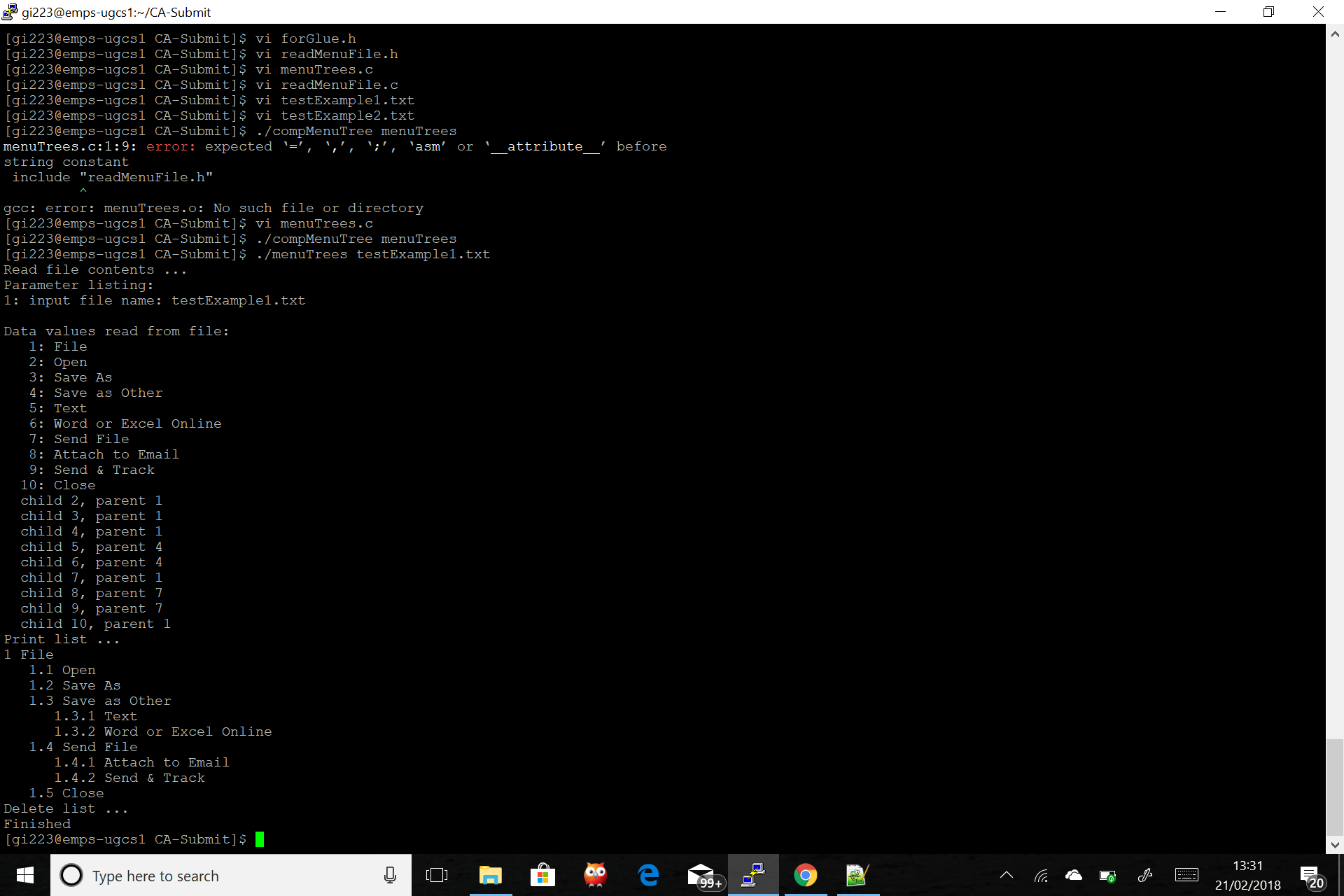
