CS29003 ALGORITHMS LABORATORY
ASSIGNMENT 3
Last Date of Submission: August 2, 2018

This assignment is about binary search trees that store strings. We will assume that the strings contain only the lower-case English letters and are ordered lexicographically, i.e., the order in which words are organized in a dictionary. Note that this enables us to compare two strings: For two strings s_1 and s_2, we will say $s_1 < s_2$ if s_1 is before s_2 in the lexicographical ordering. Examples:

- ▷ "abcd" < "acd",
- ▷ "abc" < "abcd".

You will first take some strings as inputs from keyboard, and insert them into a binary search tree in the same order as you receive them. After that you will print out the inorder traversal of the tree that you have just constructed.

Then you will take a pattern string as input from keyboard. After that you will print all the strings in the tree that has this pattern as a prefix, in lexicographic order. Example: Let the pattern string be "abc". Let the strings in your tree be "abdes", "abcz", "abc", "bc". Then your program should print "abc" and "abcz" in this order (see sample output later).

To accomplish this, please follow the algorithm that we now outline. At any stage in the algorithm, assume that you have arrived at a node v. You check whether the string in v contains the pattern as a prefix. If not, then notice that one of the subtrees of v cannot have any string with the pattern as prefix. Example: if the pattern is "abc" and the string in v is "aba", then the left subtree of v cannot have a node with a string with prefix "abc". So you discard the left subtree entirely and go to the right child of v. If the string in v contains the pattern as prefix, then both the subtrees of v can potentially contain strings with the pattern as prefix. In such a case, you first explore the left subtree and enumerate all such strings there in the right order. Then you come back to v and print the string in v. Then you go to the right child of v and continue. Remember to take care of boundary cases (one or both of the children of v being missing).

It is relatively easy to write down a recursive algorithm for this task. We, however, insist that you do not use recursion. Furthermore we require you to use only $O(1)$ extra memory. In particular, declaring variable-sized arrays and using stack, queue or linked-lists are disallowed. Each node will have pointers to its children and parent; you can use them to navigate the tree.

Finally you write a function to delete the nodes with strings containing the pattern as prefix. Your function must delete the nodes in a bottom-up fashion, i.e., it deletes a node only after all the relevant nodes in the subtree rooted at it have been deleted. You can use recursion to write this function. But we insist that you allocate only constant amount of memory in the body of the program by explicit declarations. In particular, use of variable length arrays is disallowed. After deleting all those nodes, print the inorder travsersal of the resultant tree.

Implementation: Use the following structure declaration to represent a node of the tree.

```
typedef struct treenode
{
        char word[100];
        struct treenode *leftchild;
        struct treenode *rightchild;
        struct treenode *parent;
} NODE, *NODEPTR;
```

The character array **word** contains the string that is stored in that node, terminated by NULL. Pointers **leftchild**, **rightchild** and **parent** contain the addresses of the left child, right child and parent of the node respectively.

Assume that all strings are of length at most 99. Store all strings as null-terminated character sequences in character arrays of size 100.

Part 1: Create and print inorder traversal of binary search tree

Write a function **insert()** to insert a new node in the tree. The prototype is as follows

NODEPTR insert(NODEPTR)

insert() takes in as argument a pointer to the root of the tree. It takes a string as input through the keyboard. Then it creates a node containing the input string, and inserts it in the BST. Assume that the strings are all distinct. Finally, **insert()** returns a pointer to the root. You can use the function **strcmp**() (defined in string.h) to compare strings.

Write a function **inorder()** to print the inorder traversal of a binary search tree. Feel free to use recursion. The prototype is as follows.

void inorder(NODEPTR *node)

inorder() takes in a pointer to a node as argument, and prints the inorder traversal of the subtree rooted at that node.

Part 2: Print all strings with a given prefix

Write a function with the following prototype.

void find_extensions(NODEPTR root, char pattern[100])

find_extensions() takes in the pointer to the root of the BST and a null-terminated character sequence in a character array (pattern) as arguments. It prints all strings in the tree with the pattern as prefix, in lexicographic order. It uses $O(1)$ space. Stick to the algorithm described in the beginning.

Part 3: Delete all nodes with strings with the given string as prefix

Write a function **delete_prefix()**to delete all nodes with strings containing the pattern as prefix, in a bottom-up fashion. The prototype is as follows.

NODEPTR delete_prefix(NODEPTR root, char *pattern)

Recall from the discussion in the beginning that you are allowed to allocate only constant amount of memory by explicit declarations in the body of the function. You are not allowed to use variable length array. Use of recursion is allowed.

main()

In **main()** make the following declarations.

NODEPTR root=NULL;

This creates an empty tree. The pointer variable **root** is meant to contain the address of the root node of the tree, and is initialized to NULL.

Next, take as input through keyboard the number of strings n. Then, call **insert(root)** n times to build the tree. After each call, assign the return values back to **root**.

Next, print the inorder traversal of the tree by calling **inorder(root)**.

Next, take a pattern as input through keyboard. Store it in a character array **pattern** as a null-terminated sequence.

Next, call **find_extensions(root, pattern)**.

Next, call **delete_prefix(root, pattern)**.

Finally, call **inorder(root)**.

Sample output

Enter the number of words:8
Enter word:hello
Enter word:page
Enter word:pagal
Enter word:cat
Enter word:dog
Enter word:pagoda
Enter word:random
Enter word:pagan

Inorder traversal:
cat dog hello pagal pagan page pagoda random

Enter the pattern: pag
All extensions of ”pag” in lexicographic order:
pagal pagan page pagoda

Tree after deletion is :
cat dog hello random