GIT Department of Computer Engineering
CSE 222/505 - Spring 2021
Homework # Report

Mehmet Acar
1801042095

1. SYSTEM REQUIREMENTS

Functional Requirements

1-For Part 1, SkipList.java's methods and AVLTree.java's methods are requirements. SkipList.java's methods are computeMaxCap() , logRandom() , search() , find() , add() , remove() and descendingIterator() . AVLTree's methods are two overloading add() methods , rebalanceLeft() , rebalanceRight() , incrementBalance() , decrementBalance() , traverse() , traverse2() , iterator() , headSet() and tailSet() and also, it needs all methods which are related with BinarySearchTree.

2-For Part 2, all BinarySearchTree methods, all BinaryTree() methods, BinarySearchTree() methods are requirements.

3-Part 3 needs all methods which are related with BinarySearchTree, needs all methods of Red Black Tree, needs all methods of B-tree and needs all methods of SkipList.

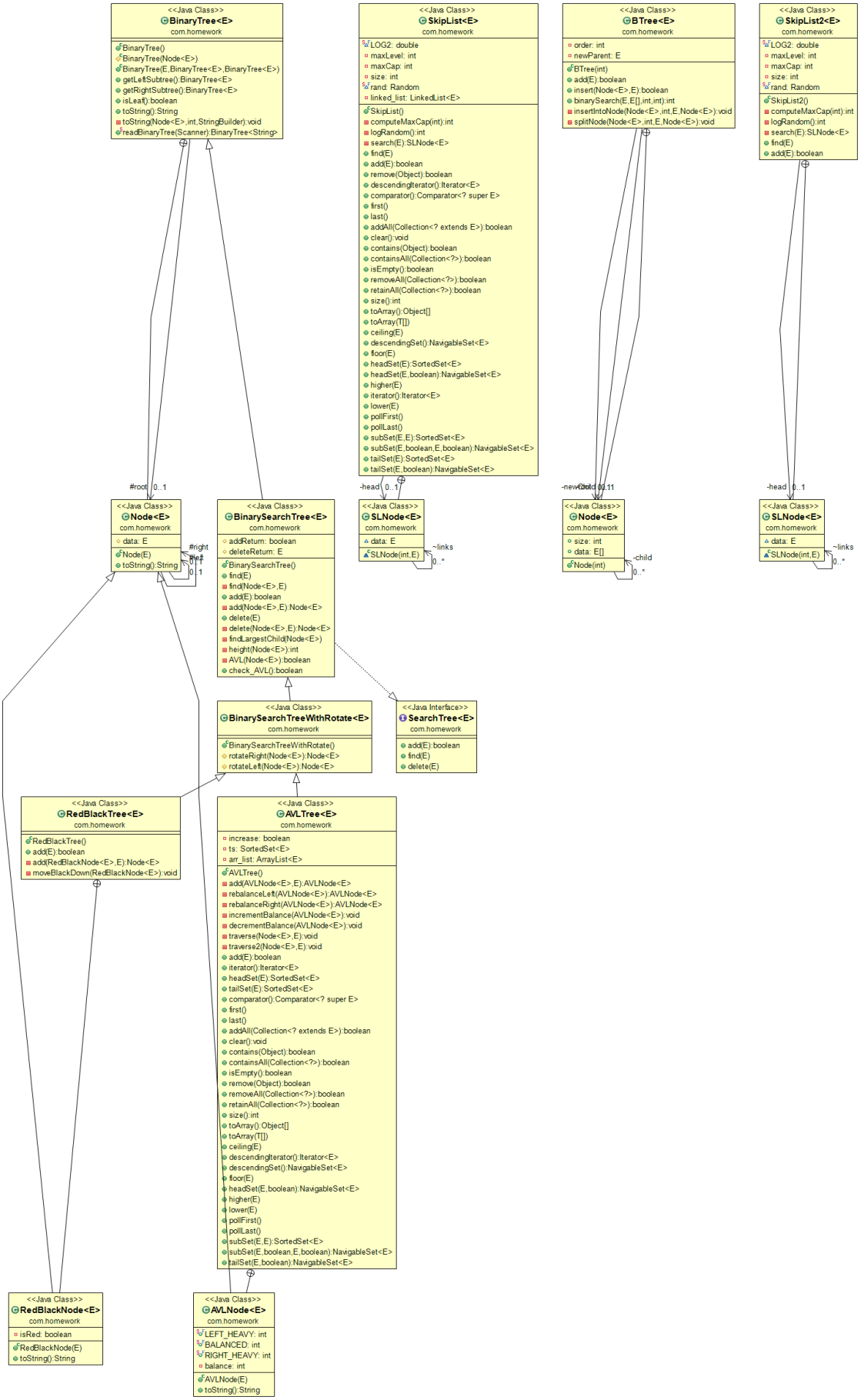
Non-Functional Requirements

1-Security

2- Maintainability

3-Performance

2. CLASS DIAGRAMS



Because of lots of data fields and methods, class diagram is not clear. I put png file of class diagram to zip.

3. PROBLEM SOLUTION APPROACH

PART 1

Problem is implementing some methods of NavigableSet interface using Skip List and AVL Tree.
In order to solve implementing some methods of NavigableSet interface using SkipList, when I wrote SkipList.java, I placed Skip List's add method from book to NavigableSet interface's add method, I placed SkipList's remove method from book to NavigableSet interface's remove method.

In order to solve implementing some methods of NavigableSet interface using AVLTree, when I wrote AVLTree.java, I placed AVL Tree's add method from book to NavigableSet interface's add method.
Also, some extra methods are given which are headset and tailSet. Book does not have these methods in AVLTree.