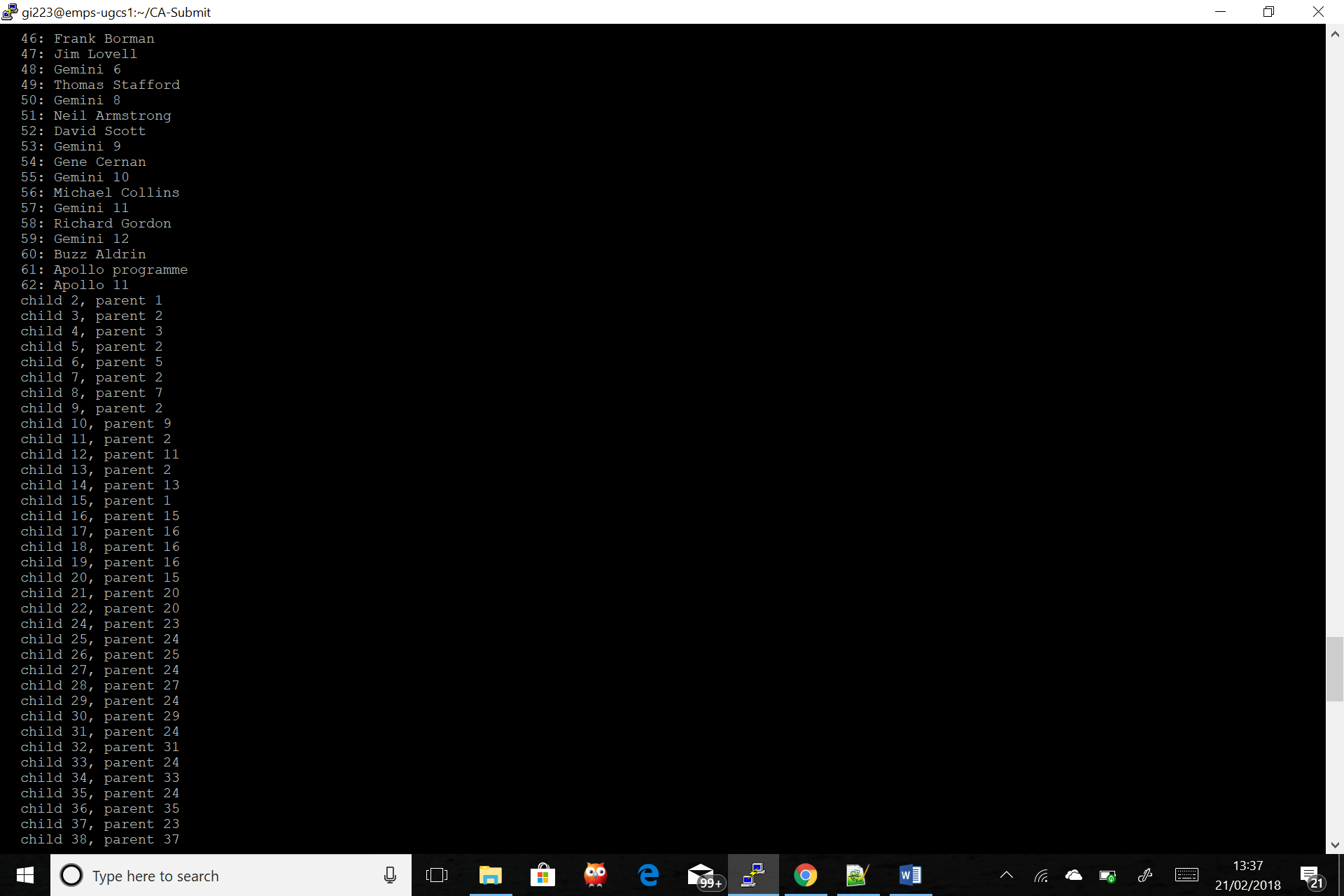
