KIT205 Data Structures and Algorithms

Assignment 2: Data Structures

Due: 24th April (Friday, Week 8) at 11:55pm

Introduction

After some recent experience with online learning, the University wishes to expand its online offerings and introduce more Massive Open Online Courses (MOOCs). You have been asked to develop some software to manage student enrolments. You decide to develop a prototype solution to test the performance of different data structures and algorithms.

The University will initially only offer a few courses, but that may expand significantly in the future. More importantly, each course may have many thousands of students – perhaps even hundreds of thousands. Your software needs to provide the following functions:

1. Add course
2. Remove course
3. Enrol student
4. Un-enrol student
5. Print an ordered summary of courses and the number of students enrolled in each course
6. Print an ordered list of students enrolled in a course
7. Print an ordered list of courses that a given student is enrolled in

The number of courses is small, but may grow significantly, so a scalable solution is required. Many functions require courses to be printed in (alphabetical) order, so an ordered data structure will be a good choice. Therefore, you decide to use a simple linked list to store the courses, with course *stored* in-order.

The number of students per course will potentially be very large and, for quick access and printing, sorted student ids are also required. Initially, you decide to test performance of a binary search tree to store student enrolments, but you would like to test an AVL tree as well. Note that, for this preliminary testing, only the student ids will be stored (as long ints).

Assignment Specification – Part A (80%)

For this part of the assignment you will implement a prototype that uses an ordered linked list to store courses, with each course having a name and a BST of students. You must use the BST and linked list code developed in the tutorials, however the data structures will be modified for the new data (and functions will also require minor modifications to accommodate these changes). The following definitions **MUST** be used:

```
typedef struct bstNode {
	long id;
	struct bstNode *left;
	struct bstNode *right;
} *BSTNodePtr;

typedef struct bst {
	BSTNodePtr root;
} BST;

typedef char* String;

typedef struct listNode{
	String course;
	BST students;
	struct listNode *next;
} *CourseNodePtr;

typedef struct list {
	CourseNodePtr head;
} CourseList;
```

The `CourseNodePtr` and `CourseList` definitions and (modified) linked list functions must be placed in files `list.h` and `list.c`. The `BSTNodePtr` and `BST` definitions and modified BST functions must be placed in files `bst.h` and `bst.c`.

All remaining code should be placed in a file called `main.c` that contains the main function and program logic. This file will contain separate functions for each of the seven operations listed in the introduction, as well as an eighth function to handle termination. Other functions may be added if required.

Program I/O

All interactions with the program will be via the console. Operations 1-7 will be selected by typing 1-7 at the command prompt. Quitting the application will be selected by typing 0. For example, the following input sequence would create a course called “abc123” and enrol a student with id “123456” in that course then quit the application:

```
1
abc123
3
abc123
123456
0
```

Note that this sequence shows the input only, not the program response (if any). You are free to add prompts to make the application more user friendly, but this will not be assessed (although it may be useful).

Program output in response to operations 5-7, should be as minimal as possible. You may print a header if you wish, but this should be followed by one record per line with spaces separating data. For example, in response to operation 5, the output might be:

```
Current enrolments:
abc123 32
def123 0
def456 10236
```

I/O Restrictions

You may assume that all input will always be in the correct format and contain no logical errors.

- Commands will always be in the range 0-7
- Course names will always be strings less than 100 characters long and may contain any alpha-numeric characters (no spaces)
- Student ids will always be positive integers in the range 0-999999
- The user will never attempt to add a student to a non-existent course
- The user will never attempt to print data for a non-existent course
- The user will never attempt to print data for a non-existent student

Note: Courses that contain enrolled students may be removed, in which case student data for that course should also be removed. Courses with zero students should not be automatically removed.

Memory Management

Course names should be stored in appropriately size dynamically allocated memory. Names will always be less than 100 characters long. For example, the course name “abc123” would be stored in a char string of length 7.

Removing (un-enrolling) a student or removing a course should free all associated dynamically allocated memory. Removing a course should free all memory for the enrolled students as well as the course. The quit function should also free all dynamically allocated memory.

Assignment Specification – Part B (20%)

This part of the assignment should only be attempted once you have a fully implemented **and thoroughly tested** solution to part A. It would be better to submit a complete part A and no part B than to submit a partially complete part A and part B.

The requirements for this part of the assignment are exactly the same as for part A except that an AVL tree must be used to store students, rather than storing them in a BST. The AVL files should be named `avl.h` and `avl.c`. The AVL node definition should be a modified version of the BST node.