In order to implement headset and tailSet, I used NavigableSet interface's headset and tailSet method. Both headSet and tailSet methods return sortedSet. Therefore, I used sortedSet data field in order to keep numbers. In headSet method, I called traverse method. In traverse method, program checks that AVLTree has smaller number than parameter toElement number and when it finds these numbers, add to sortedSet. In tailSet method, I called traverse2 method. In traverse2 method, program checks that AVLTree has bigger number than parameter fromElement number and when it finds these numbers, add to sortedSet.

PART 2

Problem is writing method that checks BinarySearchTree is an AVLTree or not.

In order to solve this problem, when I wrote BinarySearchTree.java, I add some extra methods to the book's methods in order to check BinarySearchTree is an AVL or not. I add check_AVL(), AVL() and height() methods. In check_AVL() method, root of BinarySearchTree is sending to AVL() method and according return value of AVL() method, check_AVL() method returns true or false. In AVL() method, I sent left node of root and right node of root to height() method in order to find heights of both direction. If both directions' height difference is less or equal to 1, AVL() method returns true, otherwise returns false.

PART 3

Problem is comparing insertion performance of data structures(Binary search tree, Red-Black tree, 2-3 tree, B-tree, Skip list) and calculating the average running time for each data structure and problem size(10000, 20000, 40000, 80000). Running time is calculating while inserting 100 extra random numbers into the structures.

In order to solve this problem, I toke book implementation of each data structure. For each size, I apply insertion operation 10 times for each data structure for each size and when 100 random numbers inserting into the structures, I sum up running time of 10 insertion operation for each data structure for each size. After that, I divide sum result to 10 and I print average running time of data structures for each size.

