ECM2433 The C Family

Continuous Assessment I

Report

1 Introduction

The following report will describe the design and structure of the menuTrees program. Additionally, it will outline any assumptions made and the construction process of the tree data structure. It is expected that the reader has familiarised themselves with the task for this continuous assessment before reading this report. There will be multiple sections of this report, each designed to thoroughly outline certain aspect of the exercise.

2 Compile & Link

This section will explain how to compile and link the different modules into the `menuTrees` executable. Please ensure that you have the following files inside your current directory:

- `menuTrees.c + menuTrees.h`
- `tree.c + tree.h`
- `fileIO.c + filIO.h`
- `sort.c + sort.h`
- `compileLinkBash`
- `compileLinkKsh`
- `testExample1.txt`
- `testExample2.txt`
- Any other files you would like to run against the program.

Assuming you do not have the current directory added to your classpath, enter the following command at the shell to compile and link the program.

Bash Shell
```
./compileLinkBash menuTrees
```

Korn Shell
```
./compileLinkKsh menuTrees
```

3 Execute

You should now have an executable in your current directory called `menuTrees`. To run the program with input data, type the following command at your shell.

```
./menuTrees yourFile.txt
```

4 High Level Structure

The following section will present the reader with a general structure of the program before diving into more low-level description of the data structures used to build the menus. The high-level description consists out of the following elements: Get user input, read the file supplied by the user, print the data read from the file, resolve nodes with duplicate parents, build the tree data structure, print the tree data structure, delete the tree and linked list containing the B nodes.

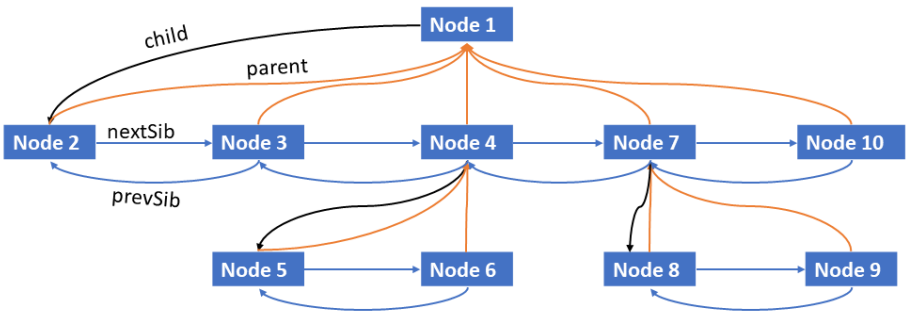
1. **Step – Get User Input (Functions: main)**
   This stage is relatively simple as the user supplies the file to be read as they start the program at the command line. The user *must* give a parameter after the name of the program, as specified in the Execute section of this report. If the user does not give this parameter, the application will print an error message and exit.
2. **Step – Read File (Functions: readFileStr, readFormatA, getFormatB, constructNode, insertB, insert)**
   After it has been verified that the user has given a single parameter to the program, it is time

to find the file of which the user has given the name and opening it. If the file does not exist or cannot be opened, the program will print an error message to the shell and exit. If the file is successfully opened, the first letter of each line is read to determine whether the following characters are interpreted in the 'A' or 'B' format. The remaining characters on that line are used to construct an 'A' or 'B' node, which is inserted into a linked list.

3. **Step – Print Data Read From File (Functions: printFormatA, printFormatB)**
   The program now contains two linked list, one with all of the 'A' nodes in ascending order according to their nodeIDs, and one with all of the 'B' nodes in ascending order according to the childIDs. Each respective list is then printed according to the format specified in the requirements. It would be possible to include this functionality while the data is actively being read from the file, however reading the data and printing it are two different functionalities. Keeping them separate is of interest because another person wanting to utilise the module might only want to read the file without printing its contents, thus making life easier for them.
4. **Step – Fix Multiple Parents (Functions: fixDuplicateParents, findNode, constructNode, mergeSort, sortedMerge, split)**
   The fact that a node which only occurs once in the 'A' lines in the file but may very well occur more than once in the 'B' lines is an issue to be dealt with. For example, Gus Grissom appears once in A (A0038) but twice in B (B0028B0027, B00280038). Because each node only has one pointer to a parent, the same node cannot point to two parents at the same time. More on this in section *5.1 Multiple Parents*.
5. **Step – Build The Tree (Functions: buildTree, findNode)**
   This step will be described in detail in the low-level section as otherwise repetitive comments would be made.
6. **Step – Print The Tree (Functions: printTree)**
   Once the tree data structure has successfully been constructed, the root node can be passed into a function utilising a depth first search algorithm which prints out the nodes of the tree. This is done recursively.
7. **Step – Delete Lists (Functions: deleteTree, deleteBNodes)**
   The last and final step of the program is to free any allocated memory by walking through both the 'A' and 'B' nodes and using the free function to deallocate memory of each node.

5 Low Level Structure

The next segment will provide an in-depth view of how the tree data structure was conceived and is implemented using elements of the C programming language. Here is an overview of the tree using the `testExample1.txt` file.



- `nodeID`: The ID of the node.
- `label`: The label of the node.
- `parent`: A pointer to the parent of the node.
- `child`: A pointer to the first child of the node.
- `next`: A pointer to the node with the immediately larger nodeID.
- `nextSib`: A pointer to the next sibling.
- `prevSib`: A pointer to the previous sibling.

The tree is a doubly linked list, with each node having multiple links to other nodes. Depending on which way the list is to be traversed, different links can be taken. This particular instance of a tree is an n-ary tree as the only limit to the number of children a node can have is the memory available to the program in the computer. Enabling multiple children per node is made possible by placing all children of a particular node in a linked list and having the parent of all of those children point to the first sibling. All children are directly linked to their parent, but the parent is directly linked to only the first child. All siblings are doubly linked as well to facilitate the creation of the menu number for each node.

5.1 Multiple Parents

Before creating the tree, it is necessary to check whether there are any nodes which have more than one parent associated with them. Neglecting this probing may result in an incomplete tree structure as the ways parents, children, and siblings are linked do not naturally allow for a node to have multiple parents. Circumventing the issue is done in multiple steps. First, a duplicate childID is found in the 'B' nodes. Second, the 'A' node which has that childID is copied, given a new nodeID and placed at the end of the 'A' linked list. Third, the duplicate childID in 'B' nodes will be changed to the new nodeID of the copied node. Fourth and last, mergesort the 'B' list to make sure it is in ascending order.

5.2 Creating The Tree

Once the file has been successfully read, the 'A' and 'B' linked lists have been constructed, and the issue of multiple parents per node has been solved, it is time to construct the tree. This occurs in two steps:

1. Link Parents to Children

Link parents to children and children to parents by traversing the 'B' nodes linked list, getting the child and parent ID, finding the respective nodes in the 'A' nodes linked list, and setting the `child` and `parent` variables of those nodes.

2. Link Siblings

Link siblings to each other by traversing the 'B' nodes linked list and for each node find the last sibling (the last node in the `nextSib` traversal) and linking it to the next sibling to be found in the 'B' nodes linked list.

6 Assumptions

There are a number of assumptions involved in the development process of this application:

- The text file supplied by the user has the A section before the B section.
- The B section will start with a top-level node, e.g. B00010000.
- The label for each node does not exceed 99 characters.
- The length of each menu number (e.g. 1.3.6) will not exceed 255 characters. This means that numbers can only be displayed accurately until a certain depth of the tree is reached.