SAN DIEGO STATE UNIVERSITY

Department of Computer Science
CS 210 Data Structures

Program 4 Instructions

Student Objective

In this assignment, you will delve into implementing essential data structures—Hash Table with Chaining and Binary Search Tree—to efficiently manage large data sets. Your tasks include:

- Developing a Binary Search Tree with specified functionalities such as size calculation, height determination, and proper node removal.
- Implementing a Hash Table using chaining with LinkedList.
- Ensuring adherence to provided guidelines, including pre- and post-conditions for functions and appropriate handling of tree operations.
- Thoroughly testing your implementations to ensure correctness and efficiency.

Instructions

The data structures studied previously are not suitable for managing large data sets. For this assignment, you will implement two very useful data structures: Hash Table (using a Linked List) and Binary Search Tree.

Class prototypes defined in `BinarySearchTree.h`, `LinkedList.h`, and `HashTable.h` are provided. You *must* review each of these files in order to fully understand the expected functionality of each class.

Program 4 will consist of the following files. You must use exactly these filenames.

`BinarySearchTree.h`	The binary search tree definition (provided).
`LinkedList.h`	The linked list definition (provided).
`HashTable.h`	The hash table definition (provided).
`BinarySearchTree.cpp`	The BST implementation (you create).
`LinkedList.cpp`	The linked list implementation (you create).
`HashTable.cpp`	The hash table implementation (you create).
`test.cpp`	The file containing your tests for each data structure (you create).

Your task is to write your own code for the public functions found in the provided header files. Please note that some functions are already implemented in the header files.

Testing your code is essential and may require more time than writing the classes. Be sure to test combinations of functions.

Additional Details

- You may add private members as needed.
- You may add public functions strictly for debugging purposes. These must be removed before submitting.
- You may only submit source files specified in the instructions.
- All source code must include the following comments
  - Header comment that includes name and RedID (See appendix)
  - Function comments for every function in the `.cpp` files (See appendix)
- You are expected to write all the code yourself, you may not use the Standard Library API for any containers.
- Your code should handle all error conditions gracefully. It should never crash.
  - Follow the comments which specify what exceptions to throw.
- Your code should not have any memory leaks.
- Your HashTable implementation **must** use chaining, and use the LinkedList class to do so.
- Your code must compile and run correctly on Gradescope to receive any credit.
- You MUST write and submit thorough test cases for EACH data structure submitted.

Due Date

Your project is due on **Thursday, May 2nd, 11:59 p.m.**. Note that this is the last day of our class for the semester. This date was selected to give you as much time as possible, not to increase your burden at the end of the semester.

Opportunity for Extra Credit

Start early and complete your project early. Why? Early submissions are eligible for extra credit points!

- +10 points if turned in by April 30th, 11:59 p.m.
- +5 points if turned in by May 1st, 11:59 p.m.

Late Program

No late programs will be accepted.

Turning in Program 4

To submit program 4, you must upload the following C++ files to your `Gradescope` account through Canvas. The automated grading platform will run.

You're limited to **10** submissions on Gradescope, so ensure that you test your code thoroughly.

```
BinarySearchTree.h
LinkedList.h
HashTable.h
BinarySearchTree.cpp
LinkedList.cpp
HashTable.cpp
test.cpp
```

Hints & Notes

Binary Search Tree

Note: The function `height` has no parameters and returns the height of the tree which is defined as the maximum of the heights of all nodes. The height of a node is the number of edges between it and the root node.

Note: The function `size` has no parameters and returns the number of nodes in the tree.

Note: The function `remove` should use the successor to replace a node with two child nodes.

Note: The function `makeEmpty` removes all the nodes from the tree and returns (deallocates) the memory used by the nodes for reuse. The `makeEmpty` function ultimately sets the root pointer value to NULL.

Hint: Consider maintaining a private `size` variable that is increased by insertion and decreased by deletion, and whose value is returned by the `size()` function. Alternatively, a traversal of the tree when the function is called could also calculate the size of the tree. Similar techniques could determine the height of the tree.

Hint: Do these tasks a few members at a time. Compile **and test** after doing each group of a few members. You will be glad you did it this way!

Hint: Before you write the operator =, and the copy constructor, note that their jobs have a common task: duplicating another tree. Write a `copyTree` function that abstracts out the common task of these operations, then use it to implement these functions.

Hint: The makeEmpty and the destructor have a common tree task.

Code Format and Comments

- Proper indentation:
  - Consistent use of spaces or tabs for indentation (typically 2 or 4 spaces).
  - Nested code blocks should be indented accordingly.
- Consistent use of braces:
  - Braces should be used consistently for code blocks, including if statements, loops, and functions.
- Descriptive variable and function names:
  - Variable and function names should be meaningful and descriptive of their purpose.
  - Use of camelCase for naming.
- Consistent naming conventions:
  - Follow consistent naming conventions for variables, functions, classes, and other identifiers.
- Proper spacing:
  - Use spaces around operators for readability.
  - Use consistent spacing within expressions and statements.
- Use of appropriate data types:
  - Choose appropriate data types for variables and functions based on their usage and requirements. (We are not concerned about memory usage, so do not worry too much about this one.)
- Avoidance of magic numbers and hardcoded values:
  - Use named constants instead of hardcoded values.
  - Magic numbers should be avoided or properly explained with comments.
- Proper use of comments:
  - Every file should have a header comment with your name and RedID. (Look at example provided in the appendix.)
  - Every function should have a leading comment. (Look at example provided in the appendix.)
  - Include comments to explain complex logic, algorithms, or non-trivial code blocks.
  - Comments should be clear and concise, adding value to the code understanding.

Cheating Policy

There is a zero tolerance policy on cheating in this course. You are expected to complete all programming assignments on your own. Collaboration with other students in the course is not permitted. You may discuss ideas or solutions in general terms with other students, but you must not exchange code. During the grading process I will examine your code carefully. Anyone caught cheating on a programming assignment (or on an exam) will receive an ”F” in the course, and a referral to Judicial Procedures.

Appendix

```cpp
/**
* @file sample.cpp
* @author <Your Name> <Your RedID>
* @brief This is a sample program to show how to comment
* @date YYYY-MM-DD
*/
#include <stdio.h>

int main() {
    printf("Hello, world!");
    return 0;
}
```

Listing 1: Example Header Comment

```cpp
/**
* Returns the Object at the specified position in this list
* @param index
* @return Object
* @throws invalid_argument if index out of range
*/
int get(int index) {
    function definition...
}
```

Listing 2: Example Function Comment