4. TEST CASES

1- **If new element is inserted to Skip List in Part 1**

```
10 is inserted to Skip List
```

2- **If element is deleted from Skip List in Part 1**

```
10 is deleted from Skip List
```

3- **After SkipList's descendingIterator() method is called, iterator next value is printed in Part 1**

```
Skip list iterator next(): 30
```

4- **If new element is inserted to AVL Tree in Part 1**

```
1 is inserted to AVL Tree
```

5- **After AVL Tree's iterator() method is called, iterator next value is printed in Part 1**

```
AVL Tree iterator next(): 1
```

6- **If AVL Tree's headSet() method is called, smaller number from given number is printed in Part 1**

```
In AVL Tree, smaller number from 4(headSet): [1, 2, 3]
```

7- **If AVL Tree's tailSet() method is called, bigger number from given number is printed in Part 1**

```
In AVL Tree, bigger number from 4(tailSet): [5, 6, 7]
```

8-When new BinarySearchTree is created, following sentence is printed in Part 2

```
New Binary Search Tree is created
```

9- If new element is inserted to Binary Search Tree in Part 2

```
12 is inserted Binary Search Tree
```

10- When BinarySearchTree's AVL_check() method is called, if given binary search tree is AVL Tree, following sentence is printed in Part 2

```
The Binary Search Tree is AVL Tree
```

11- When BinarySearchTree's AVL_check() method is called, if given binary search tree is not AVL Tree, following sentence is printed in Part 2

```
The Binary Search Tree is not AVL Tree
```

12- After data structures are filled by proper size and 100 extra random numbers for 10 times, average running time for each data structure and for each size is printed like below

```
Average execution time in nanoseconds for Binary Search Tree for 100 random numbers insertion after 10000 insertion: 21376
```

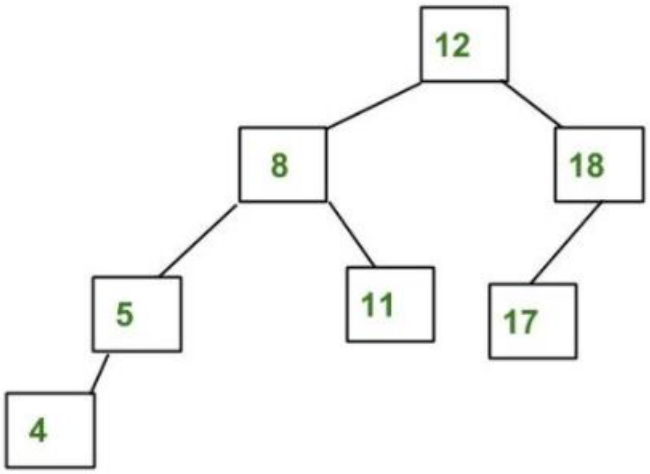
5. RUNNING AND RESULTS

```
PART 1

Skip List
10 is inserted to Skip List
20 is inserted to Skip List
30 is inserted to Skip List
10 is deleted from Skip List
Skip list iterator next(): 30
Skip list iterator next(): 20

AVL Tree
1 is inserted to AVL Tree
2 is inserted to AVL Tree
3 is inserted to AVL Tree
4 is inserted to AVL Tree
5 is inserted to AVL Tree
6 is inserted to AVL Tree
7 is inserted to AVL Tree
AVL Tree iterator next(): 1
AVL Tree iterator next(): 2
AVL Tree iterator next(): 3
AVL Tree iterator next(): 4
AVL Tree iterator next(): 5
AVL Tree iterator next(): 6
AVL Tree iterator next(): 7
In AVL Tree, smaller number from 4(headSet): [1, 2, 3]
In AVL Tree, bigger number from 4(tailSet): [5, 6, 7]
```

