Exercise 1

This exercise has total 25 points. Each point amounts to one percent of the total module mark. You are not allowed to use any external library in your code. We might ask you to explain your code.

All assessment deadlines in this module are strict. Late submissions will get a mark of 0. All submissions must work on the virtual machine as specified.

The exercise consists of two parts: Part 1 and Part 2.

In Part 1, you will be implementing a binary search tree (BST) data structure in C. This part requires you to follow the lecture materials of Week 1 and 2.

Then in Part 2, you will be implementing a multi-threaded version of the above BST for a server system where multiple clients run concurrent BST operations. This part requires you to follow the lecture materials of concurrent programming using pThreads (week 4). You can consider working on Part 2 only after finishing Part 1 and studying concurrent programming concepts.

Part 1: Implementing a binary search tree (BST) in C

[Before starting this exercise, consider having a look at the linkedlist_basic.c source code file which is present inside Modules/Week2 on Canvas.]

Your task is to implement a BST for integers (int). In this exercise, we assume that there are **no duplicate nodes**. In bst.h, there is already a header file that declares the interface of that tree. You must implement the following functions for the tree operations in bst.c file. There is a file test_bst.c which is a simple testbench for testing your BST code. You may edit this file to include more test cases.

Implement the following functions in the bst.c file.

Task 1.1

```
Node* insertNode(Node *root, int value);
```

This function is used to insert a new node in the tree as a leaf node. The member ‘data’ of the new node is assigned the input ‘value’. During the insertion, a traversal starts from the root of the tree. The traversal follows the right branch if the value of the new node is greater than the value of the currently visited node. Otherwise the traversal follows the left branch. For this question we will restrict to the case that all data values are unique (that is to say, there will be no duplicates) for simplicity.

Task 1.2

Node* deleteNode(Node *root, int value);

This function is used to delete an existing node in the tree. For this question we will restrict to the case that all data values in the tree are unique.

Task 1.3

void printSubtree(Node *N);

This function is for Inorder printing of the node values. It prints

i) the values in the left subtree

ii) then the value of the node and

iii) finally, the values in the right subtree.

Thus, the function prints the node values in sorted ascending format.

The function should print one node value per line, for example:

2
3
4
5
Not 2 3 4 5.

Task 1.4

int countNodes(Node *N);

This function returns the number of nodes in the subtree rooted at node N.

Task 1.5

Node* freeSubtree(Node *N);

This function deallocates the entire subtree rooted at node N and then returns a NULL pointer that can be assigned to N. Here you will need to free each node exactly once, so that there are neither double frees nor memory leaks. Note that the tree could be empty, so freeSubtree() should work, but do nothing, as there are no nodes to free. It returns the same pointer which it received.

When using the freeSubtree() function in your test code, use it in this way (example):

root=freeSubtree(root);

if you wish to free the entire tree as an example. This will set the root to NULL, indicating an empty tree.

Task 1.6

int sumSubtree(Node *N);

This function returns the sum of all the nodes. For example, if you pass the root of a BST, the function will return the sum of all nodes that are present in the tree. The algorithm for this function will be somewhat similar to printSubtree() as both functions perform complete tree traversal; printSubtree() prints the node-values whereas sumSubtree() accumulates the node values.

Part 2: Implementing multi-threaded BST for a server system

This part addresses the challenges in concurrent programming. You will be using the BST that you have implemented in Part 1 and then add support for multi-threaded concurrent read/write operations on the BST.

Before you start the Ubuntu virtual machine, please set the number of processors to 2 from

VM VirtualBox Manager-->Settings-->System-->Processor.

After this change, if you run the 'lscpu' command from a terminal, it should show the number of CPUs = 2.

Assume that a server machine manages a BST data structure, and many clients perform operations on the BST concurrently. Each client sends its BST-commands in the form of a command-file to the server.

Valid commands that can be executed by the clients are:

(1) insertNode <some unsigned int value>

The server will insert the node with the specified value in the tree. Note that this operation modifies the BST.

The server also prints a log using printf() in the format of

"[ClientName]insertNode <SomeNumber>\n"

where \n means a new line.

(2) deleteNode <some unsigned int value>

The server will delete the node with the specified value from the tree. Note that this operation modifies the BST.

The server also prints a log using printf() in the format of

"[ClientName]deleteNode <SomeNumber>\n"

where \n means a new line.

(3) countNodes

The server will count the current number of nodes in the BST and print the number on the display. Note that this operation does not modify the BST.

The server also prints a log using printf() in the format of

"[ClientName]countNodes = <SomeNumber>\n"

where \n means a new line.

(4) sumSubtree

The server will compute the sum of all the nodes that are present in the BST and print the sum on the display. Note that this operation does not modify the BST.

