Assignment 4

Version 1.04 (last update: Oct. 20, 16:00)
Changes highlighted in yellow
Due date: Tue, Oct 27, 11:59 PM

Summary and Purpose

For this assignment, you will be writing a collection of C functions that operate on ordered binary trees. You will use these to understand the underlying properties of ordered binary trees, their strengths and weaknesses. In particular, you can compare the performance of the trees in this assignment to the arrays of Assignment 2, and the lists of Assignment 3.

Deliverables

You will be submitting:

1) A file called **`tree.h`** that contains your function prototypes (see below).
2) A file called **`tree.c`** that contains your function definitions.
3) A **`makefile`** that contains the instructions to compile your code.

You will submit all of your work via git to the School's gitlab server. (As per instructions in the labs.)

This is an individual assignment. Any evidence of code sharing will be investigated and, if appropriate adjudicated using the University's Academic Integrity rules.

Structures for your assignment

You will be working with variables having the following structures which you must declare in your header file.

```
struct Node
{
  void *data;
  struct Node *lt;
  struct Node *gte;
};
```

This structure represents a node in the tree (our third data structure used in this course). **`data`** is a pointer to the data stored at this node, while **`lt`** and **`gte`** are pointers to nodes that are less than, and greater than or equal to the current node according to the **`compar`** function (or **`NULL`** if the node does not have lower or higher ordered nodes).

Additionally, you will be using the following structure to measure the performance of your code and count the number of memory read, memory write, malloc and free operations.

```
struct Performance
{
  unsigned int reads;
  unsigned int writes;
  unsigned int mallocs;
  unsigned int frees;
};
```

Basic function prototypes and descriptions for your assignment

```
struct Performance *newPerformance();
```

This function will allocate sufficient memory for a **Performance** structure, set **reads**, **writes**, **mallocs**, and **frees** to zero (yes, I realize there is technically one malloc in this function) and return the address of the structure. Your function should print an error message to the standard error stream and **exit** if the **malloc** function fails.

```
void attachNode( struct Performance *performance, struct Node **node_ptr,
                 void *src, unsigned int width );
```

(Add a node to a pointer.) This function will **malloc** a new **struct Node** structure, **malloc** **width** bytes of data and save the address in **data**, copy **width** bytes of data from the parameter **src** to the address **data** in the new **Node** structure. It will set the lower and higher pointers in the structure to NULL. It should copy the address of the new structure into pointer pointed to by **node_ptr**. If the **malloc** fails, it should print an error message to the standard error stream and **exit**. **mallocs** and **writes** in the **performance** structure should both be incremented by 1.

```
int comparNode( struct Performance *performance, struct Node **node_ptr,
                int (*compar)(const void *, const void *), void *target );
```

This function should return the value returned by the function pointed to by the **compar** function pointer, when applied to the data stored at **target** and the **data** variable in the structure at the address that **node_ptr** points to (in that order). **reads** in the **performance** structure should be incremented by 1.

```
struct Node **next( struct Performance *performance, struct Node **node_ptr,
                    int direction );
```

(Determine the next node in the tree to visit.) If the tree is empty it should print an error message to the standard error stream and exit. Otherwise, this function should return the address of the **lt** node pointer (double pointer), or the address of the **gte** node pointer, depending on whether direction is less than zero, or greater than or equal to zero. **reads** in the **performance** structure should be incremented by 1.

```
void readNode( struct Performance *performance, struct Node **node_ptr, void
*dest, unsigned int width );
```

(Copy data from a node in the tree into **dest**.) If the tree is empty it should print an error message to the standard error stream and **exit**. Otherwise, this function will copy **width** bytes of data from the **data** pointer in the node pointed to by the pointer pointed to by **node_ptr**, into **dest**. **reads** in the **performance** structure should be incremented by 1.

```
void detachNode( struct Performance *performance, struct Node **node_ptr );
```

(Remove an item from a tree consisting of only one node.) If the tree is empty it should print an error message to the standard error stream and **exit**. It should update the pointer pointed to by **node_ptr** to be **NULL**, and free the node structure that used to be in the tree. **frees** in the **performance** structure should ~~both~~ be incremented by 1.

```
int isEmpty( struct Performance *performance, struct Node **node_ptr );
```

Check if the pointer pointed to by **node_ptr** is **NULL**.

