O COMP 215 Algorithms Θ

Lab #4

In this lab, we will review (well, you will anyway, I'm not helping you in this description) file input and output in C++. You will also be doing funny stuff with large amounts of data you read from the file. You have to hand in a .zip file that contains all the files with your code, and a text file answers.txt that contains the answers to all the questions I ask below. Do NOT include the file with the random numbers. It is too big. I don't want to see it. Hand in everything before Friday, September 19, 11:59pm.

The instructions for this lab are given at a slightly higher level than before, I now expect you to be able to figure some stuff out on your own. Remember that if you have trouble with C++ syntax, or you do not remember the exact way to use standard functions, http://www.cplusplus.com/ and Google are very useful resources.

First we write a large file full of random numbers. The file must contain 100,000,000 (one hundred million) lines, each line contains two random integers.

1. Write a function `generateFile` that takes no input, returns no output, but writes the files with 100,000,000 lines, each containing two random integers. The file should be called `InputNumbers.dat`. You can use any method you like for writing to the file (old style `fprintf`, or file output streams), but you should use the `rand()` function to generate the 200,000,000 random numbers you will need for the file. Write two random number per line, and 100,000,000 lines.

Why did we write two numbers per line? Because we will be using them in pairs of course! For this, we will need a data structures that stores pairs of numbers.

2. Write a class `integerPairs` that contains two private integers: `value1` and `value2`.

3. Write a constructor for this class that takes two integers `a` and `b`, and gives the value `a` to `value1` and `b` to `value2`

4. Overload the operators ==, >, <, <= and >= so that we have that if (a, b) and (c, d) are pairs, then $(a, b) == (c, d)$ if and only if $a = c$ and $b = d$, and $(a, b) < (c, d)$ if and only if $a < c$ or if $a = c$ and $b < d$. All these functions should be public.

Now, we need a function that will read the big file we created before, and store it in an array of `integerPairs`.

5. Write a function `readFile` that takes an integer `nbLines` as input, allocates an array of `integerPairs` of size `nbLines`, reads the first `nbLines` from `InputNumbers.dat`, puts them in the array and returns a pointer to that array.

Great, now that we have an array, we can start doing stuff with it.

6. Write a function `findMax` which, given an array (in the form of a pointer) of `integerPairs` and its size `n`, returns the largest `integerPair` together with the index of the largest `integerPair`. Since the function returns two outputs, it should do so by passing some of its input by reference.

7. Write a function `ourSort` which, given an array (in the form of a pointer) of `integerPairs` and its size `n` sorts the values of the arrays as follows:

   (a) First find the maximum element among all `n` elements of the array and its index `i`. Then, swap the $\texttt{i}^{th}$ and $\texttt{n}^{th}$ element of the array, so that the maximum element of the array is now at the end.

   (b) Then, find the maximum element among the first `n-1` remaining elements of the array and its index `i`. Then, swap the $\texttt{i}^{th}$ and $\texttt{(n-1)}^{st}$ element of the array, so that the two maximum element of the array are now at the end.

   (c) Then, find the maximum element among the first `n-2` remaining elements of the array and its index `i`. Then, swap the $\texttt{i}^{th}$ and $\texttt{(n-2)}^{nd}$ element of the array, so that the three maximum element of the array are now at the end.

   (d) etc, all the way down to element 1.

8. Remember the function `clock()` that we used in a previous lab to determine the time taken to execute some code? We will use it again to determine how long it takes to sort arrays of various sizes. Write a function `ourSortTiming` which takes an integer `size` and does the following:

   (a) Use the function `readFile` to get a pointer to an array of `size` integerPairs.

   (b) Starts the clock.

   (c) Uses the function `ourSort` to sort the element of the array in increasing order.

   (d) Stops the clock and prints on screen the number of clock ticks taken to sort the array.

   (e) Delete the array of integerPairs.

9. Run the function `ourSortTiming` on arrays of various size. You may need to run it on fairly large arrays before it takes a measurable amount of time. In rough terms, how much longer does it take when the size of the array doubles? triples? quadruples?

Finally, remember the `binarySearchTree` we did a while ago? Modify the `treeNode` class so that it contains an `integerPair` instead of an ordinary integer and modify the constructor accordingly. You will also need to modify the input type of the functions `insert`, `insert_help`, `search` and `search_help` so that your binary search tree now works with integerPairs instead of ordinary integers. You can remove the traversal function, we will not use it here.

10. Add a function `depth` that takes no input and returns an integer to the class `binarySearchTree`. The depth of a tree is the length of the longest branch from the root. That function should follow the same general pattern as all other functions in the class: the function `depth` is public, and it calls a private recursive function `depth_help` that takes a pointer to a `treeNode` as its input. To calculate the depth of the tree, use the following:

- the depth of the empty (NULL) tree is zero.
- if the tree is not NULL, then calculate the depth `d1` of the left subtree, the depth `d2` of the right subtree and return 1 + max(`d1`, `d2`).

11. Write a function `ourTreeTiming` which takes an integer `size` and does the following:
    (a) Use the function `readFile` to get a pointer to an array of `size` integerPairs.
    (b) Initialize a binarySearchTree.
    (c) Starts the clock.
    (d) Insert all the `size` elements of the array into the tree.
    (e) Stop the clock, and prints on screen the number of clock ticks taken to insert all the elements in the tree, and also print the depth of the tree.
    (f) Delete the array of integerPairs and the binary search tree.
12. Run the function `ourTreeTiming` with various sizes. You may need to run it on fairly large sizes before it takes a measurable amount of time. In rough terms, how much longer does it take and how is the depth modified when the size is multiplied by 2? by 4? by 8? by 16?