CSE222 DATA STRUCTURES AND ALGORITHMS

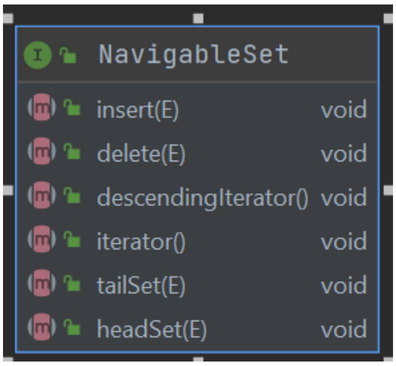
HOMEWORK 7

HATİCE SEVRA GENÇ

1801042611

(This report including problem solution approach, test cases, running command results and detailed system requirements for each part)

PART 1



This is the diagram of NavigableSet interface.

We implement these methods in two ways:

1.1) NavigableSet using SkipList

```
public class NavigableSetWithSkipList<E extends Comparable<E>>  extends SkipList<E> implements NavigableSet<E>{
```

I implement a class named NavigableSetWithSkipList. It extends SkipList and implements NavigableSet classes.

```
SkipList<E> skipList;
```

I create skipList object to use in operations.

a) insert Test

insert() function uses SkipList class's add function

```
1) Insert test  ( After Inserting 5, 12, 0, 7, 98, 79, 45, 13, 23, 1 )
Size of NavigableSet:  10
NavigableSet:  0 --> 1 --> 5 --> 7 --> 12 --> 13 --> 23 --> 45 --> 79 --> 98 -->
```

Navigable set is printed after inserting some numbers.

b) delete

delete() function uses SkipList class's remove function

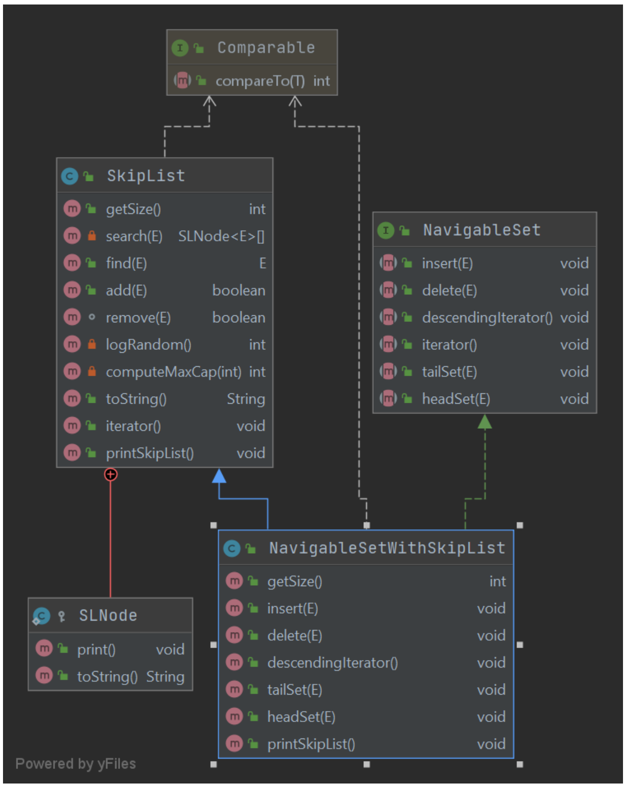
```
2) Delete Test
After Deleting 13
0 --> 1 --> 5 --> 7 --> 12 --> 23 --> 45 --> 79 --> 98 -->
After Deleting 98
0 --> 1 --> 5 --> 7 --> 12 --> 23 --> 45 --> 79 -->
After Deleting 1
0 --> 5 --> 7 --> 12 --> 23 --> 45 --> 79 -->
```

c) descendingIterator

While writing this function, I got help from the skiplist in iterator function. I made some changes to the iterator function so that the elements are visited in descending order

```
3) Descending Iterator Test
79 --> 45 --> 23 --> 12 --> 7 --> 5 --> 0
```

>> Class Diagram of NavigableSetWithSkipList <<



1.2) NavigableSet using AVLTree

```
public class NavigableSetWithAVL<E extends Comparable<E>>  implements NavigableSet<E>{
```

I implement a class named NavigableSetWithAVLTree. It implements NavigableSet class.

```
private AVLTree<E> avl;
```

I create avl object to use while doing operations.