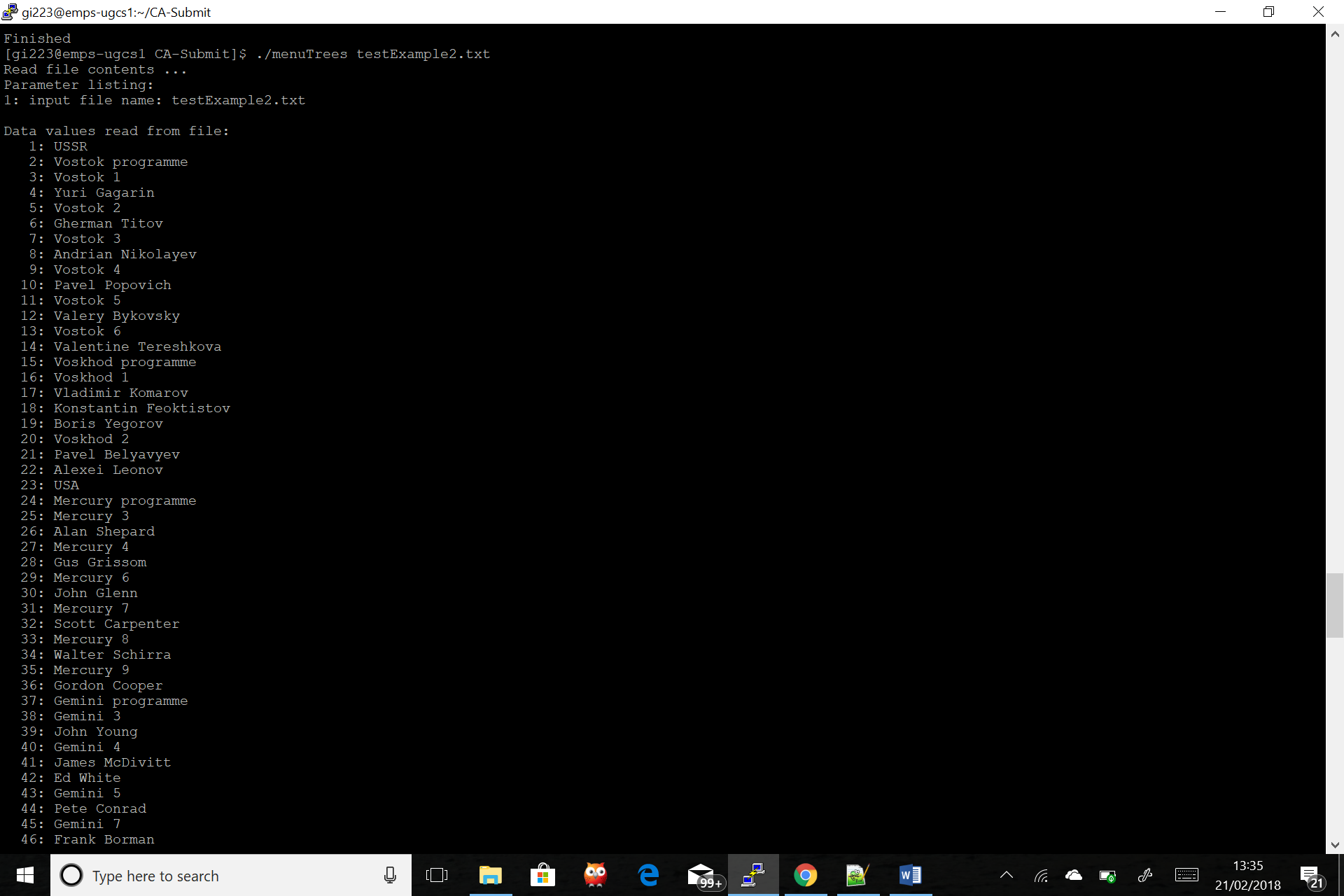
