1 Instructions

When you are finished submit all your work through the MyClasses page for this class. Create a directory called Homework11, put each programming exercise into its own subdirectory of this directory, zip the entire Homework11 directory up into the file Homework11.zip, and then submit this zip file to Homework #11.

Make sure that you:

- Follow the coding and documentation standards for the course as published in the MyClasses page for the class.
- Check the contents of the zip file before uploading it. Make sure all the files are included.
- Make sure that the file was submitted correctly to MyCLasses.

All non-templated class structures are to have their own guarded specification file (.h) and implementation file (.cpp) that has the same name as the class. All templated class structures are to be guarded and written entirely in their (.h) file. No inline coding in the class specification. In addition you must create a make file that compiles and links the project on a Linux computer with a Debian or Debian branch flavor.

2 Exercise #1

This exercise is to create a program to do timing analysis on several operations associated with binary trees, AVL trees, and arrays. In all, it will compare the complexities of inserting data into binary trees and AVL trees (essentially sorting the data) as well as the algorithm library's sort function on an array. We will then further test the speed of searching binary and AVL trees against the binary search.

1. Create a directory for this exercise, create an empty main program, copy over the BinaryTree.h file from the examples, and copy over the AVLTree.h file from the examples.
2. In the main, set up the standard program timing structure using the chrono library that we have used in the past to time processes.
3. Bring over the recursive binary search function for arrays from the examples and add it to the program. As usual, prototype at the top and the function below the main.
4. Set up the usual random number generating code with seed setting done from the system clock. Of course, put this at the top of the main.
5. In the main, define an integer binary tree and an integer AVL tree.

6. Ask the user for the number of integers to use and the number of searches they wish to do.
7. Create an integer array (in the heap) of the "number of integers" size.
8. Fill the array with random numbers between 1 and 1,000,000,000.
9. Insert each of the elements of the array into the binary tree and time that process. Print the timing result to the console.
10. Insert each of the elements of the array into the AVL tree and time that process. Print the timing result to the console.
11. Use the algorithm library sort function on the original array and time that process. Print the timing result to the console.
12. For the binary tree, do the "number of searches" on the binary tree. That is, for each search generate a random target integer between 1 and 1,000,000,000, and search the tree for the target. Time the process to do all of the searches and print the timing result to the console.
13. For the binary tree, do the "number of searches" on the AVL tree. That is, for each search generate a random target integer between 1 and 1,000,000,000, and search the tree for the target. Time the process to do all of the searches and print the timing result to the console.
14. For the (now sorted) array, do the "number of searches" on it. That is, for each search generate a random target integer between 1 and 1,000,000,000, and search the array for the target using the binary search. Time the process to do all of the searches and print the timing result to the console.
15. Have the program print out the total times for each of the binary tree, AVL tree, and array. That is, add the times for inserting and searching each tree and the times to sort and search the array. Print these totals to the console screen.
16. At the end of the program, free the memory allocated to the array.

When you run the program the output should look like the following. Of course, your timing values will be different.

```
Enter number of integers to use: 100000
Enter number of searches to use: 10000000

Time to insert into Binary Tree: 0.0325662 seconds.
Time to insert into AVL Tree: 0.0669677 seconds.
Time to sort array (algorithm library method): 0.0180369 seconds.

Time to search binary tree: 2.6732 seconds.
Time to search AVL tree: 2.42267 seconds.
Time to search array: 2.39253 seconds.
```

```
Total Times in Seconds
Binary Tree  2.70577
AVL Tree     2.48964
Array        2.41057
```

Now it is time to do some analysis of the times for different sizes of the data set and different numbers of searches. Create a report document for this analysis. For each run you do in the analysis below include the output of the program and answers to all the questions that are asked. Export the report to PDF and include it with the code for the exercise.

