CMPT 115 - Assignment #5

Summer 2016

Written Questions

Reminder: whenever we ask you to "write an algorithm", we want a complete algorithm, including a header, written in pseudocode. We will deduct 25% if your algorithms are in C++.

Use a single text file/document to contain all written questions (exercises 1, and part of exercise4). Microsoft Word documents (doc, docx) should not be handed in. Use text-only format (.txt) or PDF format (.pdf). If you are not sure about this, ask your TA. If the marker cannot open your file (for whatever reason), you will receive a grade of 0 for the written part.

Name your submission file a5-written.txt. Ensure that each answer is clearly marked for the question it relates to; and ensure that your file/document is clearly marked with your name, student number and NSID at the top.

For written questions and algorithms, you can use <- instead of ← and you can use -> to derefence followed by selecting a field of a record.

Exercise 1 Huffman Coding (20 pts)

Consider the following table of letter frequencies:

t	h	i	s	a	r	n	g	‘ ‘
16	8	6	9	11	2	3	2	5

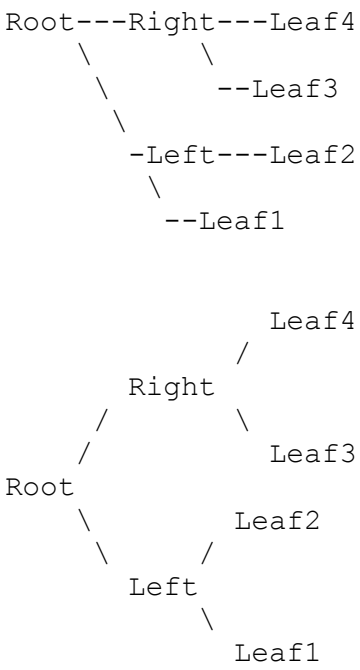
- Use the given frequencies to build a Huffman tree. There isn't just one solution, so your answer may be different from others.
- Use your Huffman tree to encode the string "this is a string" as bits. Be sure to encode the blank spaces (but not the quotation marks).

What to hand in:

In your file a5-written.txt, indicate the question (exercise) number very clearly, so it can be seen easily by a marker.

To draw a tree, you can use some drawing software, or you can submit your answer in text. If you use a software package, be sure to submit the drawing as a PDF file only. Other formats might be rejected if they cannot be opened.

To draw a tree in text, you can turn the tree on its side, like either of these two examples:



Your tree may be a little bigger than these, and your nodes will contain information as processed by the Huffman tree algorithm (as opposed to the example which shows nothing about Huffman trees).

Grading:

1. 10 marks. Your tree has all 9 characters in it. The structure of the tree is correct (binary branching). The more common characters are closer to the top of the tree, as required by the Huffman encoding.
2. 10 Marks. Your string encoding uses the tree correctly, and results in a sequence of bits that uniquely identifies each character.

Programming Questions

All C++ source code files must contain your name, student number, and NSID; and the course, assignment number, and question number as comments at the top of the file. All C++ programs must compile under g++ without errors or warnings, using the standard flags described in tutorial, namely -Wall -pedantic. Use of String class or other object-oriented code (except for cin or cout and related file-stream objects, which are allowed) will result in a flat deduction of 25%.

Exercise 2 Using trees to structure information (50 pts)

In this question, you will implement (or re-implement) operations for a Binary Tree ADT. The tree is very similar to what you have seen in class, though maybe not exactly the same. You will be given the application, which uses the ATD operations. Your job is to implement the operations so that the application runs.

The operations are specified in Tree.h, and you will create a file called Tree.cc, which implements every function specified in Tree.h. Once your job is done, the whole application can be built by using the command-line:

```
g++ -Wall -pedantic Tree.cc Game.cc -o Game
```

Before you begin, you will need to obtain the following files:

- Game.cc – the application using trees
- Tree.h – the definition of the Tree struct, and the headers for the operations
- Element.h – a place to define the data stored by the trees

These are available on the Moodle page, close to the assignment description. Read them carefully, as you will have to work with them to complete your work.

