ASSIGNMENT 3: SEARCHING WITH TREES

CS3D5A, Trinity College Dublin

Deadline: 13:00 17/11/2022

Grading: The assignment will be graded on Submitty. If your code does not work on Submitty, you will be able to do an optional demo during the lab hours of 18/11/2022– note that only code submitted on Submitty will be marked, and that you will only get full marks if your code is working on Submitty.

Questions: You will able to ask questions during the lab hours on 11/11/2022

Submission: Submit via Submitty. Include a separate .c file for each task, and the short assignment report in pdf, word, or text file as instructed below.

Goals:

- Learn how to implement a tree
- Become familiar with Binary Search Trees
- Assess the performance of Binary Search Trees
- Learn to work with multiple code files
- Practice working with legacy code

Task 1 – Binary Search Tree – 8 Marks

Implement a binary search tree using char as the data records. First edit the `TreeNode` struct in the `bst.h` file to create a suitable structure to represent a node.

Next, implement the following functions in `bst_skeleton.c`, and test them from your own separate main file (which should NOT be submitted to submitty):

1. `void tree_insert ( Tree_Node** root, char data );`
   Creates a new node in the tree in the appropriate position for a binary search tree.
2. `Tree_Node* create_bst (char data[]);`
   Creates a new binary search tree representing the data, which is terminated by '\0' (note, you do not need to optimise the tree created)
3. `Tree_Node* tree_search ( Tree_Node* root, char data );`
   If a value exists in the tree, return a pointer to that node, otherwise return NULL.
4. `void tree_print_sorted ( Tree_Node* root );`
   Traverse the tree, printing the data held in every node, in smallest-to-greatest sorted order.
5. `void tree_delete ( Tree_Node* root );`
   Delete every node in the tree without creating a memory leak.

- Submit the edited `bst_skeleton.c` and (edited) `bst.h` on Submitty for task 1 (do NOT submit the file which contains your main).

Task 1 mark allocations	
Using the function above, the string "FLOCCINAUCINIHILIPILIFICATION" can be loaded and printed in sorted order.	4
Accurately reports whether a given letter is in the tree (using the `tree_search` function).	2
Fully delete the tree and free all allocated memory using the `tree_delete` function.	2

Task 2 – A Practical Application – 7 Marks

You have been hired to build a reasonably simple database which stores documents in memory. This database will be used by a small text-based search engine to curate books that users might want to read. The database should store the **name** and **word count** of each book. It should also assign a unique identifier to each book as it is added to the database.

Because users can ask complex queries, the database needs to be extremely fast. Proper choice of data structure will have a big impact on how well the system performs. The developer who was hired before you attempted to build the database using a linked list – a poor design decision by all accounts. Aforementioned developer was summarily sacked.

You, being that much smarter, have opted to use a Binary Search Tree for your database. If it is well implemented then it should perform drastically better than the linked list.

Your employers know what they are looking for and have already provided tools to assess the performance of your implementation. All of the testing code you need is in place. All you need to do is fill out the `bstdb.c` file in the `src` directory. Specifically, you need to write six functions:

1. `int bstdb_init ( void );`
   Allocates any memory that your database will need (e.g. if you need to allocate some initial space for your binary search tree). Should return 0 if initialization fails for some reason

2. `int bstdb_add ( char *name, int word_count );`
   Adds a new book to your binary search tree. Should store the name and word count, then return a unique ID which can be used to retrieve the book later.

3. `int bstdb_get_word_count ( int doc_id );`
   Retrieves the word count of a book from the database using its document ID.

4. `char* bstdb_get_name ( int doc_id );`
   Retrieves the name of a book from the database using its document ID.

5. `void bstdb_stat ( void );`
   Use this function to run some tests of your own. It will be called once by the profiler after it has run its own tests. For example, you might use this function to show that your binary search tree is balanced. The linked list developer tested whether or not the linked list was storing as many books as it claimed. They also computed the average number of node visited when retrieving a result. These values should be printed to the terminal.

6. `void bstdb_quit ( void );`
   Release any dynamically allocated memory that your BST used.

The `bstdb.c` source file for the project has been heavily commented to explain what each function should do. You have also been provided with the previous developer's attempt in the `src/db/listdb.c` file. If you are looking for some form of guidance regarding how to proceed you might want to check those files.

This project is comprised of more than one C file, but it has been supplied as a Visual Studio Code project – so you can try to open the project folder in VSC and press F5.

If VSC fails, there is a Makefile included with the project as well. You can execute the Makefile by opening the project folder in the terminal and typing “make”. The resulting program will be called "task2". **Before you edit any of the files, ensure that you can compile the project,** with Visual Studio Code, the make file or even by command line (`gcc -Wall -g src/task2.c src/bstdb.c src/db/profiler.c src/db/database.c src/db/listdb.c -o task2 -lm`)

When you run the project, the profiler will start trying to put data in your database. It will also time how quickly your database responds to each query and how many erroneous responses it received from you.

The profiler metrics are:

1. **Total Inserts**: Total number of books which should have been loaded into the database
2. **Num Insert Errors**: The number of times your database reported that it could not store a book
3. **Avg, Var Total Insert time**: The average and variance of time taken per-insertion. Insert Time is the total time spent loading the database
4. **Total Title Searches**: The number of searches for book titles that were executed during the test
5. **Num Title Search Errors**: The number of searches for book titles which returned the wrong results
6. **Avg, Var Total Title Search time**: The average and variance of time taken per-search. Total Title Search Time is the total time spent searching for titles.
7. **Total Word Count Searches**: The number of searches for book word counts that were executed during the test
8. **Num Word Count Search Errors**: The number of searches for book word counts which returned the wrong results
9. **Avg, Var Total Word Count Search time**: The average and variance of time taken per-search. Total Word Count Search Time is the total time spent searching for word counts

There is a lot of room for innovation here. Start by implementing the BST in the simplest way you can think of. Once you have your first set of results, start to think about what you might need to do to help it perform quickly.

- Submit the `bstdb.c` file to Submitty, as well as your report.

Task 2 mark allocations	
Implementation of a database which uses a binary search tree and reports no errors when queried by the profiler. **Copy the output in your report**	4
Due consideration given to proper design of the BST to ensure best performance, e.g. what did you do to try and keep the tree balanced? If you're not careful, the linked list might actually perform better than your BST. **Document your choices in your report**	2
Extra testing to ensure that your BST is performing correctly, e.g how many nodes does it need to visit on average before it responds to a query? You should put the code for this in the `stat` function. **Document in your report.**	1