Assignment 01

Due: **11:59pm, January 26th**

Dictionary Tree Specifications

The dictionary tree is a tree storing a set of words (or strings), with each word stored along a tree path. Starting from the child nodes of the root node, each tree node represents a single character or letter of the word. Each node contains a collection of child nodes representing the next set of possible characters to continue the word from the current node. A dictionary tree constructed this way supports very efficient search and insert operations, in O(K) time with K being the length of the word searched or inserted.

For this implementation, **valid** word characters are:

- alphabet characters from a to z
- the apostrophe ' character.

Below is a typical representation of a dictionary tree node in a C struct type (similar data contents can be defined as member variables in a class for C++ implementation):

```
#define NUMOfCHARS 27 /* a-z plus ' */
typedef struct dictNode
{
    // isWord is true if the node represents the
    // last character of a word
    bool isWord;

    // Collection of nodes that represent subsequent characters in
    // words. Each node in the next dictNode*
    // array corresponds to a particular character that continues
    // the word if the node at the array index is NOT NULL:
    // e.g., if next[0] is NOT NULL, it means 'a' would be
    // one of the next characters that continues the current word;
    // while next[0] being NULL means there are no further words
    // with the next letter as 'a' along this path.
    struct dictNode *next[NUMOfCHARS];
}
```

Note: Both the big case and small case of a letter should correspond to the same index in the child node array. For example, both letters 'A' and 'a' should correspond to index 0**.**

Each node entry also has an **isWord** flag. If the flag is true, the node signifies the end of a word entry in the dictionary.

Figure 1 below shows a dictionary tree example. Non null pointers in the **next** array are shown as arcs to child nodes, and the end of word indicator is shown for each node. A label that is not part of the data structure is shown at the top of the node to illustrate how one reaches that node.

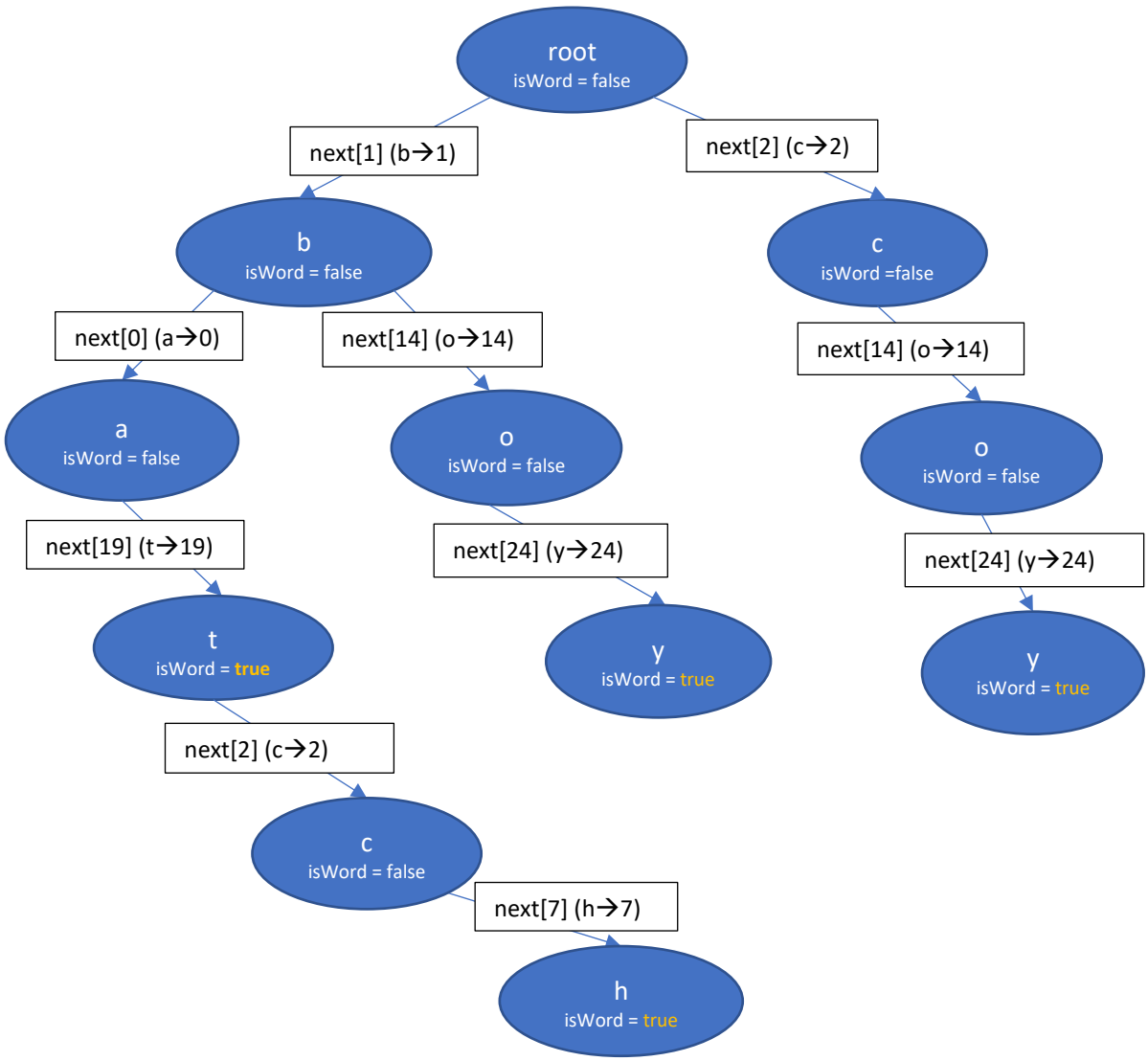

Figure 1 - Dictionary tree containing a number of words

Proposed implementation of Tree operations

You would not lose points if you do not follow the proposed implementation below; however, again, you **MUST implement the pointer tree data structure**.

