GIT Department of Computer Engineering

CSE 222/505 - Spring 202 1

Homework 7 Report

TUBA TOPRAK

161044116

1. **SYSTEM REQUIREMENTS**

User's reqquirements: a compiler to run the java code and then the examples will be run when the code is compiled.

Q1)

PROBLEM SOLUTION APPROACH

I implemented the SkipList Tree and Avl Tree data structures from the book. then I impelement the navigableSet interface to these two classes.

```
 */
public class SkipList<E extends Comparable<E>> implements NavigableSet {
    /**
     * Head of the skip-list
```

```
public class AVLTree < E
    extends Comparable < E >>
    extends BinarySearchTreeWithRotate < E >  implements NavigableSet {
```

then override was applied.

```
  @Override
  public boolean add(Object o) {
      return SkipList.this.add((E) o);
  }

  @Override
  public boolean remove(Object o) {
      return SkipList.this.remove((E) o);
  }

  @Override
```

```java
@Override
public boolean add(Object o) {
    return AVLTree.this.add((E) o);
}
```

Running and Results

For Skiplist Tree

```java
NavigableSet<Integer> ns = new SkipList<>();
ns.add(20);
ns.add(9);
ns.add(100);
ns.add(45);
System.out.println(ns.toString());
System.out.println("delete 9: ");
ns.remove( o: 9);
System.out.println(ns.toString());
/*
```

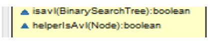
```
Main
"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-javaagent:C:\Progra
Head: 3 --> 9 |1| --> 20 |1| --> 45 |1| --> 100 |1|
delete 9:
Head: 3 --> 20 |1| --> 45 |1| --> 100 |1|

Process finished with exit code 0
```

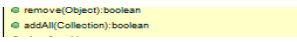
For Avl Tree

```java
AVLTree<Integer> skipList = new AVLTree<>();

NavigableSet<Integer> navl = new AVLTree<>();
navl.add(100);
navl.add(50);
navl.add(150);
System.out.println(navl.toString());
```

```
Main
0: 100
  0: 50
    null
    null
  0: 150
    null
    null


Process finished with exit code 0
```

Q2)





PROBLEM SOLUTION APPROACH

I implemented the binary search tree in the book. and I added method that takes a BinarySearchTree and checks whether the tree is an AVL tree.

```java
 */
boolean isavl(BinarySearchTree tree){
  return helperIsAvl(tree.root);
}
/**
 *
 * @param node of binary search tree
 * @return true whether tree is avl
 */
boolean helperIsAvl(BinaryTree.Node node)
{
  if (node == null) //If tree is empty then return true
    return true;
  int leftHeight = height(node.left);     //find the height of  right sub trees
  int rightHeight = height(node.right);  // find the height of left sub trees
  if (Math.abs(leftHeight - rightHeight) <= 1 && helperIsAvl(node.left) && helperIsAvl(node.right)) return true;
  return false;
}
```

RUNNING AND RESULTS

```java
//2.part
BinarySearchTree tree = new BinarySearchTree();
tree.add(100);
tree.add(50);           //         100       this tree is avl
tree.add(150);         //         /    \
tree.add(25);         //       50       150
tree.add(75);        //       /  \      /  \
tree.add(125);      //      25   75    125  175
tree.add(175);     //           / \         \
tree.add(65);     //           65  85       180
tree.add(85);    //
tree.add(180);
if (tree.isavl(tree)) System.out.println("Tree is AVL Tree");
else
    System.out.println("Tree is not AVL Tree");
BinarySearchTree tree2 = new BinarySearchTree();
tree2.add(100);      //      100  this tree is not avl
tree2.add(50);      //      /
tree2.add(10);     //      50
                  //      /
                 //      10
if (tree2.isavl(tree2)) System.out.println("Tree is AVL Tree");
else
    System.out.println("Tree is not AVL Tree");
```