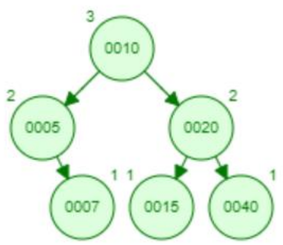
a) insert

insert() function uses AVLTree class's add function

```
NAVIGABLE SET WITH AVL TREE TEST

1) Insert test (After inserting 20, 10, 40, 15, 5, 7)
AVL Tree
10
  5
    null
    7
      null
      null
  20
    15
      null
      null
    40
      null
      null
```

This is the printed tree after insertion operation.



I've added the visualization of the tree to make it look clearer.

(Source: https://www.cs.usfca.edu/~galles/visualization/AVLtree.html)

b) iterator

iterator function calls iterateTree function in the BinaryTree. I impelement both myself. iterateTree function works like inOrder traversal. And prints the nodes using recursive method.

```
2) Iterator test
5 --> 7 --> 10 --> 15 --> 20 --> 40 -->
```

This is the run-time result of iterator() function.

As you can see, our output and the required output are the same.

0005 0007 0010 0015 0020 0040

c) headSet

The headSet function calls the inOrderHeadSet function in BinaryTree. I implemented both functions. In this function, I set the limit of the headset to 10. And I print less than 10 nodes in the tree.

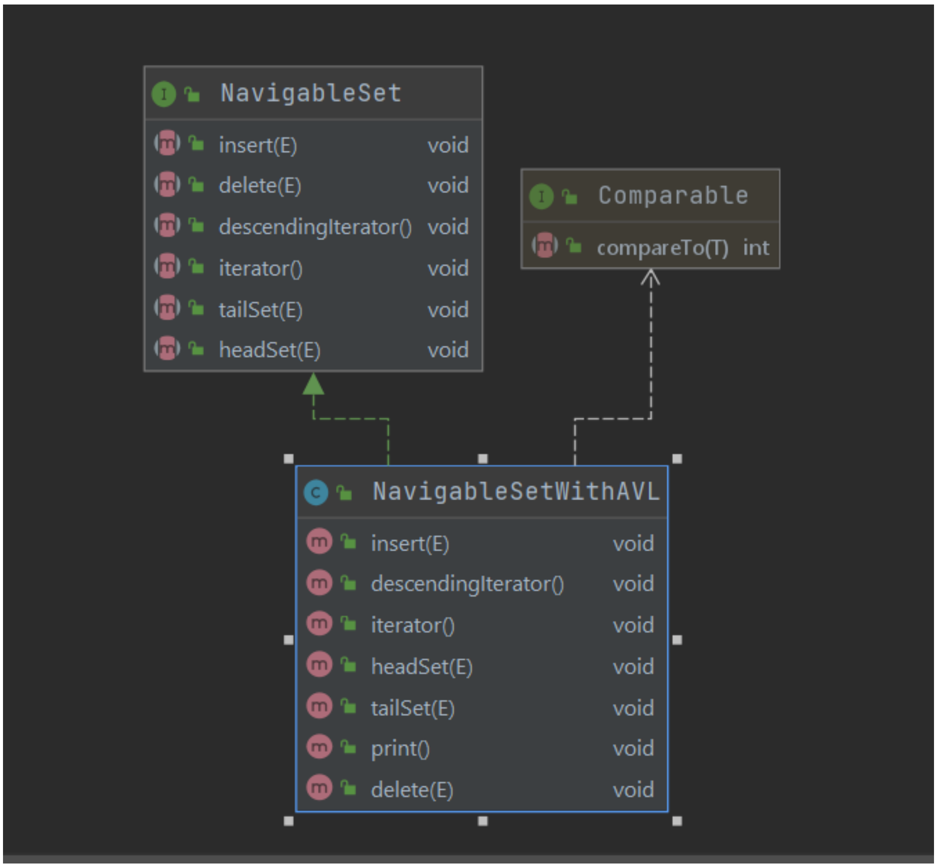
```
4) HeadSet Test (10)
10 or less : 5  7  10
```

d) tailSet

The tailSet function calls the inOrderTailSet function in BinaryTree. I implemented both functions. In this function, I set the limit of the tailSet to 10. And I print less than 10 nodes in the tree.

```
3) TailSet Test (10)
10 or more : 10  15  20  40
```

>> Class Diagram of NavigableSetWithSkipList<<



PART 2

```java
public class BinaryTree < E > implements Serializable {
    /** Class to encapsulate a tree node. */
    protected static class Node < E > implements Serializable {

        public boolean isRed = true;
```

I create a variable name isRed in Binary Tree. If the tree which is sent is Red-Black Tree it's equal 'true'. Otherwise equal 'false'.