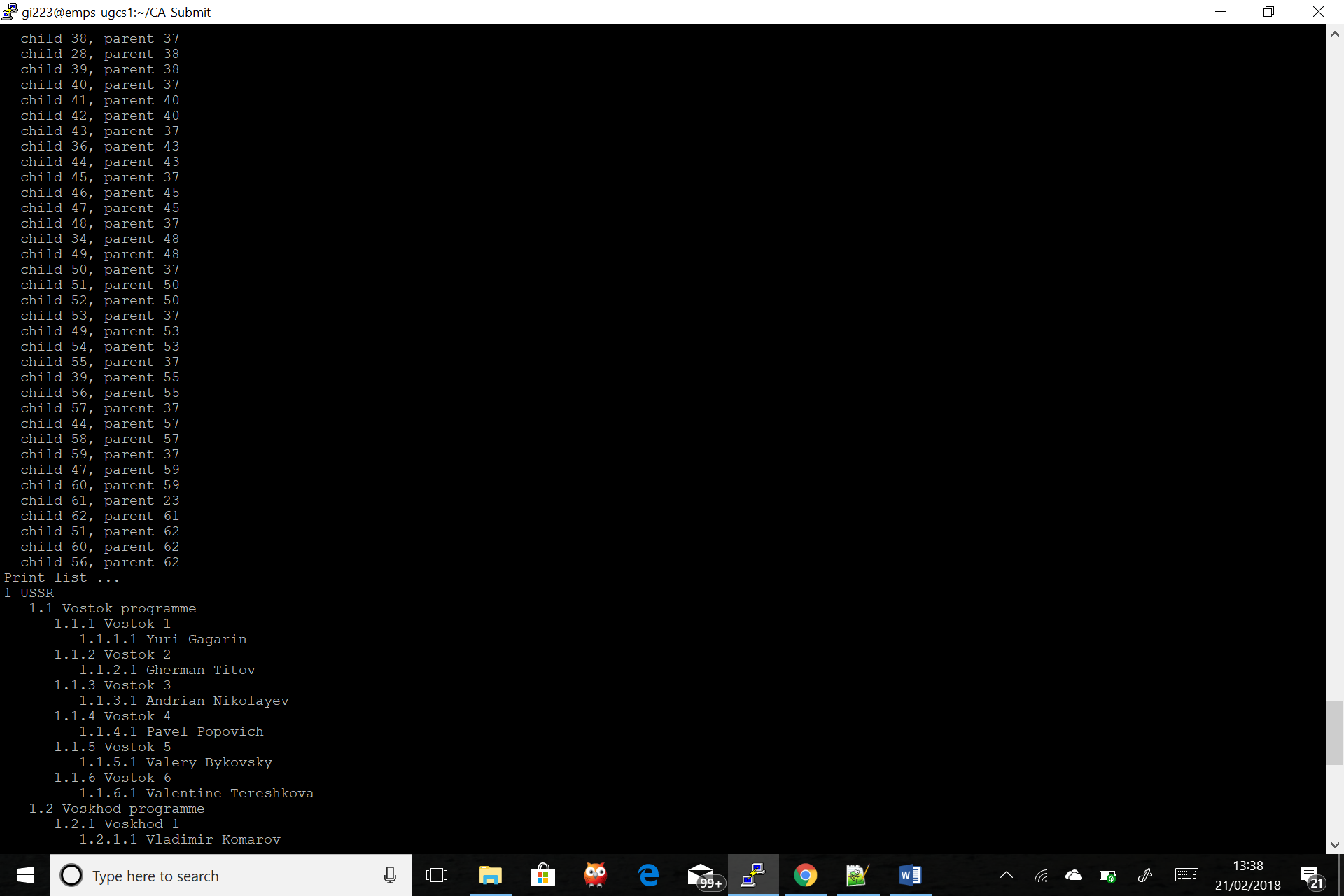
