CEN235 – DATA STRUCTURES
Fall – 2018

Lab 8: Implement Dictionary Using Binary Search Tree

Preliminary Work:

- Dictionary can be implemented using binary search tree. A binary search tree is a binary tree such that each node stores a key of a dictionary.
- Key 'k' of a node is always greater than the keys present in its left sub tree.
- Similarly, key 'k' of a node is always lesser than the keys present in its right sub tree.

Example:

```
          well
         /    \
      boot    xmas
     /    \       \
   air   bus      zebra
```

In the above example, keys present at the left sub-tree of the root node are lesser than the key of the root node. And also, the keys present at the right sub-tree are greater than the key of the root node.

- To insert an element in a binary search tree, check whether the root is present or not. If root is absent, then the new node is the root.
- If root node is present, check whether the key in new node is greater than or lesser than the key in root node.
- If key in new node is less than the key in root, then traverse the left sub-tree recursively until we reach the leaf node. Then, insert the new node to the left(newnode < leaf)/right(newnode > leaf) of the leaf.
- If the key in new node is greater than the key in root, then traverse the right sub-tree recursively until we reach the leaf node. Then, insert the new node to the left(newnode < leaf)/right of the leaf

```
InsertionInBST(T, newnode)
y<-NULL
x <- root[T]
while x != NULL
    y<-x
    if  key[z] < key
        then x <- left[x]
        else  x <- right[x]
parent[newnode] <- y
if y == NULL
   then root[T] <- newnode
else if key[newnode] < key[y]
```

then left[y] <- newnode
else right[y] <- newnode

Insert "yell" to the below binary search tree.

```
      workload
       /     \
      boot    xmas
     /   \        \
   air   bus      zebra
```

```
      workload
     /          \
   boot          xmas
  /   \       /       \
air   bus    yell     zebra
```

To delete a node from binary search tree, we have three different cases.
Node X has no children
Node X has one child
Node X has two children

Case 1:
If X has no children

```
      workload
     /          \
   boot         xmas
  /   \       /       \
air   bus    yell     zebra
```

Delete "zebra" from above binary search tree.

```
      workload
     /          \
   boot         xmas
  /   \       /
air   bus  yell
```

Case 2:
If X has only one child, then delete x and point the parent of x to the child of x.

```
      workload
     /          \
   boot          xmas
  /   \       /
air   bus    yell
```

Delete "xmas" from the above binary search tree.

```
            workload
          /          \
       boot          yell
      /    \
    air   bus
```

Case 3:
If X has two children, then find its successor 'S'. Remove 'S' from the binary search tree. And replace X with 'S'

```
            workload
          /          \
       boot          xmas
      /    \       /      \
    air   bus    yell     zebra
```

Remove "workload" from the above binary search tree. The successor for "workload"(smallest element in the right subtree of "workload") is "yell". Remove it and replace "workload with "yell".

```
              yell
            /      \
        boot     xmas
       /    \        \
     air   bus      zebra
```

Tasks:

Run the BST code to insert items to the dictionary, delete items from the dictionary, and search any item in the tree, and print the tree using inorder traversal function provided.

Task-1: Print the tree so that you can see the tree like examples given above. Here, you can use simple front slash character (/) to show relationship of tree nodes. Here, maybe you can use preorder or postorder tree traversal algorithms. **[20pts]**

Inorder Traversal (Given in the code)	*Algorithm Inorder(tree)* *1. Traverse the left subtree, i.e., call Inorder(left-subtree)* *2. Visit the root.* *3. Traverse the right subtree, i.e., call Inorder(right-subtree)*
Preorder Traversal	*Algorithm Preorder(tree)* *1. Visit the root.* *2. Traverse the left subtree, i.e., call Preorder(left-subtree)* *3. Traverse the right subtree, i.e., call Preorder(right-subtree)*
Postorder Traversal	*Algorithm Postorder(tree)* *1. Traverse the left subtree, i.e., call Postorder(left-subtree)* *2. Traverse the right subtree, i.e., call Postorder(right-subtree)* *3. Visit the root.*

Task-2: Implement the following functions [30pts]:

- Add a new function to the given program so that you can count the number words in your binary search tree (i.e. dictionary) **[10pts]**
- Compute depth of your tree **[10pts]**
- Mirror function **[10pts]**

  Change your BST tree so that the roles of the left and right pointers are swapped at every node. So the tree...

```
    4
   / \
  2   5
 / \
1   3
```

  is changed to...

```
    4
   / \
  5   2
     / \
    3   1
```

Task-3: Create a file, copy some of random English paragraphs from Internet and paste them into your file, so that you have a random text file **[50pts].**

Parse this file word by word and each new word to your dictionary. Count the number of distinct words in your dictionary (i.e., the number of nodes in the dictionary). What is the depth of your tree now? Take the mirror of your dictionary and print it on the screen.