Homework 5

Solution:

Task 1:

From our class and textbook, we learn that in Binary Search Tree, all keys are distinct, and any key in left sub-tree is smaller, any key in right sub-tree is greater. Therefore, it is clear that the time consumption of operations like Search, Insert, Delete are related to the Binary tree, i.e., in most of cases, the major time consumption are $O(h)$, where the h is the height of the tree.

Consider the two options when allow duplicated keys in the BST:

a. Add the node.count to save the duplicated keys, which is similar to the node.value feature. In this case, the height of the tree will not be changed.

b. There are two options to handle duplicated in the this case. 1). The duplicated keys will be assign to one of the side of the sub-tree, i.e., left sub-tree or right sub-tree. 2). The duplicated keys will be randomly assign to the sub-tree, i.e., it can be assign to both left sub-tree and right sub-tree under the same probability. The height of the tree will be added in both cases.

Therefore, **the option a is more efficient than the option b**. Because, In general, the time complexity is positive to the height of the tree. In the case b, the height of tree is greater when exit duplicated keys, which causes any operation for the tree consumes more comparison and assign processes. So the complexity is greater than case a.

Task 2:

2.1 Implement of both cases

To simply the program, in the case b, I assign all the same value keys to the left sub-tree. Four class are implied in my program:

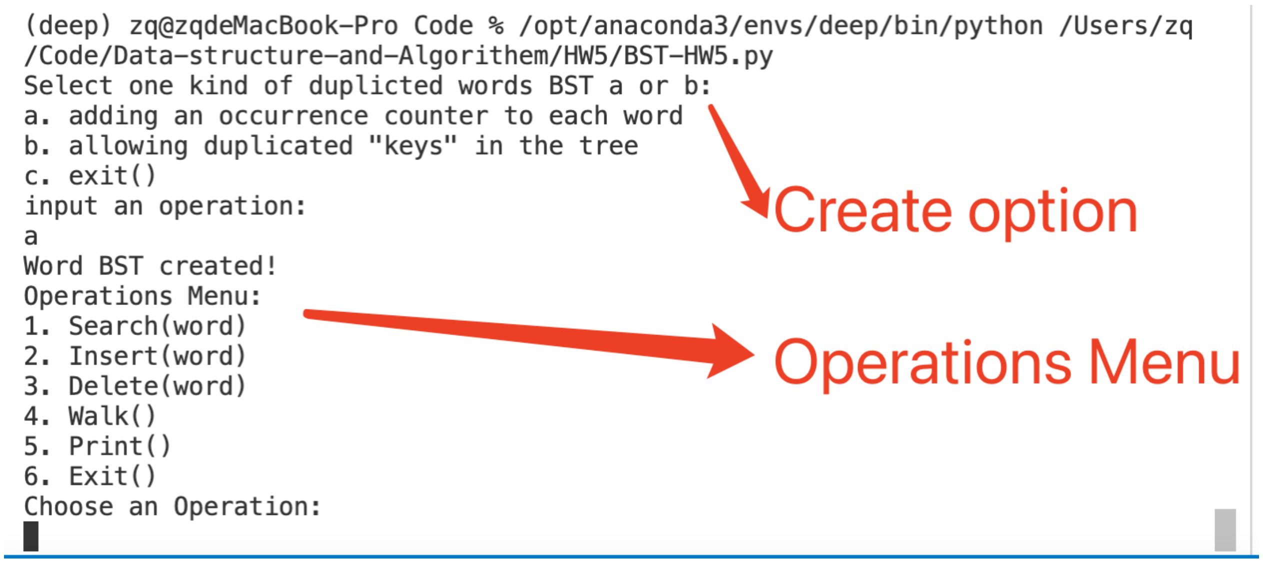
- The class of Node in case a, add the count feature.
- The class of Operations for case a.
- The class of Node in case b, which is same as the BST in our class.
- The class of Operations for case b.

More detail please see the source code I uploaded.

In the program, I create a simple user interface for the operations interactively, please see the figure 1.