- I implement a function named checkType in BinarySearchTree. It returns 1 when getRoot.isRed equals true which means the tree is an AVL Tree. Because the root of the Red-Black Tree never be red, it must be black. So when checkType returns 2 , that means this is a Red-Black Tree.

```java
public int checkType(BinarySearchTree bst){
    if(isRed(bst))
        return 1; // avl
    return 2; //red-black
}
```

```java
AVLTree<Integer> avl1 = new AVLTree<>();
avl1.add(19);
avl1.add(2);
avl1.add(9);
avl1.add(45);
avl1.add(32);
avl1.add(20);
System.out.print("After sending AVL Tree named 'avl1':        ");
typeOfBST(avl1.checkType(avl1));
```

I write driver codes like this.

- Highlighted part are the run-time results of different examples.

```
----------------PART2-----------------

After sending AVL Tree named 'avl1':        Type of the given tree is 'AVL TREE'

After sending Red-Black Tree named 'rbl1':  Type of the given tree is 'RED-BLACK TREE'

After sending Red-Black Tree named 'rbl2':  Type of the given tree is 'RED-BLACK TREE'

After sending AVL Tree named 'avl2':        Type of the given tree is 'AVL TREE'

After sending Red-Black Tree named 'rbl3':  Type of the given tree is 'RED-BLACK TREE'
```

PART 3

A) Construct an Instance of Data Structures by Inserting Different Size of Data

```
ArrayList<Double> retBST =  bstInsert();
ArrayList<Double> retRBT =   redBlackInsert();
ArrayList<Double> retTTF =   twoThreeFourInsert();
ArrayList<Double> retBT =  bTreeInsert();
ArrayList<Double> retSL =    skipListInsert();
```

I implement functions to construct instances for each data structure. Then hold their return values. Because time results are in return values.

For all structures, I created examples of 10,000, 20,000, 40,000 and 80,000 elements, respectively. I used the generateRandom function while doing this. Since this function stores the numbers it generates in the set data structure, there was no duplication. I repeat this process 10 times for each structure.

```
public Set<Integer> generateRandom(int size, int bound){
    Random randNum = new Random();
    Set<Integer> set = new LinkedHashSet<Integer>();
    while (set.size() < size)
        set.add(randNum.nextInt(bound)+1);
    return set;
}
```

B) Inserting 100 Extra Numbers to the Structures

- After creating instances, I add 100 extra element to each instance. While I doing this, measured the running-time.
- Then I calculated the averages for each size and each structure separately.

- For example, after adding 100 elements to a 10,000 element array 10 times, I took the average of these 10 running times.

```
BINARY SEARCH TREE
Average of adding 100 elements to a 10.000 element array:  0,0265
Average of adding 100 elements to a 20.000 element array:  0,0802
Average of adding 100 elements to a 40.000 element array:  0,0448
Average of adding 100 elements to a 80.000 element array:  0,0489
```

```
-----------------------------------
RED BLACK TREE
Average of adding 100 elements to a 10.000 element array:  0,0256
Average of adding 100 elements to a 20.000 element array:  0,0481
Average of adding 100 elements to a 40.000 element array:  0,1104
Average of adding 100 elements to a 80.000 element array:  0,0497
```

```
2-3-4 TREE
Average of adding 100 elements to a 10.000 element array:  0,0387
Average of adding 100 elements to a 20.000 element array:  0,0508
Average of adding 100 elements to a 40.000 element array:  0,0721
Average of adding 100 elements to a 80.000 element array:  0,0701
```

```
-----------------------------------
B-TREE
Average of adding 100 elements to a 10.000 element array:  0,0579
Average of adding 100 elements to a 20.000 element array:  0,0521
Average of adding 100 elements to a 40.000 element array:  0,0688
Average of adding 100 elements to a 80.000 element array:  0,0966
```

```
SKIP LIST TREE
Average of adding 100 elements to a 10.000 element array:  0,0252
Average of adding 100 elements to a 20.000 element array:  0,0347
Average of adding 100 elements to a 40.000 element array:  0,0422
Average of adding 100 elements to a 80.000 element array:  0,0511
```

