

const string& **getToken**() const;

private:

string token; ///>! The token we have.

CircularQ<unsigned> lineNumbers; ///>! All lines this token was found on.

};

/// \brief Stream insertion operator for Token

/// \param out The stream to write to

/// \param t The token to display

/// \post t.display has been called on out.

ostream& *operator*<<(ostream& out, const Token& t);

#endif

/// \file Token.cpp

/// \brief Implementation for Token.h

/// \author Johnathan Lee

/// \date Due 05/04/18

#include "Token.h"

#include "BST.h"

#include <cassert>

Token::Token(const string& tok, unsigned initialLine) {

token = tok;

lineNumbers.enqueue(initialLine);

}

Token::Token(const Token& tok) {

token = tok.token;

lineNumbers = tok.lineNumbers;

}

Token& Token::operator=(const Token& rhs) {

token = rhs.token;

lineNumbers = rhs.lineNumbers;

return \*this;

}

void Token::display(ostream& out) const {

out << "( " << token << " on lines " << lineNumbers << " )";

}

Token Token::operator+(Token rhs) {

Token newToken = \*this;

while (!rhs.lineNumbers.empty()) {

newToken.lineNumbers.enqueueIfUnique(rhs.lineNumbers.front());

rhs.lineNumbers.dequeue();

}

return newToken;

}

bool Token::operator<(const Token& rhs) const {

return *this*->token < rhs.token;

}

bool Token::operator>(const Token& rhs) const {

return *this*->token > rhs.token;

}

bool Token::operator==(const Token& rhs) const {

return token == rhs.token;

}

const string& Token::getToken() const {

return token;

}

ostream& *operator*<<(ostream& out, const Token& t) {

t.display(*out*);

return out;

}

template <>

void BST<Token>::insert(const Token& word) {

BinNodePointer locptr = myRoot, // search pointer

parent = NULL; // pointer to parent of current node

bool found = false; // indicates if item already in BST

while (!found && locptr != NULL) {

parent = locptr;

if (word < locptr->data) // descend left

locptr = locptr->left;

else if (locptr->data < word) // descend right

locptr = locptr->right;

else // item found

found = true;

}

if (!found) { // construct node containing item

locptr = new (nothrow) BinNode(word);

if (locptr == NULL) {

cerr << "\*\*\* Out of memory -- terminating program \*\*\*\n";

exit(1);

}

if (parent == NULL) // empty tree

myRoot = locptr;

else if (word < parent->data) // insert to left of parent

parent->left = locptr;

else // insert to right of parent

parent->right = locptr;

} else

locptr->data = locptr->data + word;

}