1.

Balanced Tree: A tree where no leaf is much farther away from the root than any other leaf. Different balancing schemes allow different definitions of "much farther" and different amounts of work to keep them balanced.

Complete Tree & Non-complete Tree: A complete tree is a tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. Otherwise, it is non-complete tree.

How does JVM manages References and Data: All the class-level data will be stored in the method Area, including static variables. All the Objects and their corresponding instance variables and arrays will be stored in the Heap Area. For every thread, a separate runtime stack will be created.

Why Java is PassByValue, how does it work:

in java, arguments are always passed by value regardless of the original variable type. each time a method is invoked, a copy for each argument is created in the stack memory and the copy version is passed to the method.

if the original variable type is primitive, then simply, a copy of the variable is created inside the stack memory and then passed to the method.

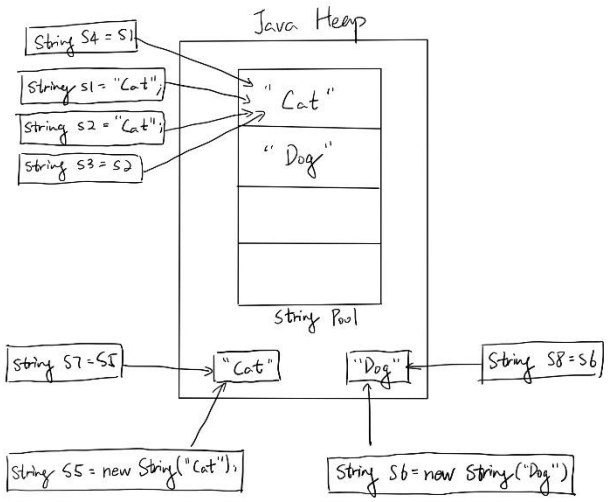
if the original type is not primitive, then a new reference or pointer is created inside the stack memory, which points to the actual object data, and the new reference is then passed to the method, (at this stage, two references are pointing to the same object data).

Why would you consider BTree:

The most time-consuming part of mechanical hard disk is seeking address. In the data structure, if we use the binary tree to store data, the frequent addition and deletion of data will make the binary tree degenerate into a linked list.

The emergence of B-tree overcomes this problem. The addition and deletion of nodes in B-tree is very small for the overall structure, so it is very suitable for data structure as big data storage.

2.



3.

a)

str[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	B	D	C	E	D	D	F	F	C	A	B	E	E	E	C	C	E	F	D	D	A	A	F

use A for 0, B for 1, C for 2, D for 3, E for 4, F for 5

Count frequencies of each letter using key as index:

str[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	B	D	C	E	D	D	F	F	C	A	B	E	E	E	C	C	E	F	D	D	A	A	F

count[]:

0	1	2	3	4	5	6
0	4	2	4	5	5	4

Compute frequency cumulates which specify destinations:

str[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	B	D	C	E	D	D	F	F	C	A	B	E	E	E	C	C	E	F	D	D	A	A	F

count[]:

0	1	2	3	4	5	6
0	4	6	10	15	20	24

Access cumulates using key as index to move items:

str[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	B	D	C	E	D	D	F	F	C	A	B	E	E	E	C	C	E	F	D	D	A	A	F

count[]:

0	1	2	3	4	5	6
4	6	10	15	20	24	24

aux[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	A	A	A	B	B	C	C	C	C	D	D	D	D	D	E	E	E	E	E	F	F	F	F

Copy back into original array:

str[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	A	A	A	B	B	C	C	C	C	D	D	D	D	D	E	E	E	E	E	F	F	F	F

count[]:

0	1	2	3	4	5	6
4	6	10	15	20	24	24

aux[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
A	A	A	A	B	B	C	C	C	C	D	D	D	D	D	E	E	E	E	E	F	F	F	F

b)

The time complexity of key-indexed counting sort is O(n+k).

The time complexity of Insertion sort is O(n^2)

even if k = n, the time complexity of key-indexed counting sort is much better than Insertion sort.

c)

The java code is shown in the zip file.

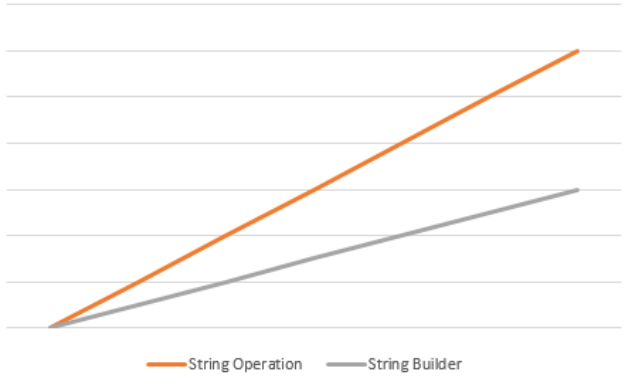
4.

c)