1. Do runs with the data size set to 1000 and searches at 1,000, 10,000, 100,000, 1,000,000, and 10,000,000.
2. Do runs with the data size set to 10,000 and searches at 1,000, 10,000, 100,000, 1,000,000, and 10,000,000.
3. Do runs with the data size set to 100,000 and searches at 1,000, 10,000, 100,000, 1,000,000, and 10,000,000.
4. Do runs with the data size set to 1,000,000 and searches at 1,000, 10,000, 100,000, 1,000,000, and 10,000,000.
5. **For each of the data size runs above**, answer the following questions.
   (a) Comparing the total times, is any method (BST, AVL, array) always better than the other two?
   (b) Does any method work consistently better on smaller numbers of searches?
   (c) Does any method work consistently better on larger numbers of searches?
6. Comparing all of the data you accumulated. Assuming the data is entering your program in a "random" stream.
   (a) If you have a "small" data set but need to search it many times (i.e. several orders of magnitude over the data set size), which method would you choose and why?
   (b) If you have a "large" data set and you need to search only a few times (i.e. several orders of magnitude under the data set size), which method would you choose and why?

3 Exercise #2

In this exercise you will simply take the program from the above exercise and move the sorting of the array above the inserting of the data into the binary and AVL trees. Hence the data that is entering the trees is already sorted.

When you run the program the output should look like the following. Of course, your timing values will be different.

```
Enter number of integers to use: 10000
Enter number of searches to use: 100000

Time to sort array (algorithm library method): 0.00391534 seconds.
Time to insert into Binary Tree: 0.286162 seconds.
Time to insert into AVL Tree: 0.00408127 seconds.

Time to search binary tree: 1.51385 seconds.
Time to search AVL tree: 0.016059 seconds.
Time to search array: 0.0181361 seconds.

Total Times in Seconds
Binary Tree  1.80001
AVL Tree     0.0201402
Array        0.0220515
```

Now it is time to do some analysis of the times for different sizes of the data set and different numbers of searches. Create a report document for this analysis. For each run you do in the analysis below include the output of the program and answers to all the questions that are asked. Export the report to PDF and include it with the code for the exercise.

1. Do runs with the data size set to 1000 and searches at 1,000, 10,000, 100,000, and 1,000,000.
2. Do runs with the data size set to 10,000 and searches at 1,000, 10,000, 100,000, and 1,000,000.
3. Do runs with the data size set to 100,000 and searches at 1,000, 10,000, 100,000, and 1,000,000.
4. **For each of the data size runs above**, answer the following questions.
   (a) Comparing the total times, is any method (BST, AVL, array) always better than the other two?
   (b) Does any method work consistently better on smaller numbers of searches?
   (c) Does any method work consistently better on larger numbers of searches?
5. Comparing all of the data you accumulated. Assuming the data is entering your program in a sorted or close to sorted stream, for example, reading from a dictionary file that is already sorted in alphabetic (lexicographic) order.
   (a) If you have a "small" data set but need to search it many times (i.e. several orders of magnitude over the data set size), which method would you choose and why?
   (b) If you have a "large" data set and you need to search only a few times (i.e. several orders of magnitude under the data set size), which method would you choose and why?

4 Exercise #3

This exercise will use the concept of function pointers to run sort timings for the bubble, insertion, selection, merge, and quick sorts using a single function to do the timing and sending a pointer to the sorting function to the timing function. In this exercise the actual timing of the sorts is not important, it is the sending of the function as a pointer parameter to another function. We did this in the set of examples on function pointers but here we will use a vector to store the pointers to the functions.

1. Create a directory for this exercise, create an empty main program, copy over the templated versions of the bubble sort, insertion sort, selection sort, quick sort, and merge sort from the previous timing assignments or the example code for the class. As usual, prototypes at the top and implementations below the main. These functions will not need to changed at all, in fact, that is one of the points to this exercise.

2. Add in the templated function `Copy` that takes in two arrays of type T (the template) and an integer holding the size of the arrays. The arrays are assumed to have the same numbers of elements. The function will simply copy all of the data from the first array to the second array. So the call `Copy(A, B, num);` will copy all of the contents of the array A to the array B, both arrays have num elements.

3. Add in the templated function `SortTiming` that takes in 4 parameters. The first is the array to be sorted, the second is the size of the array, the third is a string that will simply hold the name of the sorting algorithm. The fourth parameter is a pointer to the sorting function. It would be nice to use the typedef method here as it would simplify the code and be more readable but unfortunately the typedef does not work with templating, even C++ has some limitations. So the fourth parameter has type `void (*)(T[], int)`. Looks fairly ugly but it is defining a pointer to any function that has two parameters, an array and integer, and has no return type. This is exactly what each of our bubble sort, insertion sort, selection sort, quick sort, and merge sort functions do.

