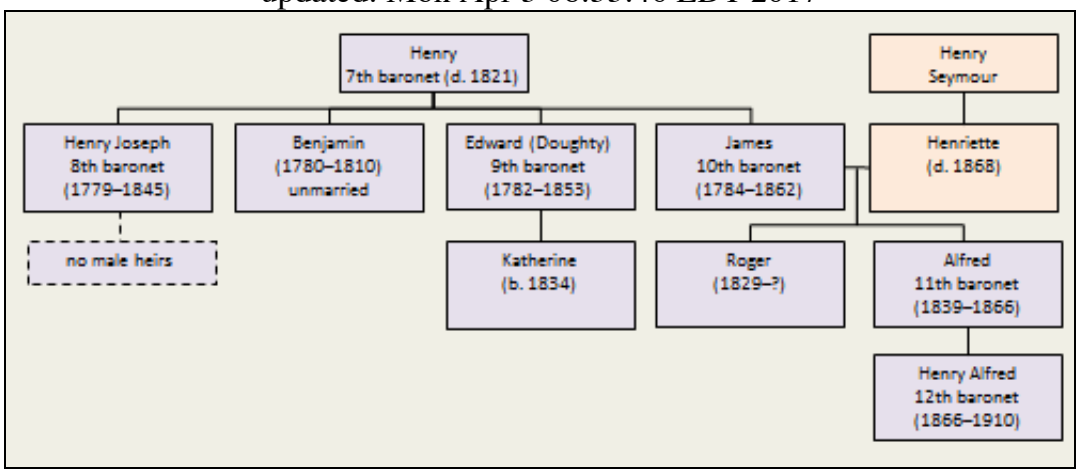
# Project 2 - Offspring

### Submission Due: 11:59PM, July 29, 2017

updated: Mon Apr 3 08:33:46 EDT 2017




**Section Links**



There are two ways to look at a family tree. One is to start with a person and identify their mother and father, and then identify their mother's mother and father and father's mother and father. A binary tree works for this approach, which details a person's lineage going back in time.

The other way is to start with a 'deep' ancestor and work forward in time. This approach leads to Biblical-style descriptions such as 'James begat Sam, Bob and Mary. Sam begat Jill and Tony. Bob begat ...' and so on through all the offspring, or descendants, in the tree. Because a person may have many children, a binary tree will not work.

---

# Assignment

You will write a program named `offspring`, which builds a descendant tree of the generations of offspring from one person. The program takes one optional command line argument. If there is an argument, it reads the file to load an *offspring tree* based on the file's content. Otherwise, it begins running with an empty (`NULL`) tree.

The program issues a prompt and waits for *user commands* to process. Here are the commands with a brief synopsis:

- `add parent-name, child-name` # create a parent-child relation.
- `find [name]` # find and print the name and its children list.

- `print [name]` # print a breadth first traversal of all the offspring from the named person down.
- `size [name]` # count of all members in the [sub]tree.
- `height [name]` # print the height of [sub]tree.
- `init` # delete the current tree and start with an empty tree.
- `help` # print information on available commands.
- `quit` # delete the current tree and exit.

## An N-ary Tree

An N-ary tree will support an arbitrary number of children, possibly zero. The node for an N-ary tree has the following form and content:

```
typedef
struct NTree_S {
   char * name;                     ///< name of the person
   struct NTree_S ** children;      ///< dynamic collection of children
   size_t child_count;              ///< number of children
} NTree;
```

It is necessary to create and initialize an instance of one of these structures for each person in the family tree.

*You may use this structure as-is or adapt it to suit your detailed design.*

Here is an outline of *a subset of the operations* needed for the manipulation of an N-ary tree and its component nodes:

- `create_node( name)` returns a node pointer or `NULL` if it fails.
- `destroy_tree( tree)` frees all memory of the tree and its offspring.
- `find_node( tree, name)` returns the node pointer or `NULL`.
- `print_tree( tree, name)` to `stdout`.
- `add_child( tree, parent-name, child-name )` finds the named parent, adds the child, and returns the possibly revised tree node pointer with the child added to the parent's set of children.

---

# Input Data Formats

## Command Inputs

The commands which take arguments, `add`, `find`, `print`, `size`, and `height`, will use the *comma* character, (`,`) as a field separator.

Below is the general pattern for interactive commands. The square brackets mean that an argument is optional for the command.

```
CommandName  [argument1  [, argument2 ...]]
```

If there is only one argument, then there are no commas on the line. If there are two arguments, then there is one comma on the line. And so forth.

Note that there is no comma between the CommandName and the first argument. There is only whitespace separating these elements.

