University of Nevada, Reno

CS 302 — Data Structures

Assignment #5

Students:
Joshua Gleason
Josiah Humphrey

Instructor:
Dr. George Bebis

May 4, 2010

Contents

1 Introduction

In this assignment, we looked at 3 different data structures that are very important for computer science. We examined the usefulness and implementation of the heap, the priority queue, and the binary search tree. The program that we designed to test these data structures were fairly trivial, but the challenge was implementing the assignment using the data structures that we were supposed to implement.

The first part of the assignment was to make a binary search tree that help users and their passwords. The binary search tree was built using the name of the user as the node name. This allowed us to quickly search the list of users. This is efficient when there are many users and when the program user needs to find a specific user quickly. The implementation of the binary search tree we implemented was a template class that could handle any data structure. For this assignment, we made our own data structure that included the name and the pass as string objects. This allowed us to do many useful things such as compare. This was useful because it allowed the quick comparison of the string objects when building the binary search tree.

In the second part of the assignment, we were asked to implement an updated priority queue that allowed the users to remove a specific item from the tree and to update the elements within the tree. This was done through inheritance. A class called "pqueue.h" was defined that was a basic priority queue. The new updated queue, U_PQType, was derived from this class in order to include the two new functions. This derived class used the heap property of the priority queue to implement the assignments challenge. We were asked to read in numbers from a file and place them into a heap.

2 Use of Code

In order to use this program, you will need data files that correspond to how we implemented the program. For assignment one, there needs to be a data file that has a user name and password that are separated by white space. It should be in the format of "user1 pass1
n user2 pass2" where
n is an actual new line. We have provided a sample file for your convenience. Once you load up the program, it asks you to enter a file name with the user info in it. It will do this until you have correctly entered in a file name that is readable and in the correct format. Once this happens, you are brought to a menu in which you can do the operations that are required. It should be fairly simple from here.

To use part 2 of the assignment, you will need to use a data file that has integers in it separated by white space. When the program begins, it asks for a file name until it is given a suitable file name is given. Once inside the main menu, you will have the options of what was assigned for this challenge. Once here, it should be fairly straight forward as to what each menu item does.

3 Functions

3.1 binaryTree.h

BINARYTREE

```
binaryTree();
```

Purpose

Constructor, simply initializes root to NULL

Input

N/A

Output

N/A

Assumptions

N/A

BINARYTREE

```
~binaryTree();
```

Purpose

Free all the memory in the tree

Input

N/A

Output

N/A

Assumptions

N/A

MAKEEMPTY

```
void makeEmpty();
```

Purpose

Recursively remove all the nodes below the node.

Input

A node.

Output

N/A

Assumptions

Node is either NULL or points to an allocated node type.

isEmpty

```
bool isEmpty() const;
```

Purpose

Test if tree is empty.

Input

N/A

Output

Returns true if tree is empty, otherwise returns false.

Assumptions

N/A

numberOfNodes

```
int numberOfNodes() const;
```

Purpose

Count the number of nodes in the tree recursively.

Input

N/A

Output

Return the number of nodes in the tree.

Assumptions

N/A

retrieveItem

```
bool retrieveItem(iType&);
```

Purpose

Attempts to retrieve an item from the tree, return false if item is not found in the tree.

Input

An item that has the same "key" value as the item in question.

Output

Return true if item is in tree, false otherwise.

Assumptions

N/A

insertItem

```
void insertItem(iType);
```

Purpose

Create a new node and insert it into the tree with the value of the parameter.

Input

The value to be placed in the tree.

Output

N/A

Assumptions

N/A

deleteItem

```
void deleteItem(iType);
```

Purpose

Remove a node from the tree.

Input

An item with the same key value as the item to be removed.

Output

N/A

Assumptions

The value of the item must exist in the tree, use retrieveItem to test before calling this unless it is certain that the item is in the tree.

resetTree

```
void resetTree(oType);
```

Purpose

Generate a queue to traverse the tree in a particular order.

Input

An order type which can be IN_ORDER, PRE_ORDER, or POST_ORDER, each one initializes a different queue.

Output

N/A

Assumptions

N/A

getNextItem

```
bool getNextItem(iType&, oType);
```

Purpose

Retrieve the next item in the queue corresponding to the value of oType.

Input

A reference parameter used to store the retrieved data and an order type of which queue to retrieve from.

Output

Return true if the item is the last item in the queue, also make the reference parameter equal to the value of the next item in the queue.