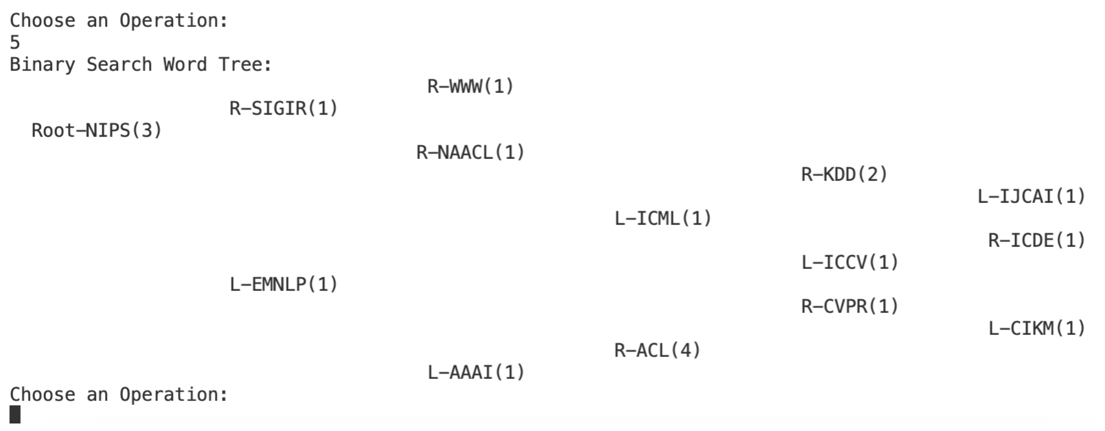
I print the tree from left to right, and add preceding string to identify the left and right, where the R means the right of the node and L means the left of the node. Please see the figure 2

All of the test of the other operations required please see figure 3. More detail, please run and check the source code and the output screenshot.



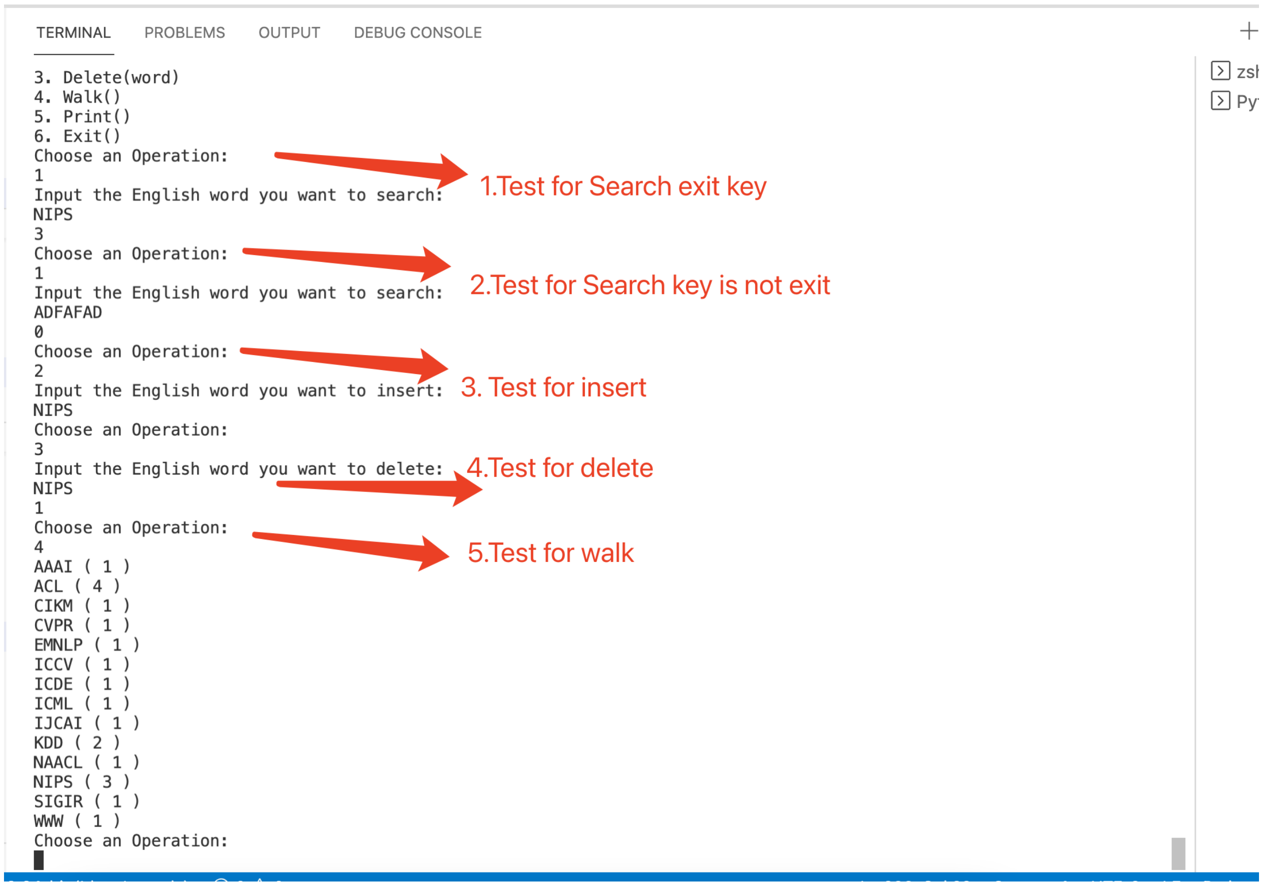
```
(deep) zq@zqdeMacBook-Pro Code % /opt/anaconda3/envs/deep/bin/python /Users/zq
/Code/Data-structure-and-Algorithem/HW5/BST-HW5.py
Select one kind of duplicted words BST a or b:
a. adding an occurrence counter to each word
b. allowing duplicated "keys" in the tree
c. exit()
input an operation:
a
Word BST created!
Operations Menu:
1. Search(word)
2. Insert(word)
3. Delete(word)
4. Walk()
5. Print()
6. Exit()
Choose an Operation:
```

