CMPS 12B
Data Structures
Assignment 3

Binary Search Tree

The purpose of this assignment is to create a UNIX program that performs Binary Search Tree (BST) operations. The BST ordering will be lexicographic (alphabetic). For this assignment the left child's string value is **lexicographically less than** the parent, while the right child's string value is **lexicographically greater than** the parent node. Two nodes with the same string are not allowed in the tree. You will write the program in C and turn in all the files and a Makefile required to create the program: **BST**.

BST will parse a set of commands from **stdin** and write all messages to **stdout**.

The format of the input is:

<command> <optional argument>

Command	**Expected Action**	**Expected Output**
erase	Delete all the tree nodes and free any memory allocated for the tree; the tree is empty afterwards	"OK"
find <string>	Binary Search for the <string>	Found <string> *-or-* Could not find <string>
height	Compute Height of Tree	Height = <integer>
help	Prints list of commands	See in the example below
insert <string>	Insert <string> into the BST; a valid <string> is a sequence of one or more graphical characters as determined by isgraph(); (i.e., spaces and other non-printing characters are not allowed in <string>)	"OK" upon success; or an appropriate error message
print <traversal>	Traverse the tree and print the value of each node in the order requested by <traversal>, which may be any of the values "inorder", "preorder" or "postorder". If <traversal> is omitted, in order traversal is assumed	One line per node, each line is the string stored in that node
quit	Exit the program	None

Any number of spaces is allowed before the command; between <command> and <optional argument>; and after the <optional argument>.

Only one command can be entered on each line. Empty lines and those containing only spaces are ignored. You can limit the length of an input line (must be at least 150 characters) but then you must check and appropriately handle input lines that are too long. An invalid command or any error in how the command is entered must be detected and reported, in which case the command is not performed. The case of characters in the command name or <traversal> should be ignored.

You should test whether **stdin** is getting input from a user (terminal) rather than a file. If it is from a terminal, print a command prompt "`BST> `" before accepting input.

Here is an example interactive session ("$ " is used to represent the UNIX shell command prompt; user input is in boldface):

```
$ ./BST
BST>                        (nothing was typed on the line)
BST> add bear
Unrecognized command "add"
Enter "help" for description of each command
BST> help
Available commands are:
erase                  -- erase the current tree
find <string>          -- see if <string> is in the tree
height                 -- print the height of the tree
help                   -- print this list of commands
insert <string>        -- insert <string> into the tree
print [inorder|preorder|postorder]    -- print tree in specified order
quit                   -- quit the program
BST> insert Cat
OK
BST> INSERT Bottle_brush
OK
BST> iNsErT cat
Duplicate string "cat"
BST> insert     horse
OK
BST> find cAt
Found "Cat"
BST> insert   bird+box
OK
BST> Find rat
Could not find "rat"
BST> insert cat in the hat
Unexpected text after "cat" argument.
Enter "help" for description of each command
BST> insert ZEBRA!
OK
BST> height of great wall of China
Unexpected text after "height" command.
Enter "help" for description of each command
BST> height
Height = 3
BST> print OutOfOrder
Unrecognized argument "OutOfOrder"
Enter "help" for description of each command
BST> print InOrder
Bird+box
Bottle_brush
Cat
horse
ZEBRA!
BST> quit
$
```

You will create at least the files **BSTClient.c, BST.c** and **BST.h**. You can have additional files but all your BST routines must be contained in **BST.c.**

BST.h -- contains the BST node definition and BST function prototypes [supplied]
BST.c -- contains the BST manipulation routines
BSTClient -- contains `main()` and parsing routines (or you can put these in another file)

Download the starting version of **BST.h** from canvas. There are a few extra function there you can delete. You can also just use the definitions below. The initial contents are shown below. You will need to add comments that describe the purpose and contents of **BST.h**.

```
#include <stdio.h>

// Exported reference type
// Tree node
typedef struct BSTObj {
    char               *term;           /* string data in each block */
    struct BSTObj      *leftChild;      /* ptr to left child */
    struct BSTObj      *rightChild;     /* ptr to right child */
} BSTObj;

// add a new node to the BST with value, must allocate space for string
// no duplicates, case insensitive comparison of ordering
// return 0 for successful insert, -1 if the term was already in the tree
int insert(char *value,  BSTObj **pT);

// print the inorder traversal to out
void inorderTraverse(FILE *out, BSTObj *T);

// print the preorder traversal to out
void preorderTraverse(FILE *out, BSTObj *T);

// print the postorder traversal to out
void postorderTraverse(FILE *out, BSTObj *T);

// return TRUE if the item_to_find matches a string stored in the tree
int find(char *term_to_find, BSTObj *T);

// compute the height of the tree
int treeHeight(BSTObj *T, int height);

// remove all the nodes from the tree, doallocate space and reset Tree pointer
void makeEmpty(BSTObj **pT);
```

In **BST.c** you will implement the list manipulation routines defined in the **BST.h** file. Expect to do significant testing to validate the proper operation of each function.