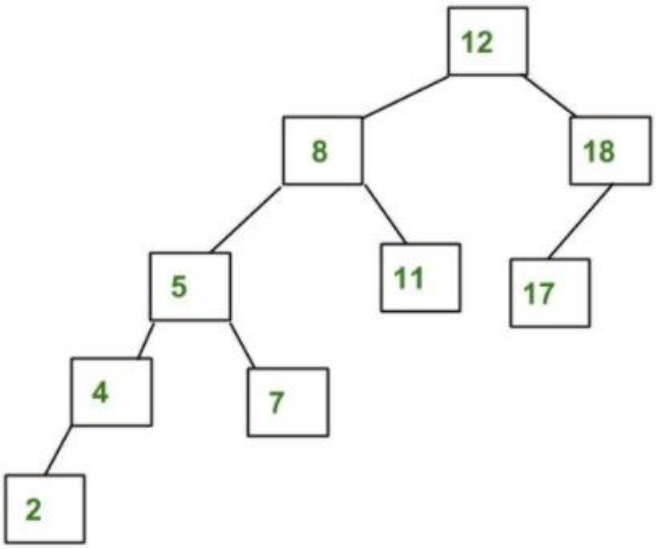
```
PART 2

New Binary Search Tree is created
12 is inserted to Binary Search Tree
8 is inserted to Binary Search Tree
18 is inserted to Binary Search Tree
5 is inserted to Binary Search Tree
11 is inserted to Binary Search Tree
17 is inserted to Binary Search Tree
4 is inserted to Binary Search Tree
The Binary Search Tree is AVL Tree

New Binary Search Tree is created
12 is inserted to Binary Search Tree
8 is inserted to Binary Search Tree
18 is inserted to Binary Search Tree
5 is inserted to Binary Search Tree
11 is inserted to Binary Search Tree
17 is inserted to Binary Search Tree
4 is inserted to Binary Search Tree
7 is inserted to Binary Search Tree
2 is inserted to Binary Search Tree
The Binary Search Tree is not AVL Tree
```

```
PART 3

Average execution time in nanoseconds for Binary Search Tree for 100 random numbers insertion after 10000 insertion: 21376
Average execution time in nanoseconds for Red Black Tree Tree for 100 random numbers insertion after 10000 insertion: 27480
Average execution time in nanoseconds for B Tree for 100 random numbers insertion after 10000 insertion: 3950592
Average execution time in nanoseconds for Skip List for 100 random numbers insertion after 10000 insertion: 27022
Average execution time in nanoseconds for Binary Search Tree for 100 random numbers insertion after 20000 insertion: 19187
Average execution time in nanoseconds for Red Black Tree Tree for 100 random numbers insertion after 20000 insertion: 26500
Average execution time in nanoseconds for B Tree for 100 random numbers insertion after 20000 insertion: 7942643
Average execution time in nanoseconds for Skip List for 100 random numbers insertion after 20000 insertion: 25296
Average execution time in nanoseconds for Binary Search Tree for 100 random numbers insertion after 40000 insertion: 20998
Average execution time in nanoseconds for Red Black Tree Tree for 100 random numbers insertion after 40000 insertion: 26860
Average execution time in nanoseconds for B Tree for 100 random numbers insertion after 40000 insertion: 17871753
Average execution time in nanoseconds for Skip List for 100 random numbers insertion after 40000 insertion: 26049
Average execution time in nanoseconds for Binary Search Tree for 100 random numbers insertion after 80000 insertion: 24825
Average execution time in nanoseconds for Red Black Tree Tree for 100 random numbers insertion after 80000 insertion: 32793
Average execution time in nanoseconds for B Tree for 100 random numbers insertion after 80000 insertion: 36654386
Average execution time in nanoseconds for Skip List for 100 random numbers insertion after 80000 insertion: 27469
cse312@ubuntu:~/Desktop/data_hw7$
```

In Part 2's running and results part, first BinarySearchTree is AVL Tree but second BinarySearchTree is not AVL Tree. In order to show the reason of this, I add two figure below.



This Binary Search Tree which is above is first tree of Part 2 and as we can see, all node's LeftSubtree and RightSubtree heights difference's absolute value is equal or less than 1. Therefore, we can say that this BinarySearchTree is AVL Tree.



This Binary Search Tree which is above is second tree of Part 2 and as we can see, some node's LeftSubtree and RightSubtree heights difference's absolute value is bigger than 1. Therefore, we can say that this BinarySearchTree is not AVL Tree.