Assignment 1

Most modern operating systems (UNIX™ based OS, Windows, etc.) were written in C/C++ and Assembly languages. For the purpose of learning in this class, you are required to do all programming assignments in C or C++ language. This first assignment is designed for you to demonstrate necessary knowledge and proficiency in C or C++ programming as part of the prerequisites to enroll in this class.

Important Notes:

1) **Due: January 30th, at the BEGINNING of the class**. As this assignment is used for determining your enrollment eligibility, and for the fairness and uniformity of grading, **NO LATE submission** will be accepted for this assignment. All students (currently enrolled or waitlisted) must submit this assignment by due time if you intend to remain in the class or enroll.
2) You must do this assignment on your own. No teamwork is allowed! An automated program structure comparison algorithm will be used to detect software plagiarism. An affidavit of academic honesty stating the program is written on your own will need to be submitted along with your submission of the code, see below.
3) Your submitted program must run in your account on edoras.sdsu.edu. If you are enrolled in the class, you will have already been e-mailed an edoras account. If not, please let me know as soon as possible.
4) You will be given the CORRECT output of the program invocation with specified arguments, and your output must match the correct output perfectly. Failure to match the output in this assignment by the due date posted above will result in you being removed from the class for lack of prerequisites.
5) Correct output from executing your program will clear the programming prerequisite to enroll in the class, but it may NOT give you a perfect score on the assignment. The structure, coding style, and commenting will also be part of the considerations for the final grade (see Syllabus and course FAQ).
    - Be sure to comment your code appropriately. Code with no or minimal comments are
      automatically lowered one grade category.
6) **What to turn in**:
    - A final version of your code must be put into the folder ~/cs570/a01/ on your edoras account (e.g., ~bshen/cs570/a01/) by due time, the grading on this assignment will be based on the test and inspection against this submitted version.
    - Hardcopy of your code.
    - Affidavit of academic honesty. See the single-programmer affidavit in the n the course frequently asked questions (FAQ).

Tree, a hierarchical data structure, is often used in representing OS entities (such as the organization of a file system, Unix-style process trees, etc.), and in implementing various OS algorithms (such as multi-level paging table for memory management, multi-level indexed files, etc.).
In this assignment, you will use C or C++ to:

1) First implement a dictionary tree that provides the abilities for a caller to insert words to a tree to build a dictionary and search the tree to find if a given word matches (case-insensitive search) any stored word in the tree.
2) Then write code to test your dictionary tree implementation by building a dictionary tree from a source dictionary text file and then use the dictionary tree to perform spell checks on all words read from another text file and print out all misspelled words there.

We start with a review of the dictionary tree data structure. The dictionary tree structure begins from a root node. Each node entry contains a collection of children representing a next set of possible characters to continue the word building from this current node. Valid characters for each word in this implementation are alphabet characters from a to z, plus the single quote ' character. Each node entry also has an EndOfWord flag indicating whether it is the end of a word entry in the dictionary. 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 illustration purpose, here 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 Nchars 27 /* a-z + ' */
typedef struct dictentry
{
    // isEndOfWord is true if the node represents end of a word
    bool isEndOfWord;
    // Subsequent characters in words.  Null pointers if there
    // are no further words with the next letter.
    struct dictentry *next[Nchars];

