COMP2119 Introduction to Data Structures and Algorithms
Programming Assignment **Due Date:** 7pm, 15th December 2017

Please send email to **xli2@cs.hku.hk** if you have any questions about the description, judging rule, test cases, or datasets.

Course Outcomes

- **[O4]. Implementation**

1 Binary search tree operation: *Splaying*

Consider a set of elements organized as a binary search tree (BST). The elements in the set may be of different popularity. That is, a few elements are searched for much more frequently than others. In order to improve the average search time, one can consider moving an element in the set that is being searched for to the root of the BST (so that if the same element is searched for again soon, the element is likely close to the root of the BST and is thus located with fewer node inspections).

Given a node x in a BST T, let $\texttt{splay}(x)$ be an operation that transforms T into another BST T' such that (1) T and T' contain the same set of elements and (2) x is the root of T'. Figure 1.1 shows an example of the splay operation.

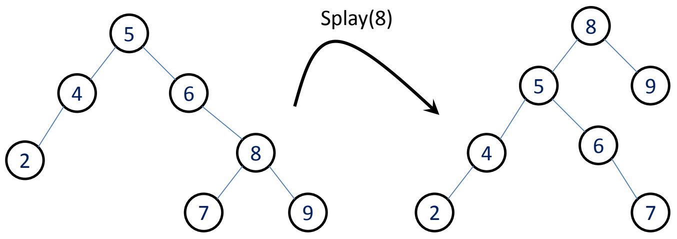

Figure 1.1: Splaying on node 8.

The splay operation can be implemented using rotations. Specifically, given a node x, if x is not the root of a tree, let p be x's parent. We consider two cases:

1. x is a left child. In this case, we execute $\textit{RightRotate}(p)$.
2. x is a right child. In this case, we execute $\textit{LeftRotate}(p)$.

Note that the above step moves x one level up the tree. Splaying can thus be done by repeating the above step until x is moved all the way to the root. For example, to transform the tree T to T' in Figure 1.1, we perform $\textit{LeftRotate}(6)$, followed by $\textit{LeftRotate}(5)$.

2 Basic BST vs. BST with Splaying

We consider two ways of maintaining a set of elements using BST.

- Basic BST (*BST-B*). We organize elements in a basic BST using the operations (insert, search, delete) discussed in class.
- BST with Splaying (*BST-S*). Same as *BST* except that whenever an element x is inserted or searched, we perform `splay`(x) right after the insert/search.

For example, consider the following sequence S_0 of operations (which includes 8 inserts and 8 searches):

```
insert 6
insert 1
search 1
insert 3
insert 4
search 4
search 3
search 4
insert 7
search 4
insert 8
insert 9
search 9
search 4
insert 2
search 8
```

Figures 2.2(a) and (b) show the result of applying the operation sequence S_0 on an initially empty BST based on strategies *BST-B* and *BST-S*, respectively.

Let us define the *cost* of a search as the the number of nodes inspected during a search. For example, with respect to the right BST shown in Figure 1.1, the search cost of node 4 is 3 units because the search of node 4 involves inspecting three nodes 8, 5, 4 in that order. Similarly, the search costs of nodes 8 and 7 are 1 and 4, respectively.

Given an operation sequence S, the total search cost of the sequence is the sum of the costs of the search operations mentioned in S assuming the sequence is applied to an initially empty BST. The total search cost depends on the BST strategy used. For example, with *BST-B*, the BST looks like that shown in Figure 2.3 after the first 9 operations (i.e., after operation “insert 7”) are done. The 10th operation, “search 4”, has a cost of 4 units. With the *BST-B* strategy, the search costs of the 8 searches in sequence S_0 are respectively, 2, 4, 3, 4, 4, 4, 4, 3 for a total of 28 units. With the *BST-S* strategy, the costs are 1, 1, 2, 2,