   The `void (*)(T[], int)` should be the fourth parameter in the prototype of the `SortTiming` function but in the implementation we need to give the function a name so that we can use it. So in the header of the implementation for the function the fourth parameter should be `void (*fct)(T[], int)`. Now we can invoke the function using the name `fct`.

   This function will run the `fct` function on the array and array size that is coming in as the first two parameters to the `SortTiming` function. It will time this process and display the results of the timing to the console screen. In the output you will incorporate the name of the algorithm that came in as the third parameter to the `SortTiming` function.

4. In the main:

   (a) Ask the user for the number of integers to use.

(b) Create two arrays, A and B, of integers that both have the input size (hence put them in the heap).

(c) Populate array A with random numbers between 1 and 1,000,000,000.

(d) Now the fun stuff. Create a vector that will hold pointers to the sorting functions. Even though all the sorting functions are templated we need to have a specific data type for the array type in this declaration. Again this is a little messy but the data type for the vector will be `void (*)(int[], int)`. That is what goes in the < >. Load into the vector the function names (i.e. the function pointers) for the 5 sorts.

(e) Create either an array of strings or a vector of strings that stores the names of the sorting algorithms in the same order as the vector of pointers above.

(f) Now drop the following loop below this. The fcts is the name of the vector of pointers and names is the name of the array holding the sort names. If you used different names for these variables you will need to substitute them here.

```
for (unsigned long i = 0; i < fcts.size(); i++) {
    Copy(A, B, num);
    SortTiming(B, num, names[i], fcts[i]);
}
```

(g) Make sure you delete the allocated memory for A and B at the end of the program. All in all your main will be about 25 lines long.

If all is in order ten a run should produce the following output. Of course, your timing values will be different.

```
Enter number of integers to use: 10000
Time to sort with Bubble Sort: 0.233875 seconds.
Time to sort with Insertion Sort: 0.0628237 seconds.
Time to sort with Selection Sort: 0.11236 seconds.
Time to sort with Merge Sort: 0.00131694 seconds.
Time to sort with Quick Sort: 0.00106066 seconds.
```

5 Exercise #4

We are now going to take the previous exercise one step further and get it to work with the `for_each` function that is included in the algorithm library of C++. To do this we need to turn our `SortTiming` function into a function with one parameter that has the same data type pointed to by the pointers in the first two parameters of the `for_each` function. So we need to convert 4 parameters into one parameter, this is easy to do with a struct.

1. Copy the code from the previous exercise over to a new directory.

2. Create a struct called `sortinfo`, the name is really not important but it will be easier to follow the discussion if we have a name for it. The `sortinfo` struct will be templated (type is named T), with 4 fields. One field is a pointer to a type T (pointing

to the array to be sorted), the next field is the size of the array and hence an integer, the third field is a string holding the name of the sorting method, and the last field is a pointer to the sorting function that will be used. I will be nice here, the field should be `void (*fct)(T[], int);`. If you have followed along with this material (I know it is a bit esoteric) up to now this should make sense and you probably did not need me to give this to you.

3. Make sure that the `SortTiming` function prototype is below this struct definition and change the parameter list to a single parameter of `sortinfo` type. Remember to template it and you probably want to send the parameter by reference.

4. In the implementation of the `SortTiming` function you will need to change the parameter to the single parameter of `sortinfo` type as well, obviously. You will also need to make the obvious syntax changes to access the fields of the struct. Also in the `SortTiming` function, move the array B from the main to the function and call the `Copy` function to copy the array pointed to in the struct to B. Then time the sorting of B. Make sure you do not forget to free the memory for B at the end of the function.

5. In the main, remove the loop,

```
for (unsigned long i = 0; i < fcts.size(); i++) {
    Copy(A, B, num);
    SortTiming(B, num, names[i], fcts[i]);
}
```

6. In its place we will put the following. Create an array of 5 `sortinfo` types that are templated to store integer arrays. I called the array `sorts`. In a for loop, populate the `sorts` array with 5 `sortinfo` structs, each has a pointer to the same array, A, to be sorted, and its size. The name should be the sorting algorithm name from the names array, and the function pointer should be the pointer to the corresponding sorting function from the fcts vector. Sounds complicated but the really the body of the loop is a single line of code.

