Project P0

Due Friday, Feb. 7th

Objective: C/C++ programming refreshment, practice with standards and header files, practice with trees, traversals, string manipulations, file IO, and use of hellbender server (hellbender-login.rnet.missouri.edu). Program must compile and run on hellbender-login.rnet.missouri.edu.

Summary: Program will accept input from different sources, create a binary search tree based on the number of characters in the string, traverse the tree using level-order, pre-order, and post-order traversals, and print out the results for all three traversals with specific indentations.

Invocation requirements:

`P0` [*filename*]

where *filename* is an optional argument. If a file name is given, the program will read from the file. Otherwise it will read from the keyboard until simulated keyboard EOF (Ctrl + D).

- `P0` // read from the keyboard until simulated keyboard EOF
- `P0` < *filename* // same as above except redirecting from *filename* instead of keyboard (***no*** extra code needed)
- `P0` *filename* // reads from *filename*

If none of the three options are provided by the user an appropriate error message is given and usage is specified. Note that user input of three or more strings on the command line should also generate this error message. For example:

```
P0 file1 file2                    // command line arguments provided by user

Fatal: Improper usage             // example error message with usage specified
Usage: P0 [filename]
```

Input data:

Assume you do not know the size of the input file. Check that input data are all character strings with letters, numbers, and/or any of the following special characters:

! " # $ % & ' ( ) * + // these characters correspond to ascii numbers 33 to 43

Assume the strings are separated with any number of standard white-space separators (space, tab, newline).

If an unallowed string is encountered, program needs to abort with an appropriate error message.

Program function:

- Program will read in strings and put them into a binary search tree (BST) with respect to the number of characters in the string.
- Each node in the tree will contain:
  - The number of characters in the string
  - Two child nodes (left and right)
  - A list of the strings already seen that have the same number of characters
- Tree is never balanced nor restructured (other than growing new nodes)

Program output requirements:

- Program will output three files named *filename.levelorder*, *filename.preorder,* and *filename.postorder*

    - The first half of the filename must match the input filename or 'out' for keyboard entry (for `P0` usage or `P0 < filename` usage)
- Output files will have one node per line, in the order of the traversal
    - Each line will be indented by 4 x the depth of the node (in number of blank spaces)
        - For example, the root node will have no indentation, a node in the first level will have 4 blank spaces before the node information is given, a node in the second level will have 8 blank spaces before the node information, etc.
    - The node information will start with level number, followed by number of characters in the string, followed by a list of the strings seen with the given number of characters, in the order in which they appeared (each of the printed strings need to be separated by a single space)

***Architectural requirements*:**

- Have the following functions minimum in addition to the main function (the types and arguments are just suggested, the **names are required**)
  ```
  node_t* buildTree(FILE *); // function that builds the tree from file
  void traverseLevelOrder(node_t*, const char[]); // parameters: tree root, and output basefilename
  void traversePreOrder(node_t*, const char[]);
  void traversePostOrder(node_t*, const char[]);
  ```

- Put the above four functions into two files with proper headers (`buildTree.h` and `traversals.h`)
- Define the node type in `node.h`
- ***Be sure to use this functional architecture***

Suggestions:

Our illustrations and discussions will use pseudocode or C/C++. We will approach the project from functional modularization point of view, so preferable design is functional.

Traversals can be reviewed in Cmp Sci 3130 textbook. Briefly:

- Level-order processes each node from left to right in each level, from top to bottom
- Pre-order processes root, followed by left subtree, followed by right subtree
- Post-order processes left subtree, followed by right subtree, followed by root

Using top-down decomposition you have three tasks in main:

1. Process command line arguments, set up file to process regardless of source, check if file is readable, set the basename for the output file, etc.
2. Build the tree
3. Traverse the tree using three different ways

The main function must handle the three functionalities. #1 should be handled inside of the main source file, functions for #2 should be in a separate source (buildTree.cpp), and functions for #3 should be in another separate source (traversals.cpp). Any node types should be defined in a separate header file (node.h).