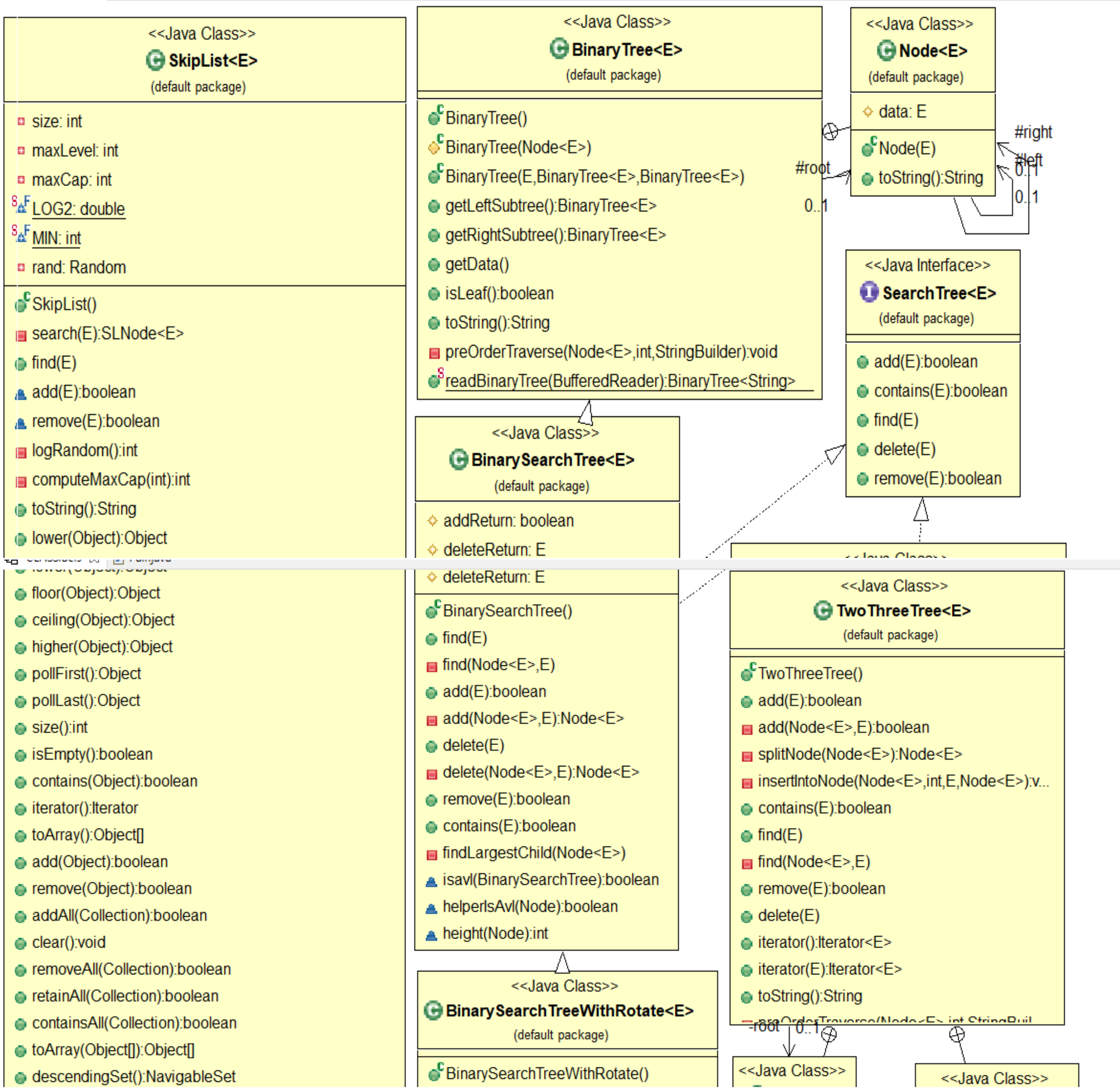
Two examples were made. The tree in the first example is an avl tree.The tree in the second example is not an avl tree.