7. Now below this loop place the single line of code, and you are done.

```
for_each(sorts, sorts + 5, SortTiming<int>);
```

Run the program and you should get the same output as you did in the last exercise.

```
Enter number of integers to use: 10000
Time to sort with Bubble Sort: 0.235215 seconds.
Time to sort with Insertion Sort: 0.0623978 seconds.
Time to sort with Selection Sort: 0.112158 seconds.
Time to sort with Merge Sort: 0.00129893 seconds.
Time to sort with Quick Sort: 0.00107236 seconds.
```

6 Exercise #5

This exercise is to add in a function to the binary tree class and the linked list class that will apply a function to the entire collection.

1. Create a new directory and put in the following main program.

```
#include <algorithm>
#include <cctype>
#include <cmath>
#include <cstdlib>
#include <ctime>
#include <iostream>

#include "BinaryTree.h"
#include "LinkedList.h"

using namespace std;

template <class T> void squarert(T &i) { i = sqrt(i); }
template <class T> void sqr(T &i) { i = i * i; }
template <class T> void pt(T &item) { cout << item << " "; }
void toupperchar(char &c) { c = toupper(c); }
void upppercase(string &str) { for_each(str.begin(), str.end(), toupperchar); }

void div() { cout << "\n-----------------------------------\n\n"; }

int main() {
    srand(time(0));

    // Create the trees.
    BinaryTree<int> bt;

    bt.insertNode(10);
    bt.insertNode(4);
    bt.insertNode(12);
    bt.insertNode(7);
    bt.insertNode(15);

    bt.PrintTree();
    cout << endl;

    bt.apply(pt);
    cout << endl << endl;

    bt.apply(sqr);
    bt.PrintTree();
    cout << endl;

    div();

    BinaryTree<double> btd;

    btd.insertNode(10);
    btd.insertNode(4);
    btd.insertNode(12);
    btd.insertNode(7);
    btd.insertNode(15);

    btd.PrintTree();
    cout << endl;

    btd.apply(squarert);
    btd.PrintTree();
    cout << endl;

    btd.apply(pt);
    cout << endl;

    div();

    BinaryTree<string> btstr;
```

```
    btstr.insertNode("Steve");
    btstr.insertNode("John");
    btstr.insertNode("Sue");
    btstr.insertNode("Kim");
    btstr.insertNode("Tom");

    btstr.PrintTree();
    cout << endl;

    btstr.apply(upppercase);
    btstr.PrintTree();
    cout << endl;

    btstr.apply(pt);
    cout << endl;

    div();

    LinkedList<int> LLi;
    LLi.appendNode(10);
    LLi.appendNode(5);
    LLi.appendNode(17);
    LLi.appendNode(121);
    LLi.appendNode(15);

    LLi.displayList();
    LLi.apply(pt);
    cout << endl;
    LLi.apply(sqr);
    LLi.apply(pt);
    cout << endl;

    div();

    LinkedList<string> LLs;
    LLs.appendNode("Steve");
    LLs.appendNode("John");
    LLs.appendNode("Sue");
    LLs.appendNode("Kim");
    LLs.appendNode("Tom");

    LLs.apply(pt);
    cout << endl;
    LLs.apply(upppercase);
    LLs.apply(pt);
    cout << endl;

    return 0;
}
```

2. Copy over the header files for the binary tree and for the linked list classes, we want the templated versions for these.

3. Add a function called `apply` to both the linked list and binary tree classes. Really you will probably want two functions, one recursive and the other not recursive that starts up the recursive one. These functions are to obviously be templated. They will take in as a parameter a pointer to a function. The parameter function will be void, have a single parameter sent by reference, and update its value. This parameter will be applied (in the `apply`) function to the node's value. The `apply` functions will traverse the entire collection class and apply the parameter function to each node in the collection.

When the above main program is run the output will be the following.

```
        15
    12
10
        7
    4

4 7 10 12 15

        225
    144
100
        49
    16


-----------------------------------

        15
    12
10
        7
    4

        3.87298
    3.4641
3.16228
        2.64575
    2

2 2.64575 3.16228 3.4641 3.87298

-------------------------------------

        Tom
    Sue
Steve
        Kim
    John

        TOM
    SUE
STEVE
        KIM
    JOHN

JOHN KIM STEVE SUE TOM

-------------------------------------

10
5
17
121
15
10 5 17 121 15
100 25 289 14641 225

-------------------------------------

Steve John Sue Kim Tom
STEVE JOHN SUE KIM TOM
```