Minimal assistance will be provided for this part of the assignment. No assistance at all will be given unless you can demonstrate a fully implemented and thoroughly tested solution to part A.

Testing

It can be very time consuming to thoroughly test a program like this when all input is done manually (imagine testing that your solution can manage 10000 students). A common method of testing code like this is to use input redirection (and possibly output redirection). When using input redirection your code runs without modification, but all input comes from a file instead of from the keyboard.

This facility is provided in Visual Studio through the project properties dialog. For example, to redirect input from a file called "test.txt", you would add:

```
<"$(ProjectDir)test.txt"
```

to *Configuration Properties|Debugging|Command Arguments*. This will be demonstrated in tutorials.

At least one small input test file with sample output will be provided. It is recommended that you construct larger files to fully test your program. As well as larger test files, it would also be wise to construct files that test edge cases.

Assignment Submission

Assignments will be submitted via MyLO (an Assignment 2 dropbox will be created). Submissions should consist of one zipped Visual Studio project for each part of the assignment that you attempt. You should use the following procedure to prepare your submission:

- Make sure that your project has been thoroughly tested
- Choose "Clean Solution" from the "Build" menu in Visual Studio. **This step is very important** as it ensures that the version that the marker runs will be the same as the version that you believe the marker is running.
- Quit Visual Studio and zip your entire project folder
- Upload a copy of the zip file to the MyLO dropbox

History tells us that mistakes frequently happen when following this process, so you should then:

- Unzip the folder to a new location
- Open the project and confirm that it still compiles and runs as expected
    - If not, repeat the process from the start (a common error occurs when copying projects, where the code files you are editing end up existing outside of the project folder structure – and therefore don't get submitted when you zip the folder)

KIT205 Data Structures and Algorithms: Assignment 1 - Data Structures

Synopsis of the task and its context

This is an individual assignment making up 10% of the overall unit assessment. The assessment criteria for this task are:

1. Adapt generic data structures for use in a specific problem
2. Implement generic list data structures and algorithms
3. Implement generic tree data structures and algorithms

A significant and extremely important part of software development is thorough testing. Small programming errors may attract a large penalty if the error is something that should have been picked up in testing! Please try to complete your program early to leave enough time for testing.

Match between learning outcomes and criteria for the task:

Unit learning outcomes	
On successful completion of this unit you will be able to …	*Task criteria:*
1. Transform a real-world problem into a simple abstract form that is suitable for efficient computation	1
2. Implement common data structures and algorithms using a common programming language	1, 2, 3
3. Analyse the theoretical and practical run time and space complexity of computer code in order to select algorithms for specific tasks	-
4. Use common algorithm design strategies to develop new algorithms when there are no pre-existing solutions	-
5. Create diagrams to reason and communicate about data structures and algorithms	-

Criteria	HD (High Distinction)	DN (Distinction)	CR (Credit)	PP (Pass)	NN (Fail)
	You have:	**You have:**	**You have:**	**You have:**	**You have:**
1. Adapt generic data structures for use in a specific problem *(correct .h files and functions calls from main.c)* ***Weighting 25%***	Correct adaption of linked list data structure for storing customers, programs or subscriptions - and - Correct adaption of tree data structure for storing customers, programs or subscriptions - and - Correct use of adapted data structures in main program	Correct adaption of linked list data structure for storing customers, programs or subscriptions - and - Correct adaption of tree data structure for storing customers, programs or subscriptions - and - Partially correct use of adapted data structures in main program		Partially correct adaption of linked list data structure for storing customers, programs or subscriptions - and - Partially correct adaption of tree data structure for storing customers, programs or subscriptions - and - Partially correct use of adapted data structures in main program	Correct code for input loop, but little attempt to adapt data structures for this problem
2. Implement generic list data structures and algorithms *(correct implementation of functions in list.c)* ***Weighting 25%***	Linked list functions implemented correctly, including freeing all memory correctly (including on quit)	Linked list functions implemented correctly, without freeing all memory	Implemented some linked list functions correctly for this problem		Some attempt to get linked list operations working (e.g. tutorial tasks completed)
3. Implement generic tree data structures and algorithms *(correct implementation of functions in bst.c)* ***Weighting 50%***	BST and AVL functions implemented correctly, including freeing all memory correctly (including on quit)	All BST functions implemented correctly - and - Some AVL functions implemented correctly	BST functions implemented correctly, including freeing all memory correctly (including on quit)	Implemented some BST functions correctly for this problem.	Some attempt to get BST operations working (e.g. tutorial tasks completed)