GTU Department of Computer Engineering

CSE 222/505 - Spring 2021

Homework 7 Report

FATİH DOĞAÇ

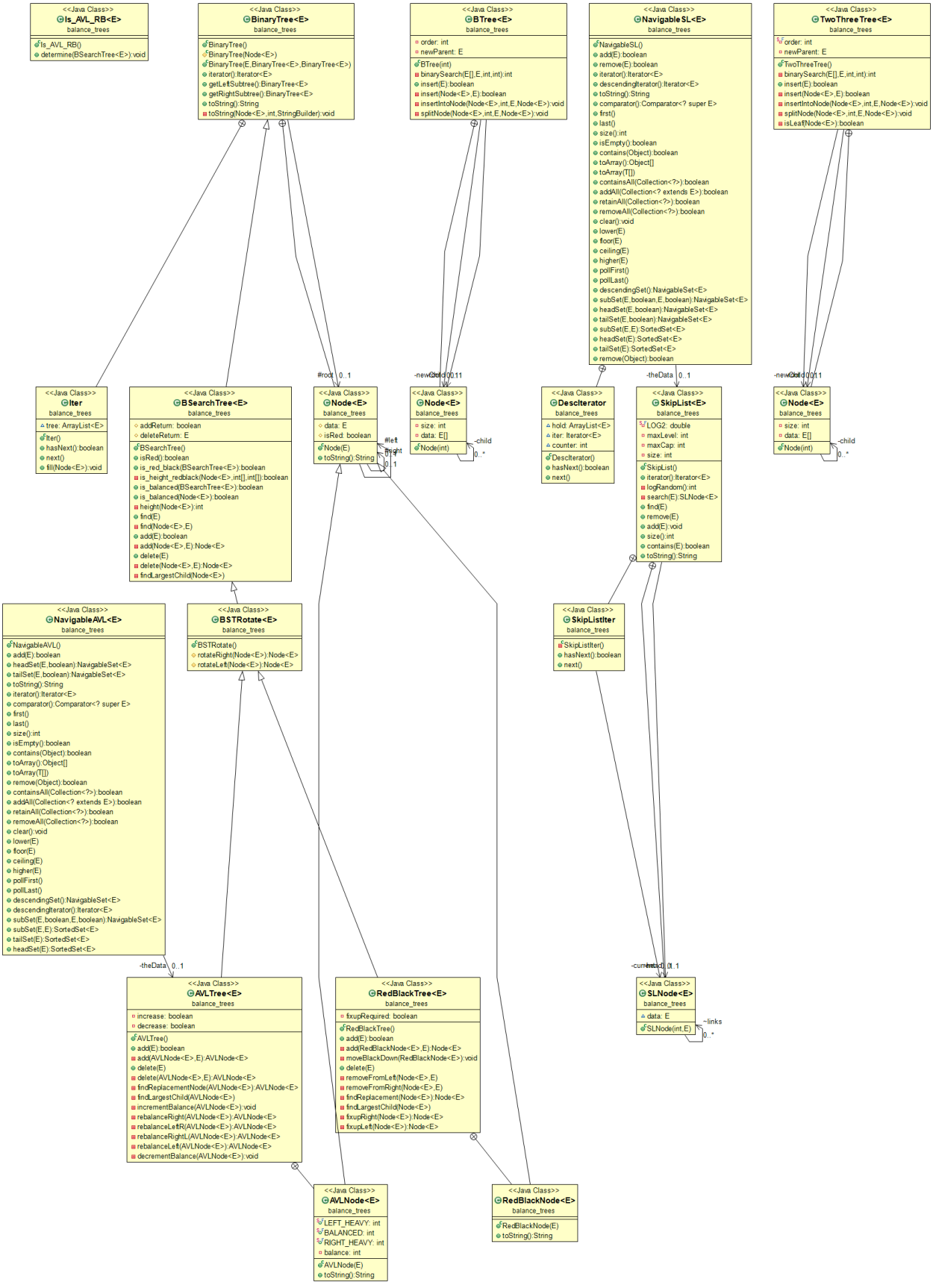
1901042654

System Requirements:

User of the program can:

- Use a Binary Search Tree structure.
- Use a Red-Black Tree structure.
- Use a B-Tree structure.
- Use a 2-3 Tree structure.
- Use a Skip List structure.
- Use a Navigable Set structure implemented by AVL Tree.
- Use a Navigable Set structure implemented by Skip List.