C) Comparing Running Times

I compare instances of each data structure of the same size.
I evaluated the results and sorted the times from the smallest to the largest, that is, from the most efficient data structure to inefficient. (highlighted part)

```
Average time of data structures (10,000 + 100 size)
BSTree:      0,0286
Red-Black : 0,0351
2-3-4 Tree: 0,0457
B-Tree:      0,0506
SkipList :  0,0217
******** Compare running times:  SkipList <  Binary Search Tree <  Red-Black Tree <  2-3-4 Tree <  B-Tree <
```

```
Average time of data structures (20,000 + 100 size)
BSTree:      0,0362
Red-Black : 0,0406
2-3-4 Tree: 0,0456
B-Tree:      0,0441
SkipList :  0,0274
******** Compare running times:  SkipList <  Binary Search Tree <  Red-Black Tree <  B-Tree <  2-3-4 Tree <
```

```
Average time of data structures (40,000 + 100 size)
BSTree:      0,0360
Red-Black : 0,0551
2-3-4 Tree: 0,0802
B-Tree:      0,0541
SkipList :  0,0353
******** Compare running times:  SkipList <  Binary Search Tree <  B-Tree <  Red-Black Tree <  2-3-4 Tree <
```

```
Average time of data structures (80,000 + 100 size)
BSTree:      0,0465
Red-Black : 0,0519
2-3-4 Tree: 0,0625
B-Tree:      0,0688
SkipList :  0,0541
******** Compare running times:  Binary Search Tree <  Red-Black Tree <  SkipList <  2-3-4 Tree <  B-Tree <
```

D) Increasing Rates

I calculate two types of increasing rate.

- First one is the rate of data structures according to their size

  For example, the percentage change in the average time I get when adding 100 elements to a SkipList of 10,000 elements and the average time I get when adding 100 elements to a SkipList of 20,000 elements

```
----------------------------------
BINARY SEARCH TREE
Average of adding 100 elements to a 10.000 element array:  0,0256
Average of adding 100 elements to a 20.000 element array:  0,0532
Average of adding 100 elements to a 40.000 element array:  0,0363
Average of adding 100 elements to a 80.000 element array:  0,0487

Increasing rates
10.000 vs 20.000: %107
10.000 vs 40.000: %42
10.000 vs 80.000: %90
```

```
RED BLACK TREE
Average of adding 100 elements to a 10.000 element array:  0,0344
Average of adding 100 elements to a 20.000 element array:  0,0565
Average of adding 100 elements to a 40.000 element array:  0,0633
Average of adding 100 elements to a 80.000 element array:  0,0526

Increasing rates
10.000 vs 20.000: %64
10.000 vs 40.000: %84
10.000 vs 80.000: %53

----------------------------------
```

```
----------------------------------
2-3-4 TREE
Average of adding 100 elements to a 10.000 element array:  0,0365
Average of adding 100 elements to a 20.000 element array:  0,0431
Average of adding 100 elements to a 40.000 element array:  0,0537
Average of adding 100 elements to a 80.000 element array:  0,0873

Increasing rates
10.000 vs 20.000: %18
10.000 vs 40.000: %47
10.000 vs 80.000: %139
----------------------------------
```

```
B-TREE
Average of adding 100 elements to a 10.000 element array:  0,0557
Average of adding 100 elements to a 20.000 element array:  0,0463
Average of adding 100 elements to a 40.000 element array:  0,0617
Average of adding 100 elements to a 80.000 element array:  0,0966

Increasing rates (when 100 elements inserted)
10.000 vs 20.000: %-16
10.000 vs 40.000: %10
10.000 vs 80.000: %73
```

```
----------------------------------
SKIP LIST TREE
Average of adding 100 elements to a 10.000 element array:  0,0229
Average of adding 100 elements to a 20.000 element array:  0,0291
Average of adding 100 elements to a 40.000 element array:  0,0356
Average of adding 100 elements to a 80.000 element array:  0,0503

Increasing rates
10.000 vs 20.000: %27
10.000 vs 40.000: %55
10.000 vs 80.000: %119
```

- Second rate type is between the performance of data of the same size by comparing data structures in pairs

  For example the percentage rate of the average time I add 100 elements to a 10,000-element SkipList and the average time I add 100 elements to a 10,000-element Red-Black Tree

```
INCREASING RATES


Binary Search Tree vs Red Black Tree
10.000 -->  %-34
20.000 -->  %-6
40.000 -->  %-74
80.000 -->  %-7

Binary Search Tree vs 2 3 4 Tree
10.000 -->  %-42
20.000 -->  %18
40.000 -->  %-48
80.000 -->  %-79
```

```
Binary Search Tree vs BTree
10.000 -->  %-117
20.000 -->  %12
40.000 -->  %-70
80.000 -->  %-98

Binary Search Tree vs SkipList
10.000 -->  %10
20.000 -->  %45
40.000 -->  %2
80.000 -->  %-3

Red Black Tree vs 2 3 4 Tree
10.000 -->  %-6
20.000 -->  %23
40.000 -->  %15
80.000 -->  %-65
```

```
Red Black Tree vs BTree
10.000 -->  %-62
20.000 -->  %18
40.000 -->  %2
80.000 -->  %-83

Red Black Tree vs SkipList
10.000 -->  %33
20.000 -->  %48
40.000 -->  %43
80.000 -->  %4

2-3-4 Tree vs BTree
10.000 -->  %-52
20.000 -->  %-7
40.000 -->  %-14
80.000 -->  %-10
```