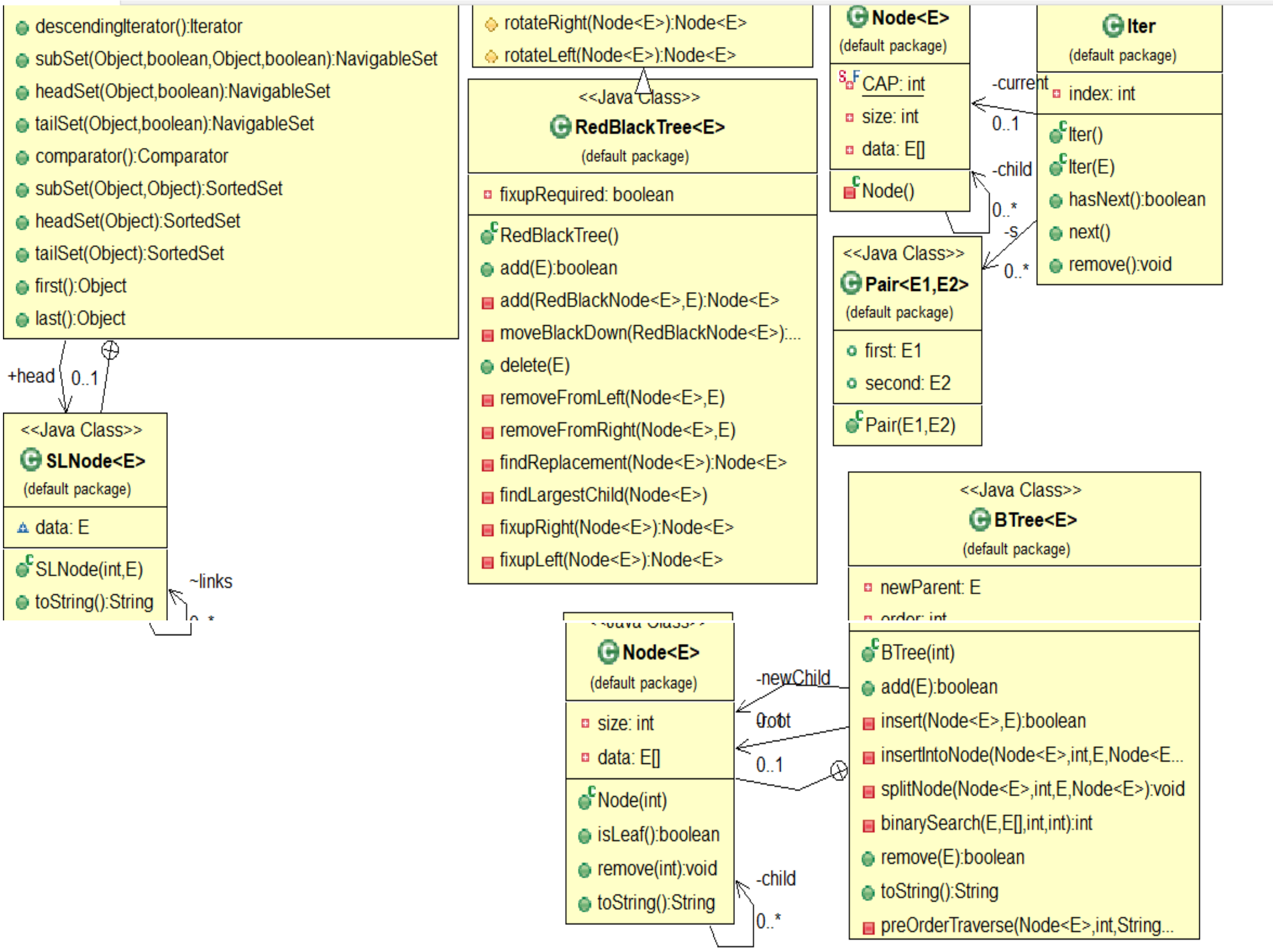
Result

```
"C:\Program Files\Java\jdk-15.0.2\bin\java.exe" "-javaagent:C:\Program Fil
Tree is AVL Tree
Tree is not AVL Tree

Process finished with exit code 0
```

Q3)

CLASS DIAGRAM





PROBLEM SOLUTION APPROACH

I implemented the data structures in the book. Then I performed it with data sizes 10 times.I measured their run time performance. I calculated the average run time for each data structure and problem size. I compared uptimes and rates of increase.

```
//3.part main driver
int size1 = 10000;
int size2 = 20000;
int size3 = 40000;
int size4 = 80000;

BinarySearchTree<Integer>[][] bst = new BinarySearchTree[4][10];
RedBlackTree<Integer>[][] rbt = new RedBlackTree[4][10];
BTree<Integer>[][] bt = new BTree[4][10];
SkipList<Integer>[][] sl = new SkipList[4][10];
TwoThreeTree<Integer>[][] ttt = new TwoThreeTree[4][10];

for (int i = 0; i < 4; i++) {
    for (int j = 0; j < 10; j++) {
        bst[i][j] = new BinarySearchTree<>();
        rbt[i][j] = new RedBlackTree<>();
        bt[i][j] = new BTree<>( order: 4);
        sl[i][j] = new SkipList<>();
        ttt[i][j] = new TwoThreeTree<>();

    }
}
Random rand = new Random();
Integer[][] size10000 = new Integer[10][10000];
```