Class Diagram:


<<Java Class>>
Is_AVL_RB<E>
balance_trees
Is_AVL_RB()
determine(BSearchTree<E>):void
<<Java Class>>
BinaryTree<E>
balance_trees
BinaryTree()
BinaryTree(Node<E>)
BinaryTree(E,BinaryTree<E>,BinaryTree<E>)
iterator():Iterator<E>
getLeftSubtree():BinaryTree<E>
getRightSubtree():BinaryTree<E>
toString():String
toString(Node<E>,int,StringBuilder):void
<<Java Class>>
BTree<E>
balance_trees
order: int
newParent: E
BTree(int)
binarySearch(E[],E,int,int):int
insert(E):boolean
insert(Node<E>,E):boolean
insertIntoNode(Node<E>,int,E,Node<E>):void
splitNode(Node<E>,int,E,Node<E>):void
<<Java Class>>
NavigableSL<E>
balance_trees
NavigableSL()
add(E):boolean
remove(E):boolean
iterator():Iterator<E>
descendingIterator():Iterator<E>
toString():String
comparator():Comparator<? super E>
first()
last()
size():int
isEmpty():boolean
contains(Object):boolean
toArray():Object[]
toArray(T[])
containsAll(Collection<?>):boolean
addAll(Collection<? extends E>):boolean
retainAll(Collection<?>):boolean
removeAll(Collection<?>):boolean
clear():void
lower(E)
floor(E)
ceiling(E)
higher(E)
pollFirst()
pollLast()
descendingSet():NavigableSet<E>
subSet(E,boolean,E,boolean):NavigableSet<E>
headSet(E,boolean):NavigableSet<E>
tailSet(E,boolean):NavigableSet<E>
subSet(E,E):SortedSet<E>
headSet(E):SortedSet<E>
tailSet(E):SortedSet<E>
remove(Object):boolean
<<Java Class>>
TwoThreeTree<E>
balance_trees
order: int
newParent: E
TwoThreeTree()
binarySearch(E[],E,int,int):int
insert(E):boolean
insert(Node<E>,E):boolean
insertIntoNode(Node<E>,int,E,Node<E>):void
splitNode(Node<E>,int,E,Node<E>):void
isLeaf(Node<E>):boolean
<<Java Class>>
Iter
balance_trees
tree: ArrayList<E>
Iter()
hasNext():boolean
next()
fill(Node<E>):void
<<Java Class>>
BSearchTree<E>
balance_trees
addReturn: boolean
deleteReturn: E
BSearchTree()
isRed():boolean
is_red_black(BSearchTree<E>):boolean
is_height_redblack(Node<E>,int[],int[]):boolean
is_balanced(BSearchTree<E>):boolean
is_balanced(Node<E>):boolean
height(Node<E>):int
find(E)
find(Node<E>,E)
add(E):boolean
add(Node<E>,E):Node<E>
delete(E)
delete(Node<E>,E):Node<E>
findLargestChild(Node<E>)
#root
0..1
<<Java Class>>
Node<E>
balance_trees
data: E
isRed: boolean
Node(E)
toString():String
#left
#right
0..1
0..1
-newChild
0..11
<<Java Class>>
Node<E>
balance_trees
size: int
data: E[]
Node(int)
-child
0..*
<<Java Class>>
DescIterator
balance_trees
hold: ArrayList<E>
iter: Iterator<E>
counter: int
DescIterator()
hasNext():boolean
next()
-theData
0..1
<<Java Class>>
SkipList<E>
balance_trees
LOG2: double
maxLevel: int
maxCap: int
size: int
SkipList()
iterator():Iterator<E>
logRandom():int
search(E):SLNode<E>
find(E)
remove(E)
add(E):void
size():int
contains(E):boolean
toString():String
-newChild
0..11
<<Java Class>>
Node<E>
balance_trees
size: int
data: E[]
Node(int)
-child
0..*
<<Java Class>>
NavigableAVL<E>
balance_trees
NavigableAVL()
add(E):boolean
headSet(E,boolean):NavigableSet<E>
tailSet(E,boolean):NavigableSet<E>
toString():String
iterator():Iterator<E>
comparator():Comparator<? super E>
first()
last()
size():int
isEmpty():boolean
contains(Object):boolean
toArray():Object[]
toArray(T[])
remove(Object):boolean
containsAll(Collection<?>):boolean
addAll(Collection<? extends E>):boolean
retainAll(Collection<?>):boolean
removeAll(Collection<?>):boolean
clear():void
lower(E)
floor(E)
ceiling(E)
higher(E)
pollFirst()
pollLast()
descendingSet():NavigableSet<E>
descendingIterator():Iterator<E>
subSet(E,boolean,E,boolean):NavigableSet<E>
subSet(E,E):SortedSet<E>
tailSet(E):SortedSet<E>
headSet(E):SortedSet<E>
<<Java Class>>
BSTRotate<E>
balance_trees
BSTRotate()
rotateRight(Node<E>):Node<E>
rotateLeft(Node<E>):Node<E>
<<Java Class>>
SkipListIter
balance_trees
SkipListIter()
hasNext():boolean
next()
-theData
0..1
<<Java Class>>
AVLTree<E>
balance_trees
increase: boolean
decrease: boolean
AVLTree()
add(E):boolean
add(AVLNode<E>,E):AVLNode<E>
delete(E)
delete(AVLNode<E>,E):AVLNode<E>
findReplacementNode(AVLNode<E>):AVLNode<E>
findLargestChild(AVLNode<E>)
incrementBalance(AVLNode<E>):void
rebalanceRight(AVLNode<E>):AVLNode<E>
rebalanceLeftR(AVLNode<E>):AVLNode<E>
rebalanceRightL(AVLNode<E>):AVLNode<E>
rebalanceLeft(AVLNode<E>):AVLNode<E>
decrementBalance(AVLNode<E>):void
<<Java Class>>
RedBlackTree<E>
balance_trees
fixupRequired: boolean
RedBlackTree()
add(E):boolean
add(RedBlackNode<E>,E):Node<E>
moveBlackDown(RedBlackNode<E>):void
delete(E)
removeFromLeft(Node<E>,E)
removeFromRight(Node<E>,E)
findReplacement(Node<E>):Node<E>
findLargestChild(Node<E>)
fixupRight(Node<E>):Node<E>
fixupLeft(Node<E>):Node<E>
-current
0..1
<<Java Class>>
SLNode<E>
balance_trees
data: E
SLNode(int,E)
-links
0..*
<<Java Class>>
AVLNode<E>
balance_trees
LEFT_HEAVY: int
BALANCED: int
RIGHT_HEAVY: int
balance: int
AVLNode(E)
toString():String
<<Java Class>>
RedBlackNode<E>
balance_trees
RedBlackNode(E)
toString():String