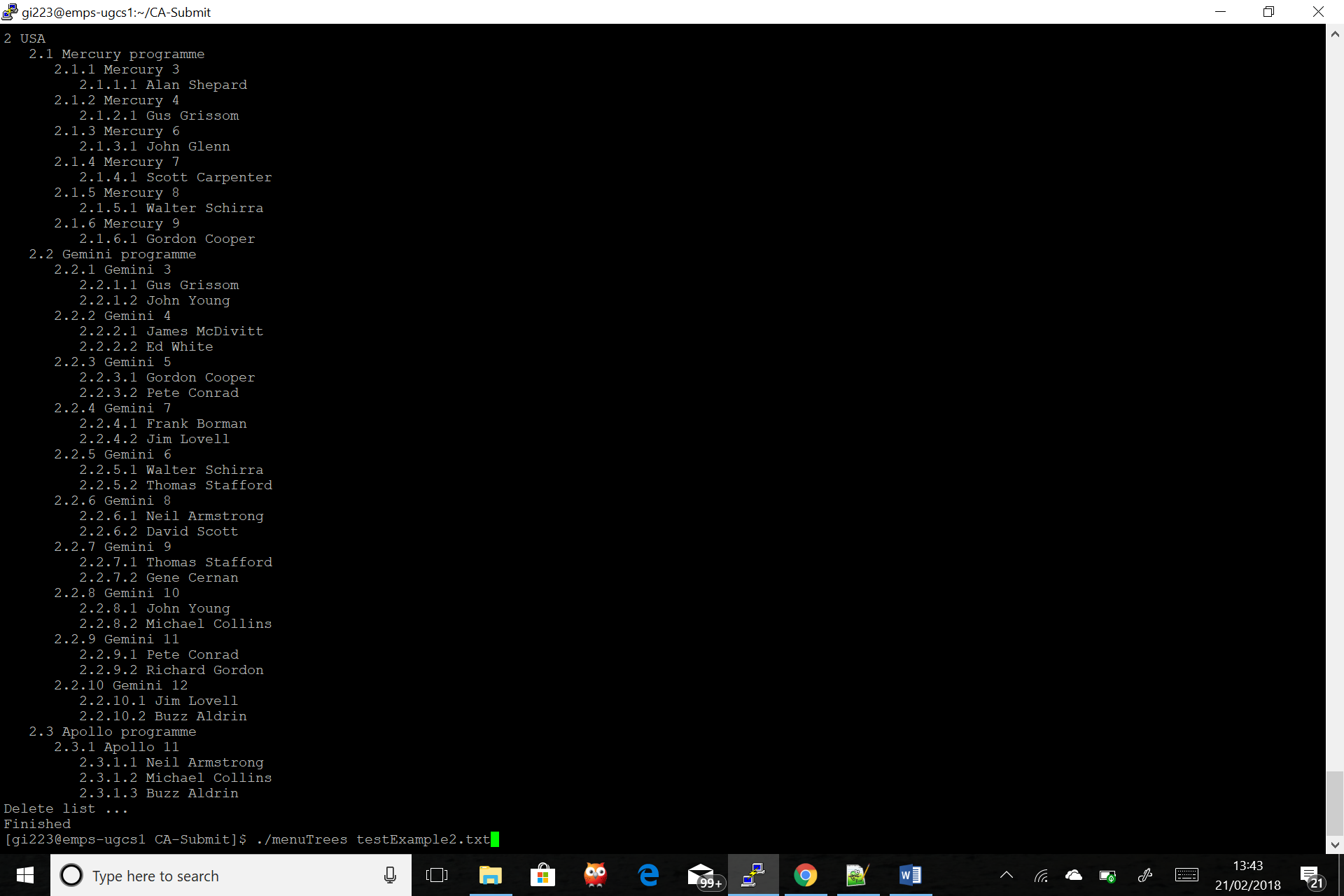
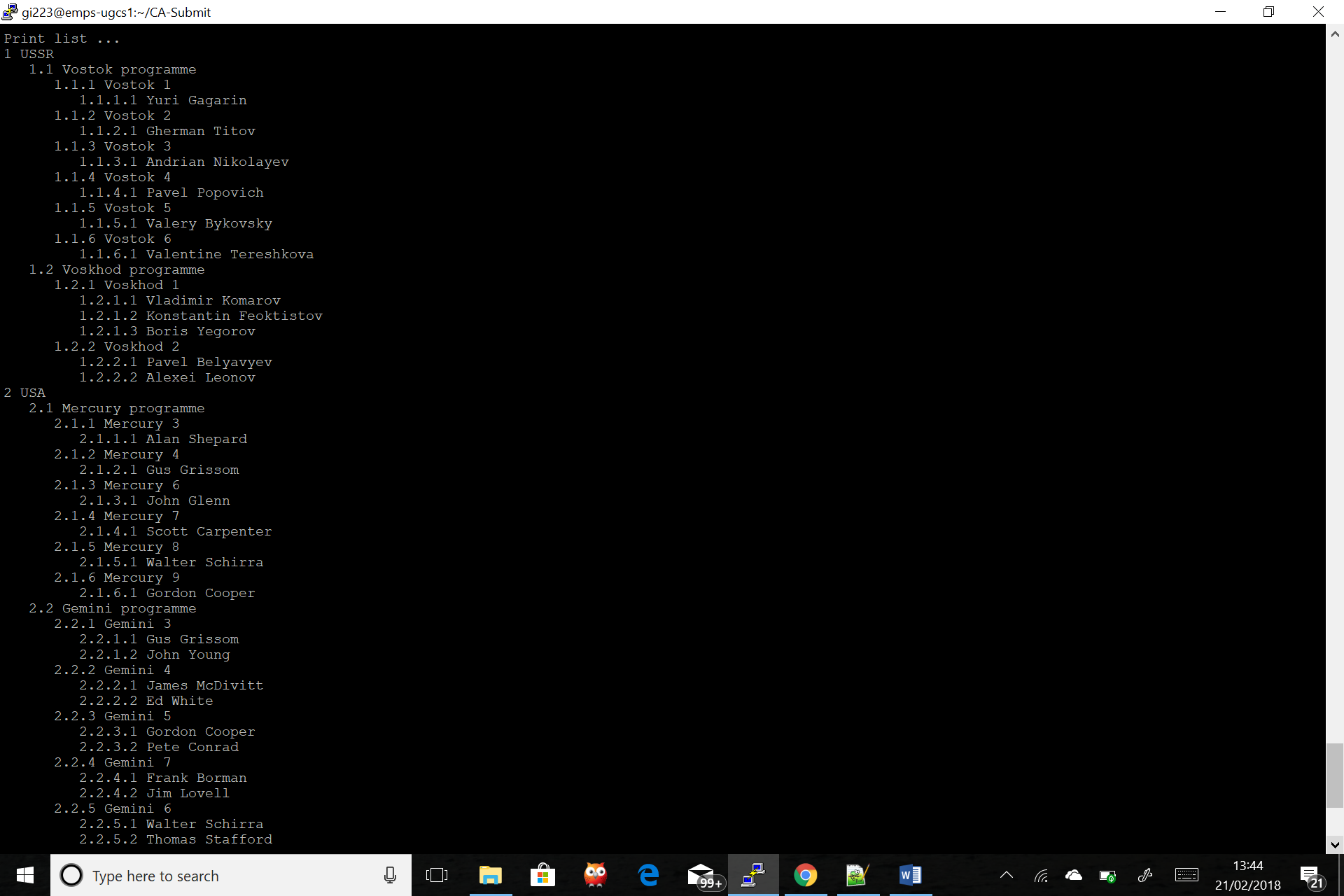
Output from C program:







ECM2433 CA1 menuTrees – report:

To accompany the source code, this report is provided to explain my choice in Structs used and how are those NODEs structured to hold data, my choice of encapsulating code into many functions and several modules and a summary of errors trapped.

I designed my program on the principle that I would read the file line by line and when the character ‘A’ is read, then a node would be created with the corresponding 4 digit ID number and string text label given in that line as specified in the CA. I then allocate memory to create the NODE and trap the possible error of the computer running out of memory; the file would then be closed and the program would exit with a failure.

The struct created for the NODEs will have the links defined as: next, prev, nextSibling, prevSibling, parent and children. Thus, the NODEs created from reading the ‘A’ characters will be linked through the ‘next’ and ‘prev’ pointers of the struct and will be sorted according to their ID number in ascending order. A doubly linked list is not needed in the specification of the CA but it allows the code to be more adaptable. For example, if I needed to move back between the NODEs, due to an additional specification, then my program can be easily modified to do so.

An assumption made is that the file will contain all the lines that contain the character ‘A’ at the start and then the following lines will begin with the character ‘B’. I also assume that only one text file will be read at a time and that the lines will follow the specification as set out in the CA. If the file does not abide by these rules, my program will simply skip over it and not read it.

When I begin reading the lines that begin with the character ‘B’, I look for the child ID first. If the ID is not found, then I return NULL and that error is trapped and printed to the stderr stream. The same thing happens when I search for the parent ID. However, I decided to keep the program running even if these errors are found because the user will be able to see the behaviour of the program with the lines in the file that are in the correct format. Hence, making it easier to identify where the possible errors are in the file.

If the parent ID and the child ID can be found, I link the child NODE to the parent NODE by using the ‘parent’ pointer of the child NODE. I do this to be able to deal with the case if a child NODE has more than one parent NODE since I can check if child -> parent != NULL and then deal with it. I handle this situation by cloning the child NODE and add it to the end of the doubly linked list, which was created through reading the ‘A’ chars, through the ‘next’ pointers, so that I can free its memory at the end of my program. This clone NODE will be connected to the parent NODE as normal.

Since the order of the file names are not guaranteed in the file being read, I created the pointers nextSibling and prevSibling to sort them at the sub menu levels. This meant that when I call the function to print text labels, all I must do is traverse the tree in a depth search manner.

I created a module “forGlue.h” that contains functions needed only for the “glue” function in the “readMainMenu.c” file for organisation purposes. That way, I was able to test whether the functions worked on sample data easier than if I had all of the functions in one file. The “glue” function solely exists to aid with the recursion of the printList function to be able to traverse up the parents after the maximum depth of the tree has been traversed.

I created many sub functions to assist with the main functions seen in the menuTrees.c file to be able to debug my program quicker and for readability for the user. During the process of creating this program, debugging was simpler as I could locate and isolate the bug faster. Also, I included comments to better explain the functionality, the arguments taken and the return value. This was useful as it meant I was able to reuse functions instead of writing repeated code.

For printing the label of the NODEs to the terminal, I passed the previous label of the NODE visited in the depth search manner and manipulated it to get the new label for the next NODE. This meant that I had to use malloc to allocate memory for the strings I was storing so that they can live outside of the function’s lifespan. Here, like in the creating new NODEs part, the possibility of ‘running out of memory’ error is trapped. If this error occurs, the file is closed and the program exits with a failure.

I used the free( ) method on the strings that I did not need while recursively going through the tree to make sure all heap blocks were freed dynamically and allow more space for the other strings to be created as the program keeps running. I use more memory than I would have liked to but with the mallocs used, I ensure that if the memory was not able to be allocated, then I would safely exit the program and close the file.

List of errors trapped:

Since I covered them above, I will just write what type of errors are trapped.

* If the file cannot be accessed, an incorrect file name was given or if there is no file given. This is found at the start of the processFile function where if the previously mentioned errors are found, the file is closed and the relevant error messages are printed to the stderr stream.
* If there is not enough memory for malloc for creating the NODE structs and the char arrays for the number label part of the text labels.
* If the queried ID from the lines starting with ‘B’ cannot be found.

Picture of the tree structure:

* The parent NODE’s children pointer always points to the child NODE that has the smallest ID relative to its siblings.
* The sibling NODEs are always sorted from smallest ID to biggest
* I did not want to create two types of structs for the NODEs, so each NODE is linked to each other through the pointers ‘next’ and ‘prev’. Thus it is a linked list, but also a tree structure as shown below.

NODE 5

NODE 2

NODE 1

NODE 4

NODE 3

Blue – Sibling links

Orange – Children link

Yellow – Parent link

**Instructions on how to run my program:**

**Once in the directory of the folder, type at the command line:**

chmod u+x compMenuTree

./compMenuTree menuTrees

**To run the program, specify a text file, for example:**

./menuTrees testExample1.txt

./menuTrees testExample2.txt