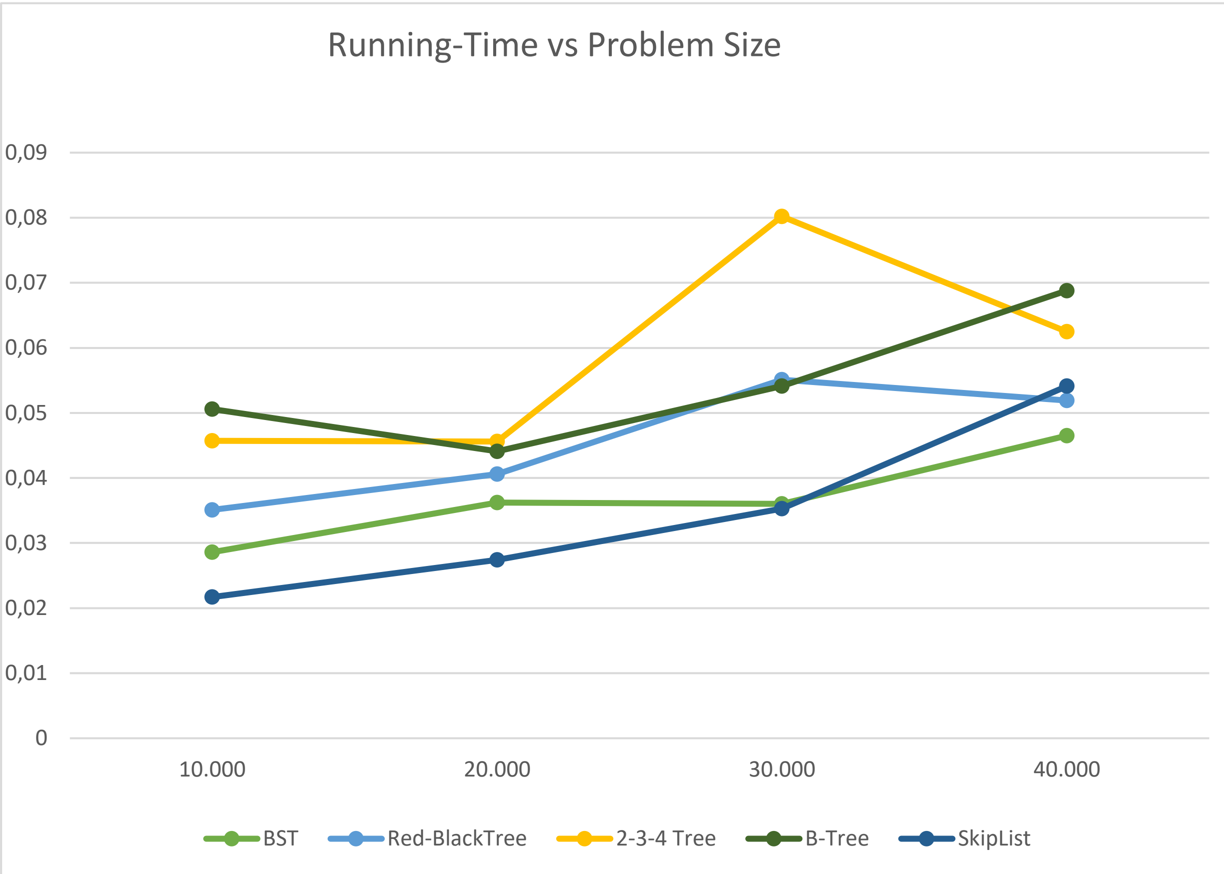
```
2-3-4 Tree vs SkipList
10.000 -->  %37
20.000 -->  %32
40.000 -->  %33
80.000 -->  %42

B-Tree vs SkipList
10.000 -->  %58
20.000 -->  %37
40.000 -->  %42
80.000 -->  %47
```

E) Running Time vs Problem Size Graph

- Representation of data with bar graph

- Representation of the same data with a line chart



>> GENERAL CLASS DIAGRAM <<