Homework 2

100 points

# Objectives:

* Implement generalized c++ functions/classes
* Use c++ topics that we have covered: const, unit testing, operator overloading, friend function
* Design and implement unit tests for a templated class

# Turn in:

* Linear.hpp, Tree.hpp OR HashMap.hpp (your choice), test.cpp, Makefile
* You are **not** required to turn in main.cpp though you are highly encouraged to write a main as you test your Linear and Tree objects!
* You do **not** need to turn in catch.hpp.

# Instructions:

You will implement templated classes for the data structures:

* Linear - using vectors

And **one** of these options (or both for extra credit):

* Binary Search tree - using linked lists
* Hash Map - using vector of vectors

Below are the interfaces for the classes you will implement. You may include additional functions as needed. Note: it is your job to determine which parameters and methods should be const!

**General reminders:**

* Cite any resources you use, including ChatGPT
* Include comments for the sections where you are required to write up answers
* Start early and ask questions often!

## Linear.hpp

|  |  |
| --- | --- |
| **Function Signatures** | **Description** |
| Linear() | Default constructor |
| void insertElement(T val) | Insert element at the last position in the vector. You should insert the elements in the order it is provided. You should not be sorting it at insertion. |
| T getElementAtIndex(int index) | Return element at specified index |
| bool search(T val) | Returns if the element is found in the vector |
| void deleteElement (T val) | Delete the element from the vector |
| std::ostream& operator<<(std::ostream& os, const Linear<U> &l); | overload the << operator for Linear<T>. This should print out the contents of the vector in Linear in the format: {T, T, T, T} |

**Tree.hpp**

This class will represent a Binary Search Tree

|  |  |
| --- | --- |
| **Function Signatures** | **Description** |
| Tree(T val) | Initialize the root node for the Tree |
| ~Tree() | Should delete the entire tree. |
| void insertElement(T val) | Insert an element at the appropriate position in the Binary Search Tree |
| bool search(T val) | Returns if the element is found in the Binary Search Tree |
| void deleteElement (T val) | Delete the element from the Tree |
| std::ostream& operator<<(std::ostream& os, const Tree<U> &t); | overload the << operator for Tree<T>. This should print out the contents of the Tree in the format: {T, T, T, T}. You will use inorder traversal for this function. Hint: using a helper function may be helpful. |

## HashMap.hpp

This class will implement a custom Hash Map. You can write a hashing function of your choosing or you may use the [hash function](https://en.cppreference.com/w/cpp/utility/hash) from std. Keep in mind that it should work for all data types and for large data sets.

|  |  |
| --- | --- |
| **Function Signatures** | **Description** |
| HashMap(std::vector<T> vals, int size) | Initialize the map with a list of elements. You will also fix the size of your hash table(map) through the constructor. **Note**: this would be independent of the size of the list of elements you are inserting. |
| int hashKey(T val) | This function should return the bucket in which the element should be inserted. |
| void insertElement(T val) | Insert an element at the appropriate position in the Map. |
| bool search(T val) | Returns if the element is found in the Map |
| void deleteElement (T val) | Delete the element from the Map |
| std::ostream& operator<<(std::ostream& os, const HashMap<U> &m); | overload the << operator for Map<T>. This should print out the contents of the Map in the format: {T, T, T, T}. You will print the elements in the order of the keys. |

**Testing the classes with different types:**

Your classes must work for types T that are new, custom types, such as programmer- defined structs and classes. Each method that you implement must be adequately tested. You do not need to test each method of every class with every type that T could be (that would be impossible!), but your different TEST\_CASEs should make use of Linear, Tree and HashMap that hold a variety of different types.

See examples [in the examples folder](https://github.com/CSCI-3010-CUBoulder/ClassResources/blob/main/examples/templates_examples.md) on github for how to write templated classes and functions, as well as the resources linked to [in the resources document.](https://github.com/CSCI-3010-CUBoulder/ClassResources/blob/main/resources.md#templating)

We are happy to clarify any methods/requirements that you'd like guidance on, so **please, make sure to ask if you have any questions!**

## As always, your functions should be well documented. Since a main.cpp is not required, include your file comment with your name(s) and instructions for running your program in test.cpp.

**Some thoughts on getting started:**

Though you may have the inclination to start by writing a non-templated version of the classes and then converting it, our experience has been that getting a templated class started in c++ can be difficult enough that this might make finding your compiler issues harder. Therefore, we recommend the following steps:

1. Define your classes with just a constructor.
2. Make sure you can create a Linear/Tree/HashMap (or some other primitive/built in type).
3. Write unit tests for one of the methods
4. Implement the method
5. Run your tests
6. Go back to step 3 and repeat until complete

**Verifying time complexities for the data structures:**

In addition to testing the methods of the classes you will also verify the time complexity for searching an element in the data structures. **Provide your observation as comments** in your test.cpp alongside your assertions.

The input is a list of unique elements. You can start with a small dataset and work up to test with the datasets provided in the files - integers.csv, decimals.csv, strings.csv

The comparisons of time complexity for the search operation you will perform will be with elements at:

1. The first position in the input list
2. The last position in the input list
3. A random position in the input list

For this task, you may use a time library of your choice or you may work with (sys/time.h)[<https://pubs.opengroup.org/onlinepubs/7908799/xsh/systime.h.html>]

**Rubric Outline**

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Description** | **points** |
| Linear, Tree or HashMap | these will be roughly equally spread between all methods that we've asked you to implement | 50 |
| Unit tests | TEST\_CASES, SECTIONs, REQUIRE used appropriately | 25 |
|  | each method appropriately tested |  |
|  | Note: no unit testing required for overloading  the << operator |  |
| Verifying time complexities | Observations on performance of Data Structures | 15 |
| Style and comments | const and overloading used appropriately | 5 |
|  | follows style guidelines | 2.5 |
|  | commented appropriately | 2.5 |
| Extra Credit options | Implement a 3rd data structure (Binary tree AND HashMap) | 15 |
|  | Balance the tree and check for time complexity for search in a balanced vs unbalanced BST | 15 |