There may be one or more whitespace characters separating these tokens. *Note also that there may be one or more whitespace characters separating tokens and commas*. The program will have to trim off any leading and trailing whitespace from the fields.

*The values of each argument may be composed of more than one word*. For example, one argument could be "John Doe", and there could be values with additional names and titles as well.

Valid CommandName values with their arguments are:

```
add    parent name , child name
find   [name]
print  [name]
size   [name]
height [name]
help
init
quit
```

*The maximum length of command input is 1024 characters*.

# File Data

The offspring file has a format also uses the *comma* character as a field separator. Each file entry line follows one of these patterns:

```
parent name
parent name , child name
parent name , child name [ , child2 name ...]
```

Here are some examples of file content. One has no offspring; the second has one child, and the third has three children. The fourth line is an error.

```
John Smith
John Smith, Ralph , Linda
Ralph , Mary , Emily, Joseph John Smith
Fred , Bill
```

In the example above, the first entry in the file will create the *root* of the offspring tree. The second line will add Ralph and Linda as offspring of John Smith. The third line will add Mary, Emily and Joseph John Smith

as offspring of Ralph.

The fourth line will be an error because Fred is not anywhere in the tree, and Bill is not the root of the tree. The program must report this error and continue processing lines after it in the file.

Note: Unlike the `add` command, the lines in the file can have more than one child. Also, there may be zero or more whitespace characters separating the names and the commas.

*The maximum length of any line in an input file is 1024 characters.*

## Provided Materials

You have been provided with some materials for this assignment. The get command below delivers the provided materials to the directory in which you run it.

```
$ get csci243 project2
```

Here are the provided items.

- `gitignore`: This file should be edited and then renamed to `.gitignore` so that the version control will use it. Remember to use `git` to set up your project folder. Edit the provided `gitignore` file to add the files that `git` should ignore and not control.
- `file2.txt`: This is an example offspring file. You should also make up your own files to test the various cases of correct and incorrect input files.
- `trimit.h`: This is the interface for a *string trimmer* function that was downloaded from the web. You can use this to trim leading and trailing *whitespace* from the input strings.
- `trimit.c`: This utility file implements the *string trimmer*.
- `TreeStructExample.txt`: This is a *starter structure* that you should take and modify to suit your design for the N-ary tree.

---

# Outputs

## Printed Outputs

There are a number of examples in this document. Please refer to these for what the printed output should look like.

The file ExampleSession.txt shows the interaction and output of an example session using the file provided file file2.txt.

## Error Outputs

There are examples of error output in this document. Each error message must start with the string "error:" at

the start of a new line.

Following that string, the error message should report the problem as shown in the examples below.

```
$ offspring some-file
error: 'Fred' is not in the tree and 'Bill' is not the root.
offspring>

$ offspring some-file
...
offspring> add Betty, Bam Bam
error: 'Bam Bam' is already a child of 'Betty'.
offspring>
offspring> init
offspring> add Wilma   ,  Pebbles
offspring> add Bing   , Betty Boop
error: 'Bing' is not in the tree, and 'Betty Boop' is not the root.
offspring> print
Wilma had Pebbles.
```

---

# Functional Requirements

When `offspring` runs, it checks whether there is a file name on the command line. If there is one, it reads the file content into memory as the offspring record. Then it enters a command loop to receive and handle user commands to query or modify the offspring record.

The functional requirements are organized into:

- Program Invocation -- How the program runs;
- Program Operations -- What the program does; and
- Helpful Tools -- What tools support and enable implementation.

---

## Program Invocation -- How the Program Runs

For the examples below, the `$` prompt represents the shell prompt such as that used by the `bash` shell.

### Command Line Argument and File Rules

*The `offspring-file` name, if present, can be a relative or absolute file path.*

The file content will either be empty or contain one or more lines of valid offspring information.

If there is no file name on the command line, the program starts with an empty offspring record. **The program must not prompt for a file name**.

## Running without an `offspring-file`

When started with no file, the program simply presents its command prompt, which is shown in this example.

```
$ offspring
offspring>
```

The program now waits for user input commands.

## Running with an `offspring-file`

If the file does not exist or cannot be opened, the program will produce the error message shown below reporting the problem to `stderr` and then begin accepting commands.

```
$ offspring no-such-file
error: could not open file 'no-such-file'
offspring>
```

If the file exists and can be opened, the program will read the file, build the offspring tree from that, and issue the prompt to read commands.

If there are errors reading individual lines of the file, report each error to `stderr` and continue processing the rest of the file.

## Interacting with the Command Line

This section describes user interaction for each command.

