Background

In the APP C33-1 EXT assignment, you learned about and implemented a **linked list** data structure. A linked list is a special case of another data structure called a **binary tree** (which is a special case of the more general **tree** data structure, which is a special case of the even more general **graph** data structure). A binary tree is a nonlinear organization of data structure elements in which each data structure element node points to at most two (hence binary) other nodes.

Figure 1 below shows a general binary tree structure and labels various aspects. The binary tree starts with a **root** node which never changes even when new nodes are added. All operations start with the root node. Each node can have up to two **child** nodes: a **left child** and a **right child**. A node which has one or more child nodes is called a **parent** node. It is important to note that while parent nodes point to their child nodes, child nodes *do not* point back to their parent nodes. Nodes which have no child nodes are found at the bottom extremities of the tree and are called **leaf** nodes.

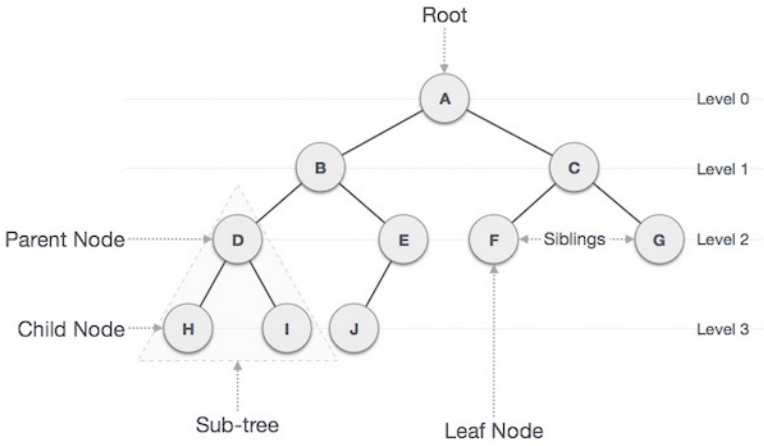

Figure 1: Binary tree structure from https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

A special and highly-used case of a binary tree is a **binary search tree**. In a binary search tree, each node has a specified **key** data member which is used to sort the nodes in the tree. A parent node's left child has a key value less than the parent node's key, and a parent node's right child has a key value greater than the parent node's key. In this implementation, each node has a unique key value, and there cannot be duplicates. Figure 2 below shows a simple example of a binary search tree where the key values are shown as integers. Note that the root node has an integer key value of 8, and that all nodes to the left of the root have a key value less than 8 while all nodes to the right of the root have a key value greater than 8.

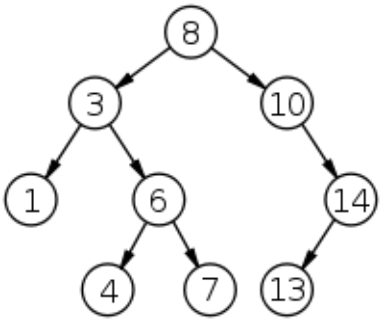

Figure 2: Binary search tree example from https://en.wikipedia.org/wiki/Binary_search_tree

Some advantages to using a binary search tree over a linked list are that the binary search tree organizes the elements in a sorted manner and that it is much faster to search for an element in a binary search tree. Some disadvantages are that a binary search tree does not retain the order of element insertion since the elements are organized by the key and that if the binary search tree is not well balanced, then insertion and searching can become very inefficient.

The process of moving through a binary search tree to visit all of its nodes (and possibly print out their data or find all nodes which meet a certain condition based on their data) is called **traversal**. The most intuitive type of traversal is called **in-order traversal**, in which the nodes are visited in order of their key values (either ascending or descending order). To display in ascending key value order, all left subtrees are visited in full before all corresponding right subtrees. Thus, for any given parent node (including the root node), all of its left child nodes are visited, then the parent is visited, and then all of its right child nodes are visited. If this parent also has a parent node (which the root node does not), then this second parent node is visited after the first parent's own right child nodes are visited. Figure 3 below shows the process of in-order ascending traversal.

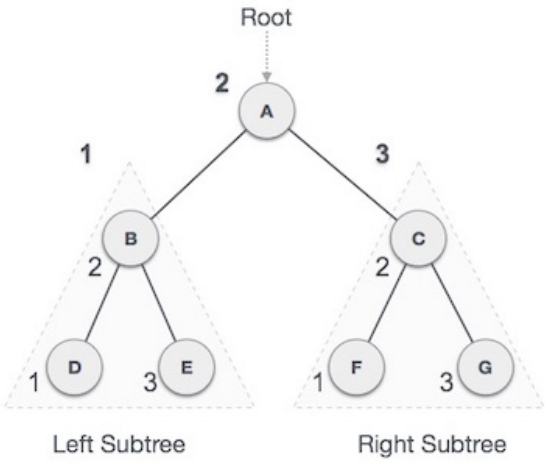

Figure 3: In-order ascending traversal in a binary search tree from https://www.tutorialspoint.com/data_structures_algorithms/tree_traversal.htm

Since this traversal process is somewhat iterative and the depth of a binary search tree may not be known, it is common to use a **stack** data structure to keep track of which levels and which parent nodes have been passed. Each time a parent node is visited which has left child nodes, the parent node address is pushed onto the stack. Then, when all of its left child nodes are visited, the parent node address is popped from the stack to be revisited. Then, the parent node's right child nodes are visited. This process works for multiple levels of parent nodes because the order of items pushed and popped with the stack remains the same. The reason this is needed is because child nodes do not point back to their parent nodes.

Here are two related fun facts:

1) The use of the stack in this way is essentially manually implementing the programming process of **recursion**. In fact, low-level assembly-level recursion for subroutines is done with the hardware stack which you have simulated.
2) **Morse code** is actually an implementation of a binary search tree where all left child nodes have a dot (".") and all right child nodes have a dash ("-"). You can see an image here: https://upload.wikimedia.org/wikipedia/commons/c/ca/Morse_code_tree3.png.

In this assignment, the scenario is that you are developing an online music streaming service which will allow users to add songs to their library and display them in different orders in different playlists. The service is still in early development, and right now, each song just has a title (character array) and an artist (character array). You are provided with an input file containing some test songs. The first line of the file has the number of songs present in the file. Each line after that has the title of a song, and then the artist for that song. The songs in the file are not already in alphabetical order, so, you will insert them into a binary search tree according to their titles.

You will use a binary search tree to store the data for the song structures by using the song titles as the keys. To compare key values, you will have to use the `strcmp()` function. Recall that letters earlier in the alphabet have a "lower" value than letters later in the alphabet. The code will be able to read in the provided songs and sort them appropriately, allow the user to enter in new song nodes into the binary search tree, and traverse the binary search tree to display the songs in both forward and backward alphabetical title order.

Instructions

- Complete the C34-1 assignment which gives more practice with structs in C.
- Copy `APP_C34_1_EXT_SKELETON.cpp` and `APP_C34_1_EXT_songs.txt` from:

  `/share/EED/class/engr1281/students/c/Class_34/Application`

- Begin by opening the skeleton code and adding your header information.
- Most of the code is provided for you, but you must fill in the provided blanks and write the code for the `display_backward()` function based on the code for the `display_forward()` function.
- Once your code is fully working, have fun adding new songs to the list and displaying them!
- **Note:** The exact implementation of the stack data structure here is slightly different than that in the APP C32-1 EXT assignment since the values to be stored on the stack are pointers. However, the general method is exactly the same.