Assumptions

Assumes the proper queue as been initialized and is not empty. Use the resetTree and isEmpty functions to ensure this is the case.

printTree

```
void printTree(ostream&) const;
```

Purpose

Prints the tree to the desired ostream type.

Input

An ostream object by reference, this could be cout or an ofstream.

Output

Outputs all values of the tree to the desired ostream.

Assumptions

Assumes that the left shift operator ¡¡ has been overloaded for ostream to output the iType values properly.

countNodes

```
int countNodes(treeNode<iType>*);
```

Purpose

The client function called by numberOfNodes that does the recursive counting.

Input

A node pointer.

Output

Recursively determines the number of nodes connected to the parameter node plus 1.

Assumptions

Assumes parameter points to valid node or is NULL.

RETRIEVE

```
bool retrieve(treeNode<iType>*, iType&);
```

Purpose

The recursive component to retrieveItem.

Input

A node pointer and the item in question.

Output

Returns true if item is found, false if it is not.

Assumptions

Assumes parameter points to valid node or is NULL.

INSERT

```
void insert(treeNode<iType>*&, iType);
```

Purpose

The recursive component to insertItem, recursively traverses the tree to find the appropriate place to put the node.

Input

A node pointer and the item to be inserted.

Output

N/A

Assumptions

Assumes the pointer points to a valid node or is NULL.

DELETEOUT

```
void deleteOut(treeNode<iType>*&, iType);
```

Purpose

This portion of the function finds the correct node to be deleted recursively, then calls deleteNode to actually do the delete.

Input

A node pointer and the item to be deleted.

Output

N/A

Assumptions

Assumes the value held by the iType parameter exists in the tree.

deleteNode

```
void deleteNode(treeNode<iType>*&);
```

Purpose

Delete the node without breaking any rules of the binary search tree.

Input

The node the be deleted.

Output

N/A

Assumptions

Assumes the node is part of a valid binary search tree.

getPredecessor

```
void getPredecessor(treeNode<iType>*,
    Type&);
```

Purpose

Return the right most node of a left child.

Input

The left child of some node.

Output

Places the info from the right most child into the iType parameter.

Assumptions

Assumes the node is a valid node.

print

```
void print(treeNode<iType>*, ostream&);
```

Purpose

The recursive part of the print function, prints the tree inorder by printing the left child, the head, and then the right child.

Input

A node to be printed and an ostream object to print to.

Output

Prints all the values below the given node as well as the node using recursion.

Assumptions

Assumes the same as printTree as well as assuming the node is a valid node or NULL.

DESTROY

```
void destroy(treeNode<iType>*&);
```

Purpose

Recursively deallocate all values in the tree.

Input

A node pointer.

Output

N/A

Assumptions

Assumes the node pointer points to a valid node or is NULL.

COPYTREE

```
void copyTree(treeNode<iType>*&,
              treeNode<iType>*);
```

Purpose

Copy the values of the tree into another tree.

Input

The first parameter is the original tree, the second is the new tree.

Output

Copy all the nodes from one tree to the other.

Assumptions

Both trees are initialized and the left one is a valid tree.

preOrder

```
void preOrder(treeNode<iType>*&,
              queue<iType>&);
```

Purpose

Initialize the given queue with the values of the given tree traversed using pre-order traversal.

Input

A valid node pointer and an empty queue.

Output

Set the values of the queue to the pre-order traversal of the tree.

Assumptions

Assumes the queue is empty and the node pointer points to a valid node or is NULL.

inOrder

```
void inOrder(treeNode<iType>*&,
             queue<iType>&);
```

Purpose

Initialize the given queue with the values of the given tree traversed using in-order traversal.

Input

A valid node pointer and an empty queue.

Output

Set the values of the queue to the in-order traversal of the tree.

Assumptions

Assumes the queue is empty and the node pointer points to a valid node or is NULL.

postOrder

```
void postOrder(treeNode<iType>*&,
               queue<iType>&);
```

Purpose

Initialize the given queue with the values of the given tree traversed using post-order traversal.

Input

A valid node pointer and an empty queue.

Output

Set the values of the queue to the post-order traversal of the tree.

Assumptions

Assumes the queue is empty and the node pointer points to a valid node or is NULL.

3.2 user.h

getName

```
string getName() const
```

Purpose

Return the name of the user as a string.

Input

N/A

Output

Returns a copy of the private name member.

Assumptions

N/A

getPass

```
string getPass() const
```

Purpose

Obtain a copy of the password string.