- `add parent name , child name`

  Search for the parent and add the child to their offspring. If the arguments are missing, print a usage message and reissue the command prompt.

- `find [name]`

  Search from the tree root for the name, and print the name and its children list. If the name is not given, substitute the empty string. If the name is not found, report that it was not in the tree.

- `print [name]`

  Print a breadth first traversal of all the offspring from the named person down. If the name is not given, use the name of the root person.

- `size [name]`

  Compute and print the count of all members in the tree from the named person down. If the name is not given, use the name of the root person. If the name is not found, print the size of a non-existent tree as the value `0`.

- `height [name]`

  Compute and print the height of the tree from the named person down. If the name is not given, use the name of the root person. The height of a single node tree is `0`, and a tree with one parent and one child is height `1`. If the name is not found, print the height of a non-existent tree as the value `-1`.

- `init`

  Delete the current tree and re-initialize the offspring tree as an empty tree.

- `help`

  Print information on available commands.

- `quit`

  Delete the current tree, clean up all dynamic memory and exit.

---

# Program Operations -- What the Program Does

## Adding a Person to the Tree

**pre-condition**: The names passed into the add function must be non-null and non-empty.

Adding a person involves creating a tree node, populating it, and installing the node into the parent's collection of children. Adding a person adds a named child as the offspring of a named parent.

Adding a person to the tree has several cases.

If the tree is empty, then add the parent as the root of the tree, and add the child as the child of the parent.

If the parent is found in the tree, then it adds the child as an offspring provided a same-named child is not already a child of the parent. It is an error if the child is already in the tree as a child of the parent.

If the parent is not found in the tree, then it has to check whether the named child is the root of the tree. In that case, the meaning is to add the parent as a **new root** of the tree, whose current root is the named child.

If the parent is not found in the tree, and the named child is not the root of the tree, then the command is an error; the program can add a parent to a child node only if the child of that parent is already the root of the tree.

Finally, it is possible to create a one-person tree only from a file. If a line of the file has only one name on it, and the tree is empty, then that name becomes the person in the tree root node. If the tree already has people in it, then it is an error. Consequently, the first line of the file is the only line which can have a single name on it; all other lines would have to have two or more names listed.

## Finding a Person in the Tree

Finding a person by name is the tree search function. If the tree is empty, the function must return `NULL`. Otherwise, search the tree to find the named person. Return the pointer to the tree node if the person is found, or return `NULL` if not found.

The algorithm for finding a person is a *breadth first search*. If the person's name does not match the current node, then check all the children of that node before progressing to the next level down the tree. This way, the search will find the earliest ancestor whose name matches the search goal. If it fails to find the name after searching all levels, the search must return `NULL`.

## Printing the Tree

**pre-condition**: If present, the name passed into the print function must be non-null and non-empty.

Printing the tree will perform a *breadth first traversal* of the tree. If there is a name on the command line, it will first find the person with the given name before it begins. If the name is not found, it prints an error message to `stderr`; here is an example:

```
offspring> print nonsense
error: 'nonsense' not found
```

Given this offspring file:

```
John Smith
John Smith, Ralph , Linda
Ralph , Mary , Emily, Joseph John Smith
Fred , Bill
Joseph John Smith, Phineas
```

Here is an example of the breadth first output of the command `print`:

```
John Smith had Ralph and Linda.
Ralph had Mary, Emily and Joseph John Smith.
Linda had no offspring.
Mary had no offspring.
Emily had no offspring.
Joseph John Smith had Phineas.
Phineas had no offspring.
```

The string " had " is between the person and their list of children. The string " no offspring" appears if a person had no children. The list of children printed uses comma separation between each child and inserts the string " and " between the last two children. All children in this comma-separated list appear on the same line of output, and the line ends with a period and a newline.

If there is no name on the command line, the print begins at the root of the tree. If the tree is empty, print an error message and issue another prompt. Since the tree is empty, this message may come out containing the text "`(null)`".

## Getting the Size of the Tree

The size function takes an optional argument as a starting name. If there is a name given, search the tree for that name and compute the size of the sub-tree that is rooted at that node. If the name is not found, print the size of an empty tree. Otherwise, if the name is not given, compute the size of the tree from the root node.

### Getting the Height of the Tree

The height function takes an optional argument as a starting name. If there is a name given, search the tree for that name and compute the height of the sub-tree that is rooted at that node. Otherwise, compute the height of the tree from the root node.

### Deleting the Tree

The `init` command must delete the tree. Deleting the tree must start at the root and free all memory associated with the current family tree and return the program to an empty state.

After initialization deletes the tree, adding a person will create a brand new tree of offspring.