I am using double dimensional data structures for 4 different sizes and 10 repetitions.

```java
    for (int j = 0; j < 10; j++) {
        for (int k = 0; k < size1; k++) { //10.000
            bst[0][j].add(size10000[j][k]);
            rbt[0][j].add(size10000[j][k]);
            bt[0][j].add(size10000[j][k]);
            sl[0][j].add(size10000[j][k]);
            ttt[0][j].add(size10000[j][k]);
        }
        for (int k = 0; k < size2; k++) {  //20.000
            bst[1][j].add(size20000[j][k]);
            rbt[1][j].add(size20000[j][k]);
            bt[1][j].add(size20000[j][k]);
            sl[1][j].add(size20000[j][k]);
            ttt[1][j].add(size20000[j][k]);
        }
        for (int k = 0; k < size3; k++) {  //40.000
            bst[2][j].add(size40000[j][k]);
            rbt[2][j].add(size40000[j][k]);
            bt[2][j].add(size40000[j][k]);
            sl[2][j].add(size40000[j][k]);
            ttt[2][j].add(size40000[j][k]);
        }
        for (int k = 0; k < size4; k++) {  //80.000
            bst[3][j].add(size80000[j][k]);
```

I apply the insert operation to each data size.

```java
System.out.println("------Red Black Tree -------");
System.out.println("10000");
time = 0;
for (int j = 0; j <10 ; j++) {
    startTime = System.nanoTime();
    for (int i = 0; i < 100; i++)
        rbt[0][j].add(arr100[i]);
    endtime =   System.nanoTime();
    System.out.println((endtime - startTime) + " nanotimes");
    time +=(endtime - startTime);
}
System.out.println("10000 (average) : "+time/10);

System.out.println("20000");
```

and then the time was measured by adding the extra 100 random numbers and the average was found

Total time to insert all data structures 10 times: 7720 ms

Average time(nano)	10000	20000	40000	80000
Binary Search Tree	76200	106300	124560	16663
Red Black Tree	90060	103260	121890	145360
BTree	118920	141650	146820	194250
2 3 Tree	128230	147110	153440	337600
Skiplist Tree	127770	145200	200590	224340

Binary Search Tree

Insert time(nano)	10000	20000	40000	80000
1 times	93200	111900	116900	180800
2 times	80100	102800	163800	219800
3 times	72800	118800	121800	138900
4 times	72000	111400	119400	187700
5 times	71000	106700	117800	203800
6 times	72100	98700	119900	149200
7 times	73100	105000	124300	134200
8 times	70500	100600	118300	142800
9 times	68900	100600	119000	138200
10 times	88300	106500	124400	170900
average	76200	106300	124560	16663

Red Black Tree

	10000	20000	40000	80000
1 times	115600	104900	125200	159700
2 times	90300	113300	125200	140300
3 times	91100	98400	116500	138400
4 times	86100	103200	121100	153000
5 times	87000	102100	117000	135000
6 times	86600	102300	118100	135700
7 times	88400	99200	117800	132600
8 times	85500	102800	122800	139100
9 times	82200	105400	132300	146600
10 times	87800	101000	125100	173200
average	90060	103260	121890	145360

BTree

	10000	20000	40000	80000
1 times	160900	161100	232900	310300
2 times	119200	146100	127900	192100
3 times	112000	139000	132200	163200
4 times	108800	152600	136000	168800
5 times	114800	144800	134600	159600
6 times	114200	130700	131500	186400
7 times	108300	139800	136100	165400
8 times	109300	138700	130100	220800
9 times	114700	131700	128400	173200
10 times	127000	132000	178500	202700
average	118920	141650	146820	194250

SkipList Tree

	10000	20000	40000	80000
1 times	132800	141800	241300	231600
2 times	131700	132800	210800	216300
3 times	168800	132200	212500	204500
4 times	145100	138900	198400	222500
5 times	114300	138000	186800	208100
6 times	120400	138200	185000	211600