Derived function prototypes and descriptions for your assignment

The following functions should all be implemented by calling the "Basic" functions, above. Most importantly, you should not be interacting with Node structures or Node pointers directly, only by calling the Basic functions. In most of these functions you will use a temporary **struct Tree**, initialized with the same values pointed to by the passed **tree** parameter. Then you can update the **root** pointer inside this temporary structure while not damaging the original tree structure.

```
void addItem( struct Performance *performance, struct Node **node_ptr,
              int (*compar)(const void *, const void *),
              void *src, unsigned int width );
```

(Add an item to the tree at the appropriate spot.) This function will use a loop that moves through the nodes of the tree, using the **comparNode** function and the **next** function when it reaches an empty pointer it will use the **attachNode** function to add a new node. This function should use the **performance** updates and error handling of the basic functions.

```
void freeTree( struct Performance *performance, struct Node **node_ptr )
```

This function will remove all the items from the tree using **next**, and **detachNode**. This function should use the **performance** updates and error handling of the basic functions.

~~void readItem(struct Performance *performance, struct Node **node_ptr, void *target, void *dest, unsigned int width);~~

The Last 20%

The above, constitutes 80% of the assignment. If you complete it, you can get a grade up to 80% (Good). The rest of the assignment is more challenging and will allow you to get a grade of 80-90% (Excellent) or 90-100% (Outstanding). Make sure you complete the first part well, before proceeding to the following additional part.

Write the following functions:

```
int searchItem( struct Performance *performance, struct Node **node_ptr,
                int (*compar)(const void *, const void *),
                void *target, unsigned int width );
```

This function will search for an item in the tree that compares with a value of zero from the **`compar`** function. It will use **`target`** as both the search term and the output. This function will use **`comparNode`**, **`readNode`** and **`next`**. If a node is found, it will return 1, otherwise it will return 0.

You can write additional helper functions as necessary to make sure your code is modular, readable, and easy to modify.

Header File

Use the #ifndef…#define…#endif construct (Lecture 02) in your header file to prevent problems if your header file is included multiple times.

Testing

You are responsible for testing your code to make sure that it works as required. The CourseLink web-site will contain some test programs to get you started. However, we will use a different set of test programs to grade your code, so you need to make sure that your code performs according to the instructions above by writing more test code.

Your assignment will be tested on the standard SoCS Virtualbox VM (http://socs.uoguelph.ca/SoCSVM.zip) which will be run using the Oracle Virtualbox software (https://www.virtualbox.org/wiki/Downloads). If you are developing in a different environment, you will need to allow yourself enough time to test and debug your code on the target machine. We will NOT test your code on YOUR machine/environment.

Full instructions for using the SoCS Virtualbox VM can be found at: https://wiki.socs.uoguelph.ca/students/socsvm.

Makefile

You will create a makefile that supports the following targets:

`all:` this target should generate tree.o.

All programs and .o files must be compiled with the –std**`=c99 -Wall -pedantic`** options and compile without any errors or warning.

`clean:` this target should delete all **`.o`** files.

`tree.o:` this target should create the object file, **`tree.o`**, by compiling the **`tree.c`** file.

All compilations and linking must be done with the **`-Wall -pedantic -std=c99`** flags and compile and link **without any warnings or errors**.

Git

You must submit your .c, .h and makefile using git to the School's git server. Only code submitted to the server will be graded. Do ***not*** e-mail your assignment to the instructor. We will only grade one submission; we will only grade the last submission that you make to the server and apply any late penalty based on the last submission. So once your code is complete and you have tested it and you are ready to have it graded make sure to commit and push all of your changes to the server, and then do not make any more changes to the A2 files on the server.

Academic Integrity

Throughout the entire time that you are working on this assignment. You must not look at another student's code, now allow your code to be accessible to any other student. You can share additional test cases (beyond those supplied by the instructor) or discuss what the correct outputs of the test programs should be, but do not share ANY code with your classmates.

Also, do your own work, do not hire someone to do the work for you.

Grading Rubric

newPerformance	1
attachNode	3
comparNode	2
next	2
readNode	2
detachNode	2
addItem	2
freeTree	2
style	2
makefile	2
searchItem	5
Total	25

Ask Questions

The instructions above are intended to be as complete and clear as possible. However, it is YOUR responsibility to resolve any ambiguities or confusion about the instructions by asking questions in class, via the discussion forums, or by e-mailing the course e-mail.