---

## Helpful Tools

This section describes what library and system tools will support and enable the implementation.

### Queue Data Structures

Since there are two breadth first operations in this program, it will be useful to have a *queue* data structure to manage things during the breadth first activities. Here is an outline of a breadth first processing pattern:

```
search( node, goal)
        queue = makequeue()
        enqueue( queue, node)
        while ( size( queue) > 0 )
                node = front( queue)
                dequeue( queue)
                if ( ... node *** goal ... )
                        return node
                else
                        for child in node.children
                                enqueue( queue, child)
                ...
        return ...
```

*** Note that, like Java, it will be necessary to define some sort of *comparator* that the queue can use to determine the equivalence between a node and the goal.

### Tokenizing Lines of Commas and Spaces

Since the input of both the file and the command line will have comma-separated fields, it will be useful to have some sort of utility to process strings with comma separations. The `strtok` function is useful for

designing a function that can build a *field list* from comma-separated fields.

You will have to design one or more function(s) to process the comma-separated values of the file and the command line.

Also useful is a function that can trim leading and trailing spaces from a string and yet allow the string to be freed when no longer needed. Remember that the virtual address returned by `malloc` and other memory management functions must be the address passed to `free` when the time comes to free the memory.

The `trim` function provided to you handles the trimming of character strings.

## File Operations

Unless you prefer to read/write directly using system calls, you will need to use some of the following I/O library functions:

```
FILE * fopen( const char *restrict filename, const char *restrict mode);

int fclose( FILE *stream);

int scanf( const char * format, ...);

int fscanf( FILE * stream, const char * format, ...);

int sscanf( const char * str, const char * format, ...);

int fflush( FILE * stream);
```

Of course, you will also need `fprintf` to send error messages to `stderr`.

## Memory Management

You will need to use some of the following memory management functions:

```
void * calloc( size_t count, size_t size);

void free( void * ptr);

void * malloc( size_t size);
```

You must manage all dynamic memory allocations so that, upon termination of the program, all dynamic allocations have been properly freed, and there are no memory leaks as reported by **valgrind**.

In addition, there must be no other issues of memory reported by valgrind. That means there must not be memory reads and writes outside allocation boundaries, or conditional jumps based on possibly uninitialized variables.

---

# Grading

The grade will be based on the following:

- **80% Functionality**:
  - 10% Implementation that reads a file containing lines of format: `parent [, child [, child2...]]`
  - 70% Implementation of the following **COMMAND** examples, shown with required and [optional] inputs and description:
    - `add parent, child` to search for parent and add child.
    - `find [name]` to search from tree root; print information on name if found.
    - `print [name]` to print the breadth first traversal.
    - `size [name]` to count of all members in the [sub]tree.
    - `height [name]` to return the height of [sub]tree.
    - `init` to free all dynamic memory and start with an empty tree.
    - `help` to provide command help.
    - `quit` to free all dynamic memory and exit.
- **10% Design, Testing and Memory Management**:

  Module structure should reveal only those functions that are to be used by external code, and the rest should be encapsulated. This means proper use of `static` for making private things private.

  Code does not leak dynamic memory allocations. Severity of leaks will be measured by the number of leaks and the amounts detected by `valgrind`. Also means addressing other problems detected by `valgrind` such as memory overruns and uninitialized variables.
- **10% Style, Documentation and Version Control**:

  **Consistency of horizontal and vertical spacing is most important to aid the readability of the code.** The program code conforms to style and documentation guidelines for the course found in C-Style-Recommendations.pdf.

  Version messages are brief, yet complete and thoughtful; phrases or 'bullet points' are preferred. The `revisions.txt` file contains clear evidence of the diligent use of a version control system.

---

# Submission

When finished, submit your work using the ***try*** command below.

```
try grd-243 project2-1 offspring.c revisions.txt .gitignore [...]
```

The optional files must include any `.h` header files you defined for your design, and additional `.c` source files you defined.

The command `git log > revisions.txt` typically creates the `revisions.txt` file.

To check your submission, you can use the try query command like this:

```
try -q grd-243 project2-1
```

Please refer to try-summary.html for more information and guidance on using *try*.

---

Update History

- Version 1.2: updated text outputs to match try tests
  Mon Apr 3 08:33:46 EDT 2017 Ben K Steele, bks@cs.rit.edu
- Version 1.1: fixed typo in tree node structure example
  Sat Apr 1 09:28:59 EDT 2017 Ben K Steele, bks@cs.rit.edu
- Version 1.0: initial release
  Wed Mar 8 17:35:57 EST 2017 Ben K Steele, bks@cs.rit.edu

---