Figure 1: The simple user interface for the operations interactively.



```
Choose an Operation:
5
Binary Search Word Tree:
                                      R-WWW(1)
                    R-SIGIR(1)
  Root-NIPS(3)
                                     R-NAACL(1)
                                                                        R-KDD(2)
                                                                                        L-IJCAI(1)
                                                       L-ICML(1)
                                                                                         R-ICDE(1)
                                                                        L-ICCV(1)
                    L-EMNLP(1)
                                                                        R-CVPR(1)
                                                                                         L-CIKM(1)
                                                       R-ACL(4)
                                      L-AAAI(1)
Choose an Operation:
```

Figure 2: The print of the tree.



```
3. Delete(word)
4. Walk()
5. Print()
6. Exit()
Choose an Operation:
1
Input the English word you want to search:
NIPS
3
Choose an Operation:
1
Input the English word you want to search:
ADFAFAD
0
Choose an Operation:
2
Input the English word you want to insert:
NIPS
Choose an Operation:
3
Input the English word you want to delete:
NIPS
1
Choose an Operation:
4
AAAI ( 1 )
ACL ( 4 )
CIKM ( 1 )
CVPR ( 1 )
EMNLP ( 1 )
ICCV ( 1 )
ICDE ( 1 )
ICML ( 1 )
IJCAI ( 1 )
KDD ( 2 )
NAACL ( 1 )
NIPS ( 3 )
SIGIR ( 1 )
WWW ( 1 )
Choose an Operation:
```

Figure 3: Test for Search, Insert, Delete operations

Table 1: Counters in the search operation.

	Comparison Counter	
Search Item	a(Count)	b(Duplicte)
AAAI	3	4
ACL	3	11
CIKM	4	7
CVPR	3	7
EMNLP	2	10
ICCV	4	6
ICDE	4	7
ICML	3	7
IJCAI	4	8
KDD	3	9
NAACL	2	10
NIPS	1	20
SIGIR	1	2
WWW	1	3
NOT EXIST	54	115

2.2 Compare empirically

To compare the efficiency, the time consumption is important. Because the comparison costs most of time, I add the counter after every comparison. Discussion of each operation:

1) Search

Counters have been added in both of the search operations. I search every word in the tree and one word not in the tree. The test result please see the table 1. **It is obvious that the counter in a is less than in b**. Because in the case a, it is similar with the BST without duplicated key, the add step is to return the node.count. But in case b, the first step is to find the key, and then search the left sub-tree to find other value same as the duplicated key. The additional search step increase the comparison. The test result is same as the analysis.

2) Insert

The test result please the table 2 . The result also shows that the a is more efficient. Because in the case a, if the key is exist, the insert is to add the count for the exist key; if the key is not exist, the condition is similar to the case b. But the height of the tree a in less than the case b. So the test result is same.

3) Delete

The test result of delete please table 3. The first step in the delete is to identify whether the key exist in the tree, so the major consumption of delete is search. If the key exist in the tree, the occurs need to be minus in case a, and in the case b, the tree need to be adjusted.

Table 2: Counters in the insert operation.

	Comparison Counter	
Insert Item	a(Count)	b(Duplicte)
AAAI	55	118
ACL	55	125
CIKM	56	121
CVPR	55	121
EMNLP	54	120
ICCV	56	120
ICDE	56	121
ICML	55	121
IJCAI	56	122
KDD	55	123
NAACL	54	120
NIPS	53	121
SIGIR	53	116
WWW	53	117

Table 3: The plot of Length and Counters.

	Comparison Counter	
Delete Item	a(Count)	b(Duplicte)
AAAI	9	14
ACL	11	56
CIKM	20	35
CVPR	13	33
EMNLP	13	39
ICCV	18	27
ICDE	20	35
ICML	16	39
IJCAI	21	44
KDD	14	46
NAACL	7	40
NIPS	5	24
SIGIR	5	11
WWW	9	9

4) Walk, Print

The tree walk and print is go through the whole tree and print the each node in the tree. Because the height of tree in case a is less than in case b. So it is obvious that case a is more efficient.

In a word, adding an occurrence counter to each word (option a) is more efficient than allowing duplicated "keys" in the tree (option b). But I only test the tree in a limited height. If the duplicated key randomly assign to the left and right sub-tree and the structure of tree is special, as the height of tree increased, the situation may be more complex (the case b has some probability to be more efficient than case a). I believe the discussion is enough in my report to derive this conclusion: in the most of cases, the option a is more efficient.