For development purposes, do either 1 or 2 first. 3 should follow 2, first with one traversal only.

Processing keyboard input can be done in either of the following ways:

1. Read the input into a temporary file, after which the rest of the program always processes file input, the same as for when a file is provided
2. Set file pointer to stdin, then process always from the file pointer

Suggested files:

- `Makefile` (first part)
  - `CC = g++`
  - `CFLAGS = -g -Wall -std=c++11`
  - `OBJS = main.o buildTree.o traversals.o`
  - `TARGET = P0`
- `node.h, main.cpp, traversals.cpp, traversals.h, buildTree.cpp` and `buildTree.h`
- `main.cpp`

```
#include "node.h"
#include "traversals.h"
#include "buildTree.h"

int main(int argc, char* argv[]) {
  // process command line arguments and make sure file
  is readable, error otherwise
  // set up keyboard processing so that hereafter the
  input method is not relevant

  node_t *root = buildTree(file); // 'file' is the input file

  levelOrder(root);
  preOrder(root);
  postOrder(root);

  return 0;
}
```

A sample function for printing tree:

```
static void printParseTree(nodeType *rootP, int level) {
  if (rootP == NULL) return;
  printf("%*c%d:%s ",level*4,' ',level,NodeId.info); // assume some info
printed as string
  printf("\n");
  printParseTree(rootP->child1,level+1);
  printParseTree(rootP->child2,level+1);
}
```

Note that `printf("%*c", num, char)` prints *num*-1 spaces followed by *char*.

Suggested initial testing (you need to add further tests so that all functionality is tested):

1. Create test files:
   1. P0_test1 containing empty file
   2. P0_test2 containing one string, e.g.: aB$d
   3. Tests containing some strings with same number of character and some with different numbers of characters across same line and across multiple lines, e.g. P0_test3:

2Dogs c#T3 A**()/
li^n B zOo
!&lizARD C1 Zebra eAgle++ DoVe leOpard
Ant++ MON5-key tU;key

4. Tests with input errors, e.g.: lion Li_on~ Li?on (edit to make other errors for additional tests)

2. For each test file, draw by hand the tree that should be generated. For example, P0_test2 should create just one node with the key of '4' with one string 'aB$d'
3. Decide on invocations and what should happen, what should be output filenames if no error, and what the output files should look like - using the hand drawn trees for each file
4. Run the invocations and check against predictions
    1. $ P0 < P0_test1

       Error: Missing data. // Use some type of meaningful error message

    2. $ P0 P0_test2

       Outputs P0_test2.levelorder P0_test2.preorder P0_test2.postorder, each containing a single line of output:

       0 4 aB$d

    3. $ P0 P0_test3

       Outputs P0_test3.levelorder P0_test3.preorder P0_test3.postorder, each containing the information in the tree in the correct order and with correct indentations. For example, 'P0_test3.levelorder' should look like this:

```
0 5 2Dogs Zebra
    1 4 C#T3 li^n DoVe
    1 6 A**()/
        2 1 B
        2 8 !&lizARD
            3 3 zOo
            3 7 eAgle++ leOpard
                4 2 C1
```

Submission:

1. All source and header files
2. Makefile
3. Academic Integrity Statement
4. Readme (optional)
5. ***Do NOT include executable or .o files!***
6. Tar and zip files prior to submission (Academic Integrity Statement can be included in zipped package, uploaded separately, or a hard copy can be submitted in class)

Rubric:

Meaningful and concise comments throughout source code	20 pts.
Programming architecture and style (using selected department standards)	10 pts.
Program reads from a file and from the keyboard	10 pts.
Ordering of nodes in output files	30 pts.
Additional output formatting (missing level or key, incorrect spacing)	30 pts.

Programs that are missing a working makefile, fail to compile on hellbender, or fail to produce an executable named P0 will not be graded. Academic Integrity Statement is required to receive a grade.