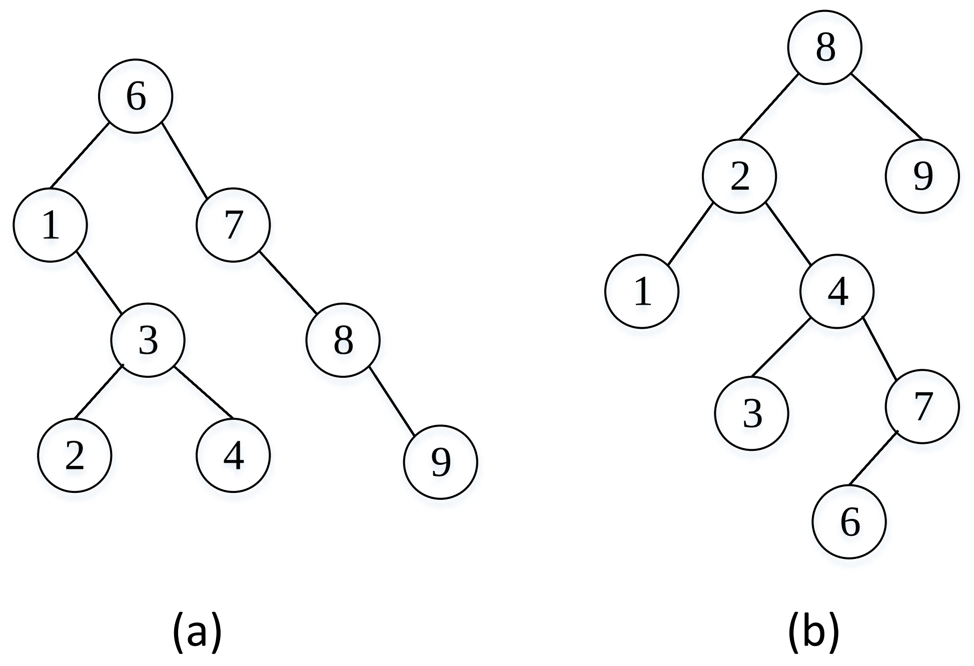

Figure 2.2: The results based on strategies (a) *BST-B* and (b) *BST-S*

2, 1, 3, 4 for a total of 16. For the sequence S_0, the *BST-S* strategy has a much smaller search cost compared with *BST-B*. You are advised to work out the search costs of the above example to make sure that you understand the discussion and definitions.

3 Program

The objective of your program is to compute the total search costs of a given operation sequence S under the *BST-B* and the *BST-S* strategies. The I/O specifications are given below:

3.1 Input Format

The input file is a text file. Each line (except the last) in the text file is either

```
insert x
```

or

```
search x
```

where `x` is a positive integer specifying the value of an element. You can assume that all

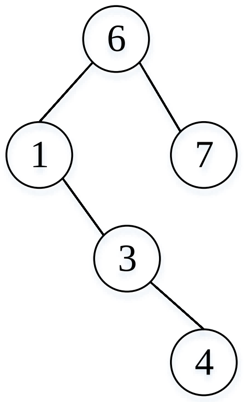

Figure 2.3: The BST based on *BST-B* strategy after the first 9 operations

search operations are successful (i.e., the element being searched for has been previously inserted into the tree). No duplicate values will be inserted.

The last line in the input file is:

```
END
```

3.2 Output Format

Your program should output two comma-separated numbers in the form:

```
a,b
```

where `a` and `b` are the total search cost of the operation sequence S under *BST-B* and *BST-S*, respectively. (Note that there should not be white space between `a` and `b`, such as "`a,␣b`").

4 Sample Inputs and Outputs

Example 1:
Input
insert 1
insert 2
search 2
END
Output
2,1

Example 2:
insert 1

```
insert 2
search 2
insert 3
search 3
END
```

Output

```
5,2
```

5 Others

Languages. We only accept C++ programming languages.

Node Structure To facilitate implementing rotation, a tree node structure should include a *parent pointer* in addition to the left-child pointer and the right-child pointer. (Note that the parent pointer of the root is NULL.)

Judging. Please note that your solution is automatically judged by a computer based on 10 test cases with each 10 marks. Solutions that fail to compile or execute get 0 points.

Self Testing. You should test your program by yourself using the provided sample input/output file. The sample input/output is **different** from the ones used to judge your program, and it is designed for you to test your program by your own. Note that your program should always use standard input/output. To test your program in Windows:
1. Compile your program, and get the executable file, "main.exe"
2. Put sample input file, "input.inp", in the same directory as "main.exe"
3. Open command line and go to the directory contains "main.exe" and "input.inp"
4. Run main.exe < input.inp > myoutput.oup
5. Compare myoutput.oup with sample output file.
6. Your output needs to be **exactly** the same as the sample output file.

To test your program in Linux or Mac OS X:
1. Put your source code "main.cpp" and sample input file "input.inp" in the same directory.
2. Open a terminal an go to that directory.
3. Compile your program by "g++ main.cpp -o main"
4. Run your program using the given input file, "./main < input.inp > myoutput.oup"
5. Compare myoutput.oup with sample output file.
6. Your output needs to be **exactly** the same as the sample output file.
7. You may find the **diff** command useful to help you compare two files. Like "diff -w myOutput.inp sampleOutput.oup". The $-w$ option ignores all blanks (SPACE and TAB characters)

Note that myoutput.oup file should be **exactly** the same as sample output. Otherwise it will be judged as wrong.

Sketch file. We provide a source file as a sketch. You write your code based on that. You can also write your own code as long as the format is correct.

Test files. We put the test cases under *test* directory. input*.txt are input files and output*.txt are sample answer.

Submission. Please submit your source file through moodle. You should submit the source file only (**without** compression). Please write your code in one file named in format of *university_number.cpp*, e.g. *1234567890.cpp*. **Do not use multiple files**.

sketch.cpp

```cpp
#include <iostream>
#include <string>
#include <assert.h>


using namespace std;

typedef struct node{
    //your code starts here
}TreeNode;

int algorithmBSTB(){
    //your code starts here
}

int algorithmBSTS(){
    //your code starts here
}

void calculateCost(){
    cout<<algorithmBSTB()<<","<<algorithmBSTS()<<endl;
}

int main(){
    //you code starts here
    return 0;
}
```