The server also prints a log using printf() in the format of
"[ClientName]sumSubtree = <SomeNumber>\n"

where \n means a new line.

The server executes the commands that are present in a command file one-by-one starting from the first line.

Example: Assume that the server has received a file 'client1_commands' from Client1. Let, the file contents be (starting from the beginning of the file):

```
insertNode 19675
…
…
sumSubtree
```

Thus, for Client1 the server will first insert a node with value 19675, then perform the following operations, and finally compute the sum of the tree etc.

The server serves multiple clients concurrently. For each client, the server calls

```
void* ServeClient(char *clientCommands)
```

in a concurrent thread and passes the name of the command-file using the string pointer clientCommands.

Task 2.1
Implement the function in serve_client.c

```
void* ServeClient(char *clientCommands)
```

that reads the file specified by *clientCommands* for commands and executes the BST operations according to the content of the file. Note that multiple clients will be executing BST operations concurrently using threads. So, proper care must be taken to avoid concurrency issues.

Code submission

Use the Makefile to compile your code. There is a script in test.sh, that runs test_bst and compares the output with a reference output.

Use Valgrind to check memory leaks and errors. The Makefile produces test_bst file. The command for checking memory leaks will be valgrind --leak-check=full ./test_bst

For Part 2, put your bst.c and serve_client.c inside the directory part2.

Both directories (part1 and part2) should be inside a folder named with your student ID. Then **compress the folder** to a .zip file and upload that to Canvas as your assignment submission.

You may need to add executable permission to the test.sh file using 'chmod +x test.sh' command.

For example:
A student with student id of 1234567 should submit 1234567.zip and this file should be able to be extracted to a folder with the following structure:

```
1234567/
 |- part1/
 |      ├── bst.c
 |
 |- part2/
        ├── bst.c
        └── serve_client.c
```

Both bst.c files inside part1/ and part2/ should exactly be the same. During our tests, we will choose only one of the two bst.c files.

Any other form of submission will not be accepted.

Marking scheme: You get less points when your solution leaks memory, handles errors incorrectly or crashes for some inputs. Your code must compile on the virtual machine we had recommended. Any code which does not compile on the virtual machine provided will be awarded 0 marks and not be reviewed.

For marking, we will use additional, more advanced, test scripts which will check whether your program satisfies the specification and functions as expected for more test vectors. If the provided test scripts fail, all of the more advanced test scripts are likely to fail as well

- 3 points are given if insertNode() inserts correctly.
- 8 points are given if deleteNode() deletes correctly without causing memory leaks.
- 1 point is given if printSubtree() prints the subtree correctly (i.e in ascending order)
- 1 point is given if sumSubtree() sums the subtree correctly
- 1 point is given if countNodes() returns the number of nodes correctly
- 4 points are given if freeSubtree() deallocates the subtree without causing memory leaks
- 7 points are given if ServeClient() functions correctly

We might ask you to explain your code.

Remarks

- Only return an error when malloc or a similar function fails. Do not set a hard limit for the memory in your code.
- Do not use static array declaration of variable length. E.g., int a[*variable*];
- Do not upload object files or binaries, just the C code.
- Any code which does not compile on the virtual machine using the Makefile provided will be awarded 0 marks and not be reviewed.
- You must not change the compiler options in the Makefile. These options imply that any warnings will be treated as errors, resulting in code that doesn't compile and thus will not be marked.
- For marking we will use additional, more advanced, test scripts which check whether your program satisfies the specification. If the provided test script fails, all the more advanced test scripts are likely to fail as well.

- There must be no memory leaks in your tree implementation.

Useful Links

Using make and writing Makefiles

https://www.cs.swarthmore.edu/~newhall/unixhelp/howto_makefiles.html

Tips: How does the tests for this assignment work?

To test your code just run $ make test

Part1:
For the sample test_bst.c file provided by us, the correct solution is provided in sol.txt file. When you perform a testing, the test script test.sh will run your code and write the output values to the file out.txt. Finally, the test script will compare out.txt with sol.txt to see if both match or not. If they do not match, then that means your code is not producing the correct output. If the two files match, then it means that your program is '*perhaps*' correct for the provided inputs. However, this simple testing does not guarantee that for more advanced test vectors, your code will function correctly. You should consider more test cases.

Part2:
In test_bst.c, it will read files defined in *client_names*, then execute several clients in parallel. In the end, this should result in a tree only having one node, i.e., the number of nodes should be 1. The program checks if the tree is as expected in the end to determine if your code pass this primary test.
If your program does not handle concurrency very well, there is a chance that it may crash or get unexpected result in this test.

You should also check log_1 to see if there are any concurrency problem.

Be aware, these are only primary tests, passing these tests does not mean your code does not contain any bug. More extensive test cases will be used when grading this assignment.