Input

N/A

Output

Returns a copy of the private pass member.

Assumptions

N/A

setName

```
void setName( string& rhs )
```

Purpose

Change the name of the user.

Input

A string which will replace the current name.

Output

N/A

Assumptions

N/A

setPass

```
void setPass( string& rhs )
```

Purpose

Change the password for the user.

Input

A string which will replace the current password.

Output

N/A

Assumptions

N/A

operator*compare*

```
bool operator>( const user& rhs )
bool operator<( const user& rhs )
bool operator>=( const user& rhs )
bool operator<=( const user& rhs )
bool operator==( const user& rhs )
```

Purpose

Used to compare one user to another.

Input

Using the string class's compare function return how the current user compares to the right hand side.

Output

Return correct bool value depending on which operator is being used.

Assumptions

N/A

OPERATOR<<

```
friend ostream& operator<<( ostream &out,
        user& rhs );
```

Purpose

The overloaded left shift operator for ostream using a user in the right hand side, this makes ostream object correctly output user information using the ¡¡ operator.

Input

When used with an ostream object only the rhs is actually input.

Output

Print to the ostream the values inside of user.

Assumptions

N/A

3.3 part1.cpp

READFILE

```
bool readFile( string fileName,
        binaryTree<user>& tree );
```

Purpose

Read in a file and store it in the tree, this function randomizes the values to be inserted to prevent the tree from being unbalanced.

Input

The filename where the users are stored and a binary tree to store those users in.

Output

Store all the users in the file into the binary tree in random order, return true if file exists, false if it doesn't.

Assumptions

Assumes that if the file exists it does in fact contain a list of users and passwords separated only by whitespace.

STORETREE

```
void storeTree( binaryTree<user>& tree,
        oType order, string fileName );
```

Purpose

Store the tree into a file in a particular order.

Input

A binary tree to be stores, an enumerated order *IN_ORDER*, *PRE_ORDER*, *POST_ORDER*. Also a file name where to store the information.

Output

Output the values in the given order to a file.

Assumptions

N/A

promptForMenu

```
menuChoice promptForMenu();
```

Purpose

Prompt the user for a menu option and return the option when a valid choice is made.

Input

N/A

Output

Display the menu and continue to prompt the user until a valid value is entered. Return the value as an enumerated type menuChoice which can be ADD, DEL, VERIFY, PRINT, IN_ORD, PRE_ORD, POST_ORD, and EXIT.

Assumptions

N/A

3.4 heap.h

reheapDown

```
void reheapDown(int root, int bottom);
```

Purpose

Assumes the root is messed up and fixes the heap property from the root down

Input

root

The int that describes the root of the list

bottom

The int that describes the array value for the bottom

Output

None

Assumptions

Assumes that the heap has been initialized and contains elements

reheapUp

```
void reheapUp(int root, int bottom);
```

Purpose

Fixes the heap property when something has been added at the bottom left of the tree. Fixes all the way to the top of the tree.

Input

root
: The int that describes the root of the list

bottom
: The int that describes the array value for the bottom

Output

None

Assumptions

Assumes that the heap has been initialized and contains elements

swap

```
void swap(ItemType &a, ItemType &b);
```

Purpose

Swaps two elements in the array

Input

a
: Item to be swapped

b
: Item to be swapped

Output

None

Assumptions

None

3.5 pqueue.h

makeEmpty

```
void makeEmpty();
```

Purpose

Empties the entire heap.

Input

None

Output

None

Assumptions

None

isEmpty

```
bool isEmpty() const;
```

Purpose

Tells you if the heap is empty or not

Input

None

Output

Bool. True if empty, false if not empty

Assumptions

None

isFull

```
bool isFull() const;
```

Purpose

Tells you if the heap is full

Input

None

Output

Bool. True if full, false if not full.

Assumptions

None

ENQUEUE

```
void enqueue(ItemType newItem);
```

Purpose

Puts an item into the heap

Input

newItem

The item to be inserted into the heap

Output

None

Assumptions

Assumes there is room in the heap

DEQUEUE

```
void dequeue(ItemType& item);
```

Purpose

Pops out the root of the heap

Input

item

The item that will hold the value that was located in the root of the heap

Output

None

Assumptions

Assumes the heap is not empty

3.6 U_PQType.h

U_PQTYPE

```
U_PQType(int);
```

Purpose

Calls the constructor for PQType

Input

max

The integer value to make for the new heap

Output
None

Assumptions
None

Remove

```
void Remove(ItemType);
```

Purpose
Removes an item from a heap

Input
item
The item that will be removed from the heap

Output
None

Assumptions
None

Update

```
void Update(ItemType, ItemType);
```

Purpose
Updates the heap with a new value

Input
item
The item that is to be replaced
newItem
The item that will replace the old item