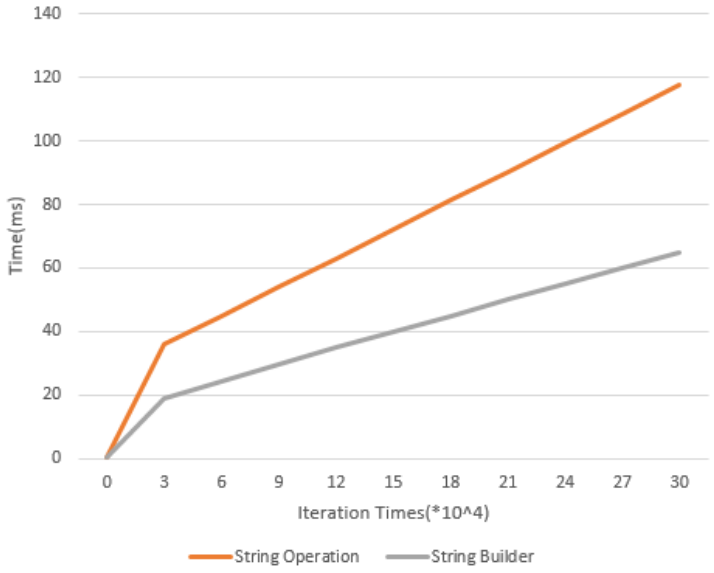
The time complexity is O(n) for String operations and is O(n/2) = O(n) for StringBuilder.

d)

The graph of the time complexity is shown as the following figure:



The graph of the results that is shown as the following figure:



5.

arr[]:

i	arr[i](section)	name
0	2	Anderson
1	3	Brown
2	3	Davis
3	4	Garcia
4	1	Harris
5	3	Jackson
6	4	Johnson
7	3	Jones
8	1	Martin
9	2	Martinez
10	2	Miller
11	1	Moore
12	2	Robinson
13	4	Smith
14	3	Taylor
15	4	Thomas
16	4	Thompson
17	2	White
18	3	Williams
19	4	Wilson

Count frequencies of each grade using key as index:

arr[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	3	4	1	3	4	3	1	2	2	1	2	4	3	4	4	2	3	4

count[]:

0	1	2	3	4
0	3	5	6	6

Compute frequency cumulates which specify destinations:

arr[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	3	4	1	3	4	3	1	2	2	1	2	4	3	4	4	2	3	4

count[]:

0	1	2	3	4
0	3	8	14	20

Access cumulates using key as index to move items:

arr[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
2	3	3	4	1	3	4	3	1	2	2	1	2	4	3	4	4	2	3	4

count[]:

0	1	2	3	4
3	8	14	20	20

aux[]:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4

Copy back into original array:

arr[]:

i	arr[i](section)	name
0	1	Harris
1	1	Martin
2	1	Moore
3	2	Anderson
4	2	Martinez
5	2	Miller
6	2	Robinson
7	2	White
8	3	Brown
9	3	Davis
10	3	Jackson
11	3	Jones
12	3	Taylor
13	3	Williams
14	4	Garcia
15	4	Johnson
16	4	Smith
17	4	Thomas
18	4	Thompson
19	4	Wilson

count[]:

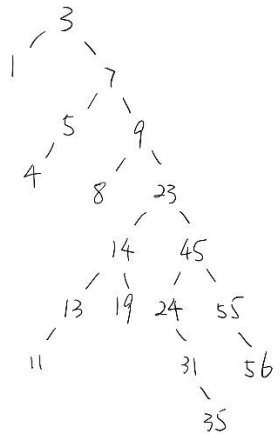
0	1	2	3	4
3	8	14	20	20

aux[]:

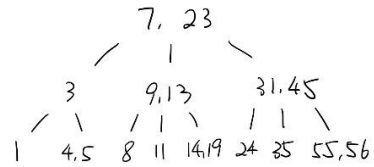
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	2	2	2	2	2	3	3	3	3	3	3	4	4	4	4	4	4

6.

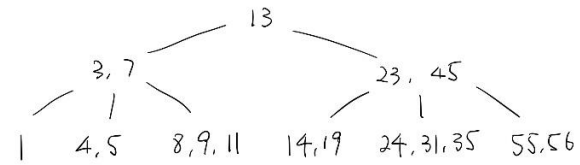
a)



b)



c)



d)

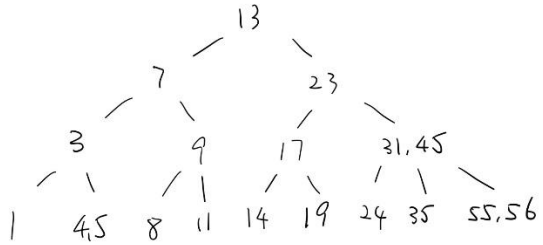
The time complexity of Search, Insert and Delete method for BST are all θ(log(n)) and O(n). BST is easier in coding and have a well performance of Insert and Delete when the tree is balanced.

The time