}
```

Figure 1 shows a small dictionary tree structure containing the words: bat, batch, boy, and coy. 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.

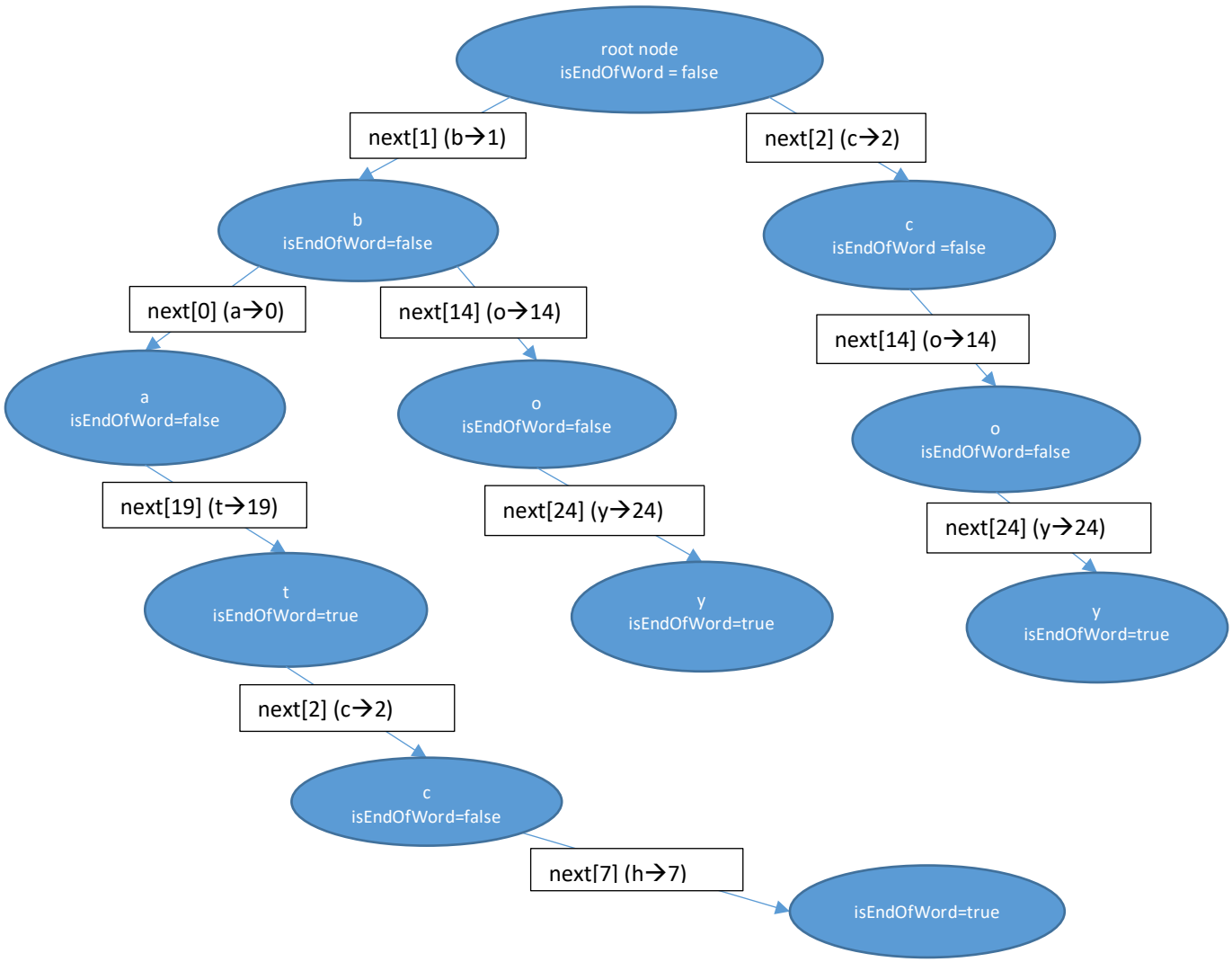

Figure 1 - Tree dictionary representation of the words: bat, batch, boy, and coy.

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 match a given word.

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

- dictionary.h (in C or C++)
    a. Defines the data structure of the dictionary tree entry described above, and the signatures of its operations for searching a word in the dictionary and for inserting a word to the dictionary. We recommend calling the structure dictentry. If coding in C++, make dictentry a class.
    b. Search operation signature:

- In C, `ResultType find(struct dictentry *dictnode, const char *word, const char *targetword);`
- In C++, `dictentry::ResultType find(const char *word, const char *targetword=nullptr);`
- ResultType here could be an ENUM to indicate the result from the find call, suggested values are:

```
typedef enum {
  CORRECT,   // word has correct spelling
  MISSPELLED,
  BADCHARS,  // word has characters outside [A-Z,a-z,']
  ADDED,
  EXISTS,
} ResultType;
```

- Implementation tips:
  1. This method is initially called on the root node to start the search.
  2. Use recursive calls to traverse the tree structure to find 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 "word" argument is for passing the remaining portion of the word being searched in a recursive context.
  4. The "targetword" 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.

c. Insert operation signature:

- In C, `ResultType insert(struct dictentry *dictnode, const char *characters, const char *targetword);`
- In C++, `ResultType insert(const char *characters, const char * targetword =nullptr);`
- See above for ResultType definition
- Implementation tips:
  1. This method is initially called from the root node to insert the new word
  2. Use recursive calls 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 "characters" argument is for passing the remaining portion of the word being inserted in a recursive context.
  4. The "targetword" argument is for remembering what is the target word in the insert, this is for debugging purpose in a recursive call context.

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

With the dictionary tree implementation, you need to write code to test the dictionary tree and generate output to be compared with the CORRECT output provided to you.

Your testing code should be in file spellcheck.c (C) or spellcheck.C (C++). It should contain the **main(int argc, char **argv)** function for testing your dictionary tree implementation. It should first build a dictionary tree from reading and parsing a dictionary text file, then use the dictionary tree to perform spell checks on each word read from another text file, and prints out all the words from the second text file (in the order of from beginning of the file to the end of the file) that are NOT present in the dictionary, **one MISSPELLED word on one line, to the standard output**. It is VERY IMPORTANT that you print out the MISSPELLED words in this format, as it is the format of CORRECT output file that is given to you to verify if your program generates the correct output.

- 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 std::getline.
    a. The execution of your main(**int argc, char **argv**) function should specify two arguments. The first argument is the file path to the text file to be checked for spelling. The second is for the file path to the dictionary source text file. Minimal error checking is required, fail gracefully when there are the wrong number of arguments or a file does not exist.
    b. For reading a text file line by line
        - In C, use FILE *fp = fopen("filename", "r"), then use fgets(line, sizeof(line), fp) to read each line to a char buffer.
        - In C++, use std::ifstream and std::getline.
    c. 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. Example:

```
char *token = (char*)strtok(line_c, separators);
  while (token != NULL)
  {
    //do something with the extracted token

    // get the next token from the line_c buffer
    token = strtok(NULL, separators);
}
```

    You can use the following delimiter string to separate words:

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

Follows are a number of files given to you for compiling and testing:

- **Please copy these files from ~bshen/cs570/a01/ on Edoras**
- Makefile

a. This is a sample Makefile for compiling and linking C++ code to generate an executable file. Suppose you have dictionary.h and dictionary.C for dictionary tree implementation, and spellcheck.C having the main function that drives the dictionary population and spell checking. If you are writing in C, add the line, and two lines:
    - CC=gcc
      Instructs the make program to use the GNU C Compiler
    - CCFLAGS=-std=c11 -g
      Flags passed to the compiler specified by CC. Instructs the compiler will use the ISO 9899 standard C implementation published in 2011 (commonly called C11) and adds debugging information to the executables. Debugging information lets use the GNU symbolic debugger (gdb) to debug your programs. There are a number of graphical user interface front ends for this, ranging from simple interfaces in emacs and vim to eclipse and Microsoft's cross-platform VSCode. For more on this, please refer to the course FAQ.

   If you are using C++, you will see that there are similar lines already in the Makefile, CXX is the variable used for the C++ compiler. C11 and C++11 have functionality that is desirable (e.g. the bool type in C and the type safe nullptr in C++).

b. To compile your code, simply type **make** at the prompt. For the C++ version, it will execute the following if you do not have any errors:

```
g++ -std=c++11 -g   -c -o dictionary.o dictionary.C
g++ -std=c++11 -g   -c -o spellcheck.o spellcheck.C
g++ -std=c++11 -g -o dictionary dictionary.o spellcheck.o
```

   With either C or C++, -o specifies the output file. The -c flag implies that we are compiling part of a program to an object file (machine code, but not a stand-alone program). Makefiles usually do this as it permits us to not recompile the whole program when only one module has changed. The last line links the object files together and adds some glue to create an executable program.

- Austen-Pride-and-Prejudice.txt
  A text file that provides words to be spell checked, see above.
- Poe_V2.txt
  Another text file to test your spell checker against.
- aspell_list_en.txt
  A list of English words that can be used to build your dictionary.
- a01_CorrectOutput.txt
    a. The CORRECT output file for you to use to verify the outputs from your program testing. This CORRECT output file is from checking the spelling of the given Austen-Pride-and-Prejudice.txt

You will also need to do the following that are **critical to the grading algorithm** being able to grade your program:

- Use an editor to a .plan file in your home account. This is a file that is used with the finger command, which provides information about users, e.g. finger jKikumba. It should contain the following, updated with your information:

  Student: Kikumba, Jules
  Nickname: Papa Wemba (you may omit this line)
  RedID: 123 45 6789
- Your program must have a Makefile. For this assignment, a Makefile is provided to you. Makefiles are program compilation specifications. The make tool examines to see if source files have changed and will compile only when sources are newer than executables. See the class FAQ for details on how to create a makefile. You can test if your makefile works by typing make. The makefile should produce a program called dictionary.

Start early on this assignment, your enrollment in this class depends on it.