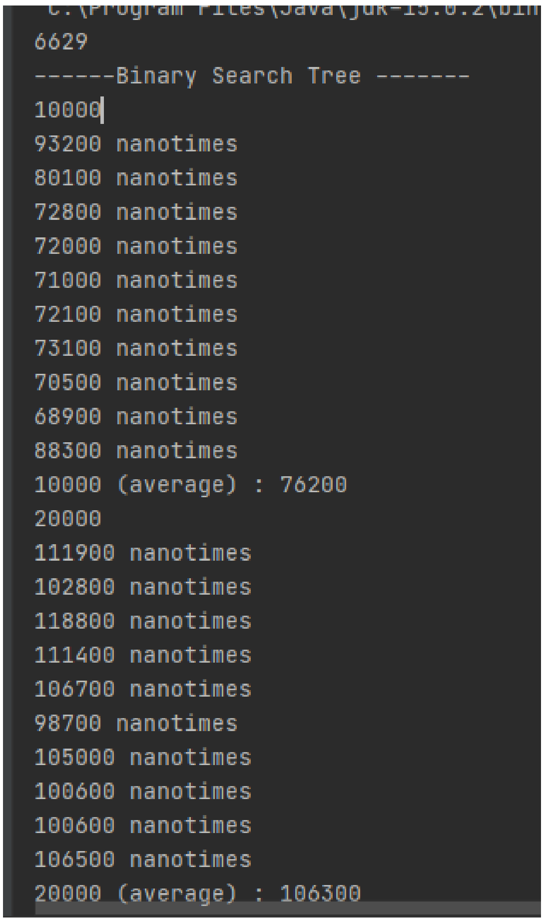
7 times	113700	133900	183000	200000
8 times	115200	138100	179100	241500
9 times	123100	177300	178100	208800
10 times	112600	180800	230900	298500
average	127770	145200	200590	224340

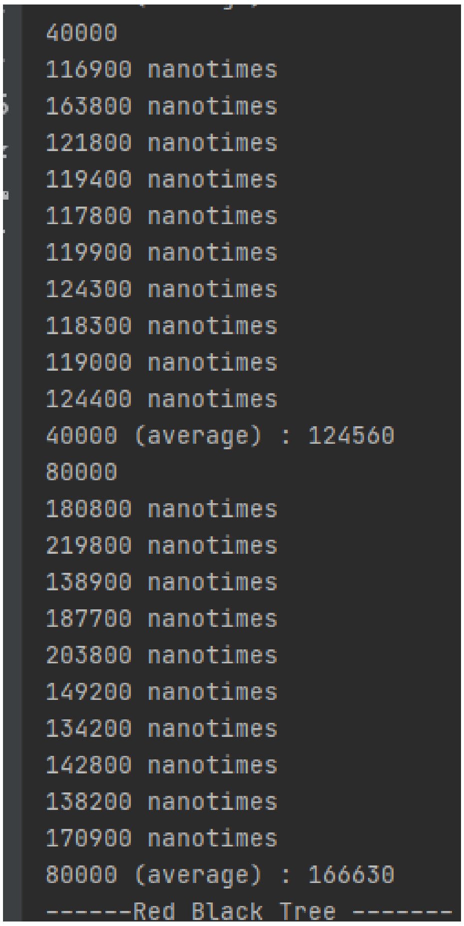
2 3 Tree

	10000	20000	40000	80000
1 times	118800	167100	144600	149600
2 times	114100	167000	188100	170300
3 times	106100	176300	145100	537000
4 times	108000	153200	151500	158900
5 times	110200	128100	169800	777500
6 times	117200	128800	167000	842000
7 times	117100	134500	128600	159400
8 times	111200	130800	143100	166900
9 times	117100	130300	128200	241300
10 times	262500	155000	168400	173100
average	128230	147110	153440	337600