You've done an exercise like this before, so this is just more of the same. To make good progress, you should create operation stubs (functions that have the right parameters, but do nothing useful), and then implement each operation, one at a time. If you write a simple test program (different from Game.cc, you might be able to catch your bugs a little easier than if you write everything and test everything all at once.

The application is not a serious application. It uses the tree structure for a conversation between the program and a user. The important part is the experience you obtain working with trees.

What to hand in:

Hand in your C++ program as contained in the following file:

- Your Tree ADT implementaiton, in the file Tree.cc.

You should not add any files to this list, because the markers are going to try to compile your program exactly once, using the following command:

```
g++ -Wall -pedantic Tree.cc Game.cc -o Game
```

If your program does not compile with this command, the marker will not try anything else.

Grading:

- 5 marks: Your program compiles without error using the compiler flags -Wall -pedantic, as detailed above.
- 45 marks: Tree ADT:
  - (5 marks) createTree() correctly creates a tree.
  - (5 marks) destroyTree() correctly destroys the whole tree.
  - (5 marks) insertLeft() correctly puts one tree as left subtree of the other.
  - (5 marks) insertRight() correctly puts one tree as right subtree of the other.
  - (5 marks) getLeft() correctly returns the left subtree of a given tree.
  - (5 marks) getRight() correctly returns the right subtree of a given tree.
  - (5 marks) preOrder() correctly displays the Elements in the tree using pre-order traversal.
  - (5 marks) inOrder() correctly displays the Elements in the tree using in-order traversal.
  - (5 marks) postOrder() correctly displays the Elements in the tree using post-order traversal.

Exercise 3 Building Binary Search Trees (25 pts)

In this question, we will build a partial Binary Search Tree ADT from the Binary Tree ADT we implemented in the previous question. To keep things relatively simple, you will only implement the following two operations (that's why it's partial):

- search()
- insert()

These operations are specified in the file BST.h.

Notice that BST.h usesTree.h. The trees you build in this exercise will use the Tree structure directly (whereas in the Notes, we had a Node structure). There are advantages and disadvantages to this approach, but the experience will be valuable. One advantage is that we can reuse the Tree operations directly (like inOrder()). However, the implementation of the Binary Search Tree operations cannot be copied directly from the notes; you'll have to adapt them to this purpose.

Notice also that you will need to edit the file Element.h because in this question, the data stored in the tree are integers. If you read the file Element.h, you'll see that the change is pretty easy.

We have also provided an application for your operations, which you can find in the file BSTDriver.cc. This is a simple program to store a bunch of random integers in a binary search tree, and then display them all to the console using the inOrder() operation. There are a few other tests in the application that you can review.

Before you begin, you will need to obtain the following files:

- BSTDriver.cc – the application using binary search trees
- Tree.h – the definition of the Tree struct, and the headers for the operations
- BST.h – the headers for the Binary Search Tree operations
- Element.h – a place to define the data stored by the trees

These are available on the Moodle page, close to the assignment description. Read them carefully, as you will have to work with them to complete your work.

Note: The two programming exercises use different definitions for Element.

What to hand in:

Hand in your C++ program as contained in the following file:

- Your Binary Search Tree ADT implementation, in the file BST.cc, containing the two operations.

You should not add any files to this list, because the markers are going to try to compile your program exactly once, using the following command:

```
g++ -Wall -pedantic BST.cc Tree.cc BSTDriver.cc -o BSTDriver
```

If your program does not compile with this command, the marker will not try anything else. Note that you will need to use your Tree.cc file from the previous question. However, since you will not need many of the operations from Tree.cc in this question, you do not need to complete the previous question completely to do this question.

Grading

- 5 marks: Your program compiles without error using the compiler flags -Wall -pedantic, as detailed above.
- 20 marks: BST ADT:
  - (5 marks) search() works correctly.
  - (10 marks) insert() works correctly.
  - (5 marks) No other changes were made to the Tree ADT.