Output
None

Assumptions
None

printTree

```
void printTree(std::ostream&);
```

Purpose

Prints the tree to console

Input

out

The ostream object that will hold the text that is written to the console

Output

None

Assumptions

That there is a console to write to

3.7 part2.cpp

readFile

```
bool readFile( string fileName,
               U\_PQType<int>* &tree);
```

Purpose

Reads the file, counts how many numbers there are, initializes the heap and places the file contents into the heap

Input

fileName

The string object that will hold the file name of the file to be opened

tree

A U_PQTypw pointer that will be initialized inside the function

Output

Bool telling you if the read was a success

Assumptions

None

promptForMenu

```
menuChoice promptForMenu();
```

Purpose

Displays the menu and sets up the case for the action menu

Input

None

Output

The menuChoice, which is an enumerated type

Assumptions

None

4 Bugs and Errors

For the first section of the assignment there were no errors creating the binary tree, but there were however some problems when I began trying to use the binary tree in the code. The main problem was that the list of users we were given was stored in order which caused some headaches when trying to create a binary tree because I was ending up with a linked list rather than a tree. The way I knew this was that the pre-order traversal was outputting the same values as the in-order traversal. To correct this issue I created a function that read the file into an array first and then; using the property that any element in the array can be accessed in constant time; picked random elements in the array and placed them into the tree which worked very nicely in fact.

Another rather silly issue I spent way too much time on was that I forgot that ostream is part of the std namespace. It took nearly an hour of debugging to realize why I was receiving an entire page of cryptic compile errors. After adding a single line (using namespace std) to the beginning of the binaryTree.h file all those errors disappeared.

In part 2 of the assignment, the biggest bug that was found was that I could not figure out how to access the heaps element pointer in the U_PQType class. Even though the data members of pqueue were protected, the compiler would not let me access them. This bug was avoided by using a different approach to the problem that worked just as well in the end.

Another problem that was encountered was the correct syntax for using a template class as a pointer and passing that pointer by reference to a function. This was not so much an error, but more as an experience that helped me understand the syntax more clearly.

5 What was Learned

The main thing that was learned in this assignment was the implementation and use of different data structures. We were exposed to the binary search tree implemented as a link list and a heap and priority queue implemented as a dynamic array. This provided valuable experience to the students in the data structures involved and helped them understand what makes a well designed data structure and when to use the appropriate data structure.

The students also furthered their knowledge of templates and how to implement complex data structures as templates. This knowledge will help them in the future because they will be equipped with the knowledge of how to implement complex data structures using templates instead of rewriting data structures to handle every object type. The students are therefore better equipped to overcome the challenges that they will meet in the real world

and are better prospective employees as a result of their knowledge and practical application of using the data structures involved in this programming assignment.

6 Division of Labor

For this assignment, Josh did part 1 of the assignment. He wrote the binary tree implementation, the user class, and the driver. The binary tree was implemented using a linked list approach to allow for it's ease of expandability. He also worked on the documentation for those parts and helped overall with the report and with its writing.

Josiah wrote part 2 of the assignment. This involved writing the pqueue implementation which relied on the heap implementation that he also wrote. Josiah also wrote the U_PQType class that was derived from the pqueue class. Josiah implemented all of this using dynamic arrays so that the heap could be implemented efficiently. Josiah also worked on the report for the assignment and wrote the documentation for the functions that he implemented.

7 Inner workings of Remove and Update

The way the Remove function was implemented for the U_PQType is that it actually dequeues the entire list and then enqueues only the values that are not equal to the value being removed. While we do realize this is definitely not the most efficient method to accomplish this, we were rather pressed for time due to many end of the semester projects, tests, and assignments. The time complexity for this function is $O(N*(lg_2(n)+lg_2(n)) = O(n*lg_2(n))$ to dequeue the entire list then the same to enqueue it again. This makes the entire function a total of...

$O(n*lg_2(n))$

The method used for the Update function nearly identical to that of Remove where we simply dequeued the entire list and enqueued everything back onto it. The only difference is that when dequeueing the list, when we came across the desired item to be updated we changed it's value. The efficiency is therefore the exact same being...

$O(n*lg_2(n))$