With the representation of the dictionary tree structure, you need to implement the insert operation for inserting a word to the dictionary tree and the search operation for search the tree to count the words that start with a specific string.

Below is a proposed code design for implementing the dictionary tree data structure and its operations:

- dicttree.h (in C++ or C)
    a. Defines the data structure of the dictionary tree node described above, and the signatures of its operations for inserting a word to the dictionary, and for searching and counting words starting with a string.
    b. Adds other supporting helper methods if needed.
    c. Add node method signature:
        - In C++, `bool add(const char *remainingChars, const char * wordBeingInserted = nullptr); // wordBeingInserted is for debugging purpose`
        - In C, `bool add(struct dictNode *dictnode, const char * remainingChars, const char * wordBeingInserted); // wordBeingInserted is for debugging purpose`
        - Return value (boolean) would indicate whether the word is added / inserted to the tree successfully
        - Implementation tips:
            1. This method is initially called from the root node to insert the new word.
            2. Use recursive calls (or iterative loop) to traverse the tree structure to insert the word, one character at a time starting from the first character (you need to convert each character from the word to an index of a child node).
            3. The "`remainingChars`" argument is for passing the remaining portion of the word being inserted in a recursive context.
            4. The "`wordBeingInserted`" argument is for remembering what the word in the insert is, this is for debugging purpose in a recursive call context.
    d. Search node operation signature for finding the node (in the tree) that ends a string, i.e., the node that contains the last character of the string being searched:
        - In C++, `dictNode* findEndingNodeOfAStr(const char *remainingStr, const char *strBeingSearched);`
        - In C, `dictNode* findEndingNodeOfAStr(struct dictNode *dictnode, const char *remainingStr, const char * strBeingSearched);`
        - Implementation tips:
            1. This method is initially called on the root node to start the search.
            2. Use recursive calls (or iterative loop) to traverse the tree structure to find the string, one character at a time starting from the first character (you need to convert each character from the word to an index of a child node). **Returns the node pointer that ends the string if found; otherwise, return NULL pointer.**
            3. The "`remainingStr`" argument is for passing the remaining portion of the string being searched in a recursive context.
            4. The "`strBeingSearched`" argument is an argument that is passed without modification in the call. Its main purpose is to help you understand the context in recursive debugging.

e. Count word operation signature for counting the number of words (in the tree) that start from a node:
    - In C++, `void countWordsStartingFromANode(int &count);`
    - In C, `void countWordsStartingFromANode(struct dictNode *dictnode, int &count);`
    - Implementation tips:
        1. This method is initially called on a starting node to start the counting.
        2. Starting from the dictnode, use recursive calls (or iterative loop) to traverse the whole sub-tree from the node (including the node) to count all words that start from the node. Use the count argument passed by reference to count**.**

f. For searching the tree to count the words that start with a specific string:
    - First call `findEndingNodeOfAStr` to find node that ends the string,
    - Then call `countWordsStartingFromANode` to count the words starting from the node returned from `findEndingNodeOfAStr`

- dictionary.cpp (for C++) or dictionary.c (for C) for implementing dictionary.h.

Main program

With the above dictionary tree implementation, you would then need to write the main program to:

a. Read words from a dictionary text file and insert them to a dictionary tree.
b. Read words from test text file, and **for each word read in**: search, count, and print the number of words in the dictionary tree that start with it.

Suppose you put the main program code in a file countwords.cpp (C++) or countwords.c (C). It should contain the **main(int argc, char **argv)** function. Implementation tips are below.

Note that for any of the library functions we suggest, you can read the manual page on Edoras by typing "man fn_name" at the shell prompt, e.g. man strtok; or refer to https://www.kernel.org/doc/man-pages/.

Your main(**int argc, char **argv**) function should expect two arguments. The first argument is the file path to the dictionary source text file, the second is for the file path to the test text file. Minimal error checking is required, fail gracefully when there are the wrong number of arguments, or a file does not exist.

For reading a text file line by line

- In C++, use std::ifstream and std::getline.
    - std::ifstream dictstream(filepath); // open file for parsing
    - std::string line;
    - // iterate over dictionary file line by line
    - while (std::getline(dictstream, line))
- In C, use FILE *fp = fopen("filename", "r"), then use fgets(line, sizeof(line), fp) to read each line to a char buffer.

To extract all words from each line read in, you can use the strtok() function from <string.h> to parse each line buffer read from the file. The strtok function iterates across a buffer, pulling out groups of characters separated by a list of characters called delimiters (see snippet below for an example).

You MUST use the following delimiter string to separate words:

```
const char *delimiters = "\n\r !\"#$%&()*+,-./0123456789:;<=>?@[\\]^_`{|}~";
```

Example:

```
char *word = strtok(line_c, delimiters);
  while (word != nullptr)
  {
   // call add method to insert word to build the dictionary tree
   …….
   // handle error from insertion result
   …..

   // read next word
   word = strtok(NULL, delimiters);
  }
```