Either in **BSTClient.c** or a separate file you should include the following macro definitions for strings written to **stdout**. You may change the names of the macros, but the output should be exactly the same or your grade will suffer!

```
// Output messages
#define COMMAND_PROMPT  "BST> "

#define HELP_PROMPT     "Enter \"help\" for description of each command\n"
#define HELP_MESSAGE    \
      "Available commands are:\n" \
      "erase                  -- erase the current tree\n" \
      "find <string>          -- see if <string> is in the tree\n" \
      "height                 -- print the height of the tree\n" \
      "help                   -- print this list of commands\n" \
      "insert <string>        -- insert <string> into the tree\n" \
      "print [inorder|preorder|postorder] -- print tree in specified order\n" \
      "quit                   -- quit the program\n"

#define MSG_NONE        "OK\n"
#define MSG_FOUND       "Found \"%s\"\n"
#define MSG_HEIGHT      "Height = %d\n"

#define ERR_BAD_STR     "Invalid character (decimal value %d) in string\n"
#define ERR_DUPLICATE   "Duplicate string \"%s\"\n"
#define ERR_EXTRA_ARG   "Unexpected text after \"%s\" argument
. \n" HELP_PROMPT
#define ERR_NEED_ARG    "Command requires an argument\n" HELP_PROMPT
#define ERR_NOT_FOUND   "Could not find \"%s\"\n"
#define ERR_TOO_LONG    "Input line too long; length must not exceed %d characters\n"
#define ERR_UNK_ARG     "Unrecognized argument \"%s\"\n" HELP_PROMPT
#define ERR_UNK_CMD     "Unrecognized command \"%s\"\n" HELP_PROMPT
```

Each function definition must include a comment block as a header that describes any requirements for using the function, any side effects, return values, any error conditions that can result and how you handle these error conditions. Any preconditions assumed by the function should be documented as well. In addition to the function header, you should include a file header comment block which describes the contents of the file. It is acceptable to use either "/* … */" or "//…" comments for your headers but you must be consistent or you will lose points.

Implementation Notes

1. Each node in the BST will contain a string. The ordering of nodes is determined by a lexicographic string comparison, that is, their dictionary order. Case of letters should be ignored (for example, the strings "dog" and "DOG" compare the same). You can easily determine the ordering using the `strcasecmp ()` function in the C standard library. You will need to include **string.h**.
2. The `isgraph()` function in the C standard library can be used to test whether there are any invalid characters in <string>. You will need to include the header file **<ctype.h>**.
3. To determine whether standard input is coming from a file or from the terminal, you can use the `isatty()` function, which is defined in **<unistd.h>**:

```
int input_from_terminal = isatty(fileno(stdin));
…
while (!feof(stdin)) {
    // read command from input; if on a terminal, prompt first:
    if (input_from_terminal)
        fputs(COMMAND_PROMPT, stdout);
…
```

What to turn in
You should turn in (using canvas) an **assignment3.zip** file that contains your **Makefile, README, BSTClient.c, BST.c** and **BST.h** (plus any other files you may have created). You are responsible for creating your own test cases to exercise the code and verify correct operation yourself; this is part of the assignment. You are not allowed to share test cases with students: You can *talk* about testing strategies and cases but you must not *share code* that exercises the list functions.

Your project will be evaluated with the weighting of 70% for code correctness and 30% for programming style, documentation and clarity. Your file headers must include your name, date, and class information as well. Non-compiling programs will earn 0 points as they are impossible to evaluate.

Some advice on getting started

1. Create your **Makefile**; it must support a target named **BST.** In Lab4 you were given a generic Makefile style for any ADT. You can create the initial **BST.c** file with functions that are stubs (doesn't do anything but compile).
2. Now create **BSTClient.c**. Write a basic command processing loop that just implements the **help** and **quit** commands, and correctly reads input from a terminal or from a file.
3. When you are satisfied that works correctly, add the "erase" and "height" commands.
4. Now add code to parse the **insert**, **find**, and **print** commands.
5. Edit the **BST.c** file with all the functions declared in **BST.h**. You might want to start by having each function just contain a printf() saying the function was called (e.g., "insert was called with string <string>").
6. Modify the code in **BSTClient.c** to call the appropriate functions declared in **BST.h**. Modify the **Makefile** so that the **BST** target depends on **BST.o**.
7. Test your program to confirm that when you enter a command, the corresponding function in **BST.c** is being called.
8. Implement the `insert()` function and one of the traversal functions (I suggest `inorderTraverse()`). Test your code by alternating between **insert** and **print** commands to confirm the tree is being built correctly.
9. Now implement another function (perhaps one of the other traversals) and test again.
10. Continue to incrementally add new functions and test until the program is complete.

As an alternative, you could start by writing the BST functions first and just create a simple `main()` program to test their behavior. The important concept here is to start small, get your code to compile and run, and keep your code compiling and functioning while you continue to add functionality and test each improvement. Small steps is the key to success!