Problem Solution Approach:

First , I found the wanted codes from the book. Then I implemented the ones those wanted from me.

To code determining a binary search tree if it is a Red-Black tree or AVL tree. I checked their heights. In Red-Black, the longest path from a node, cannot be greater than twice the shortest path and In AVL Tree , height of the left subtree cannot be greater than height of the right subtree + 1. Same goes for opposites.

Test Cases:

There is a Driver.java file to test all the features of the program in tar.gz file.

Running and Results:

Part1:

```
Adding elements to the Navigable Skip List Set:
[3, 4, 5, 10]
Adding a non-existing element: (109)
[3, 4, 5, 10, 109]
Adding an existing element: (5)
[3, 4, 5, 10, 109]
Deleting an existing element: (3)
[4, 5, 10, 109]
Deleting a non-existing element: (111111)
[4, 5, 10, 109]
Traversing the set with a descending iterator:
109 10 5 4
/*****************************************************/

Adding elements to the Navigable AVL Set:
5 , 4 , 3 , 10
Adding a non-existing element: (109)
5 , 4 , 3 , 10 , 109
Adding an existing element: (5)
5 , 4 , 3 , 10 , 109
Traversing the set with an iterator:
5 4 3 10 109
Usage of headSet method with true inclusive:
4 , 3 , 5 , 10
Usage of headSet method with false inclusive:
4 , 3 , 5
Usage of tailSet method with true inclusive:
10 , 109
Usage of tailSet method with false inclusive:
109
```

Part2:

```
 PART2 *********

Properties of tree1: (Which is red-black and AVL)
The tree is Red-Black-Tree.
The tree is an AVL tree.

Properties of tree2: (Which is only Red-Black)
The tree is Red-Black-Tree.

Properties of tree3: (Which is neither of them.)
The tree is neither a Red-Black-Tree nor an AVL tree.
```

Part3:

```
Avarage time of insertion of the 100 elements to the structures over 10K elements
Binary Search Tree: 0.06629 ms
Red Black Tree: 0.05746 ms
Two Three Tree: 0.10501 ms
B-Tree: 0.08117 ms
Skip List: 0.08535 ms

Avarage time of insertion of the 100 elements to the structures over 20K elements
Binary Search Tree: 0.02508 ms
Red Black Tree: 0.0315 ms
Two Three Tree: 0.0518 ms
B-Tree: 0.04606 ms
Skip List: 0.03592 ms

Avarage time of insertion of the 100 elements to the structures over 40K elements
Binary Search Tree: 0.04043 ms
Red Black Tree: 0.03924 ms
Two Three Tree: 0.04653 ms
B-Tree: 0.03963 ms
Skip List: 0.04416 ms

Avarage time of insertion of the 100 elements to the structures over 80K elements
Binary Search Tree: 0.02091 ms
Red Black Tree: 0.02752 ms
Two Three Tree: 0.04895 ms
B-Tree: 0.02868 ms
Skip List: 0.03289 ms
```

Graph for running-time(ms) vs problem size :