Assignment 4

Version 1.01 (last update: Oct. 21, 16:00)

Changes highlighted in yellow

Due date: Tue, Oct 26, 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 (do not include the instructor's functions from **`memsys`** or a **`main`**).
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
{
  int data;
  int lt;
  int gte;
};
```

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

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

```
struct Tree
{
  unsigned int width;
  int root;
};
```

Malloc vs memmalloc

Unless otherwise noted, you should use **memmalloc** to request all the memory required for the functions below from the **memsys** system. All **malloc** and **memmalloc** calls should be **free/memfree**'d at the appropriate time, and definitely before the program ends.

Functions on **Node**s

```
void attachNode( struct memsys *memsys, int *node_ptr,
                 void *src, unsigned int width );
```

(Create a node; store its address.) This function will **memmalloc** a new **struct Node** structure, **memmalloc 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 addresses in the structure to **MEMNULL**. It should copy the new **Node** structure into **memsys**. It should copy the address of the new structure into the integer pointed to by **node_ptr**. (You may assume that the integer originally pointed to by **node_ptr** was **MEMNULL**.) If either **memmalloc** fails, it should print an error message to the standard error stream and **exit**.

```
void attachChild( struct memsys *memsys, int *node_ptr,
                  void *src, unsigned int width, int direction )
```

(Attach a child node to the parent.) This function should retrieve the node stored at the address pointed to by **node_ptr**. If direction is less than (greater or equal to) zero, it should use **attachNode** to create a new node and update the **lt** (**gte**) variable in the original node. Finally it should update the original node in **memsys**.

```
int comparNode( struct memsys *memsys, int *node_ptr,
                int (*compar)(const void *, const void *), void *target,
                unsigned int width );
```

(Compare data in a variable to data in a node.) 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 in **memsys** at the address given by the **data** variable in the structure at the address that **node_ptr** points to (in that order). You will need to use **malloc** and **free** to allocate and deallocate **width** bytes of memory to temporarily store the **data** from the nodes in **memsys**.

```
int next( struct memsys *memsys, int *node_ptr, int direction );
```

(Determine the next node in the tree to visit.) This function should return the address of the **lt** node, or the address of the **gte** node for the node whose address is pointed to by **node_ptr**, depending on whether **direction** is less than zero, or greater than or equal to zero.

```
void readNode( struct memsys *memsys, int *node_ptr, void *dest,
               unsigned int width );
```

(Copy data from a node in the tree into **dest**.) If **node_ptr** 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** address in the node whose address is pointed to by **node_ptr**, into **dest**.

```
void detachNode( struct memsys *memsys, int *node_ptr );
```

(Remove a leaf node from the tree.) If **node_ptr** is empty, it should print an error message to the standard error stream and **exit**. It should update the integer pointed to by **node_ptr** to be **MEMNULL**, and **memfree** the node structure and its **data** that used to be in the tree. If the node has child nodes, they can be ignored and will be "lost" in **memsys**. In other words, this function should never be called on a node with children.

```
void freeNodes( struct memsys *memsys, int *node_ptr );
```

(Free all the nodes including and below **node_ptr**.) This function should identify the two children of the node pointed to by **node_ptr** using the **next** function. It should call **freeNodes** recursively on each child and finally **detachNode** on the original node. If any nodes are missing (as indicated by **MEMNULL**), then they should not be freed or detached.

Functions on **Tree**s (a.k.a. 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.

```
struct Tree *newTree( struct memsys *memsys, unsigned int width );
```

This function should use a regular **malloc** to allocate memory for the **struct Tree**. If the **malloc** command fails, it should print an error to **stderr** and **exit**. The list structure should be initialized with the given **width** and a **root** equal to **MEMNULL**. This function should return a pointer to the structure.

```
void freeTree( struct memsys *memsys, struct Tree *tree );
```

(Free the tree.) This function should call **freeNodes** on the **root** of the tree and **free** the **tree** structure itself.

```
void addItem( struct memsys *memsys, struct Tree *tree,
              int (*compar)(const void *, const void *),
              void *src );
```

(Add an item to the tree at the appropriate spot.) If the tree is empty this function will call **attachNode** to add a **root** node to the tree. Otherwise, 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 go back one node (hint use two variables to store the original and new addresses of the next function), and use the **attachChild** function to add a new node.

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 function:

```
int searchItem( struct memsys *memsys, struct Tree *tree,
                int (*compar)(const void *, const void *),
                void *target );
```

This function will search for a **`Node`** in the tree whose **`data`** generates a value of zero from the **`compar`** function when matched against the data at **`target`** by performing a binary search in the tree. Once the target node is found, its **`data`** will be copied to **`target`** (replacing the original search term). 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 **`clang`** and 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

attachNode	1
attachChild	2
comparNode	2
next	1
readNode	2
detachNode	2
freeNode	2
newTree	1
freeTree	1
addItem	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.