- **Running and Results**

Binary Search Tree





Red Black Tree

```
80000 (average) : 166630
------Red Black Tree -------
10000
115600 nanotimes
90300 nanotimes
91100 nanotimes
86100 nanotimes
87000 nanotimes
86600 nanotimes
88400 nanotimes
85500 nanotimes
82200 nanotimes
87800 nanotimes
10000 (average) : 90060
20000
104900 nanotimes
113300 nanotimes
98400 nanotimes
103200 nanotimes
102100 nanotimes
102300 nanotimes
99200 nanotimes
102800 nanotimes
105400 nanotimes
101000 nanotimes
20000 (average) : 103260
40000
```

```
101000 nanotimes
20000 (average) : 103260
40000
125200 nanotimes
123000 nanotimes
116500 nanotimes
121100 nanotimes
117000 nanotimes
118100 nanotimes
117800 nanotimes
122800 nanotimes
132300 nanotimes
125100 nanotimes
40000 (average) : 121890
80000
159700 nanotimes
140300 nanotimes
138400 nanotimes
153000 nanotimes
135000 nanotimes
135700 nanotimes
132600 nanotimes
139100 nanotimes
146600 nanotimes
173200 nanotimes
80000 (average) : 145360
```

BTree

```
80000 (average) : 145360
------BTree Tree -------
10000
160900 nanotimes
119200 nanotimes
112000 nanotimes
108800 nanotimes
114800 nanotimes
114200 nanotimes
108300 nanotimes
109300 nanotimes
114700 nanotimes
127000 nanotimes
10000 (average) : 118920
20000
161100 nanotimes
146100 nanotimes
139000 nanotimes
152600 nanotimes
144800 nanotimes
130700 nanotimes
139800 nanotimes
138700 nanotimes
131700 nanotimes
132000 nanotimes
20000 (average) : 141650
40000
```

```
20000 (average) : 141650
40000
232900 nanotimes
127900 nanotimes
132200 nanotimes
136000 nanotimes
134600 nanotimes
131500 nanotimes
136100 nanotimes
130100 nanotimes
128400 nanotimes
178500 nanotimes
40000 (average) : 146820
80000
310300 nanotimes
192100 nanotimes
163200 nanotimes
168800 nanotimes
159600 nanotimes
186400 nanotimes
165400 nanotimes
220800 nanotimes
173200 nanotimes
202700 nanotimes
80000 (average) : 194250
------Skiplist Tree -------
```

Skiplist Tree

```
------Skiplist Tree -------
10000
132800 nanotimes
131700 nanotimes
168800 nanotimes
145100 nanotimes
114300 nanotimes
120400 nanotimes
113700 nanotimes
115200 nanotimes
123100 nanotimes
112600 nanotimes
10000 (average) : 127770
20000
141800 nanotimes
132800 nanotimes
132200 nanotimes
138900 nanotimes
138000 nanotimes
138200 nanotimes
133900 nanotimes
138100 nanotimes
177300 nanotimes
180800 nanotimes
20000 (average) : 145200
40000
```

```
40000
241300 nanotimes
210800 nanotimes
212500 nanotimes
198400 nanotimes
186800 nanotimes
185000 nanotimes
183000 nanotimes
179100 nanotimes
178100 nanotimes
230900 nanotimes
40000 (average) : 200590
80000
231600 nanotimes
216300 nanotimes
204500 nanotimes
222500 nanotimes
208100 nanotimes
211600 nanotimes
200000 nanotimes
241500 nanotimes
208800 nanotimes
298500 nanotimes
80000 (average) : 224340
```

2 3 Tree

```
------2 3 Tree -------
10000
118800 nanotimes
114100 nanotimes
106100 nanotimes
108000 nanotimes
110200 nanotimes
117200 nanotimes
117100 nanotimes
111200 nanotimes
117100 nanotimes
262500 nanotimes
10000 (average) : 128230
20000
167100 nanotimes
167000 nanotimes
176300 nanotimes
153200 nanotimes
128100 nanotimes
128800 nanotimes
134500 nanotimes
130800 nanotimes
130300 nanotimes
155000 nanotimes
20000 (average) : 147110
40000
```

```
40000
144600 nanotimes
188100 nanotimes
145100 nanotimes
151500 nanotimes
169800 nanotimes
167000 nanotimes
128600 nanotimes
143100 nanotimes
128200 nanotimes
168400 nanotimes
40000 (average) : 153440
80000
149600 nanotimes
170300 nanotimes
537000 nanotimes
158900 nanotimes
777500 nanotimes
842000 nanotimes
159400 nanotimes
166900 nanotimes
241300 nanotimes
173100 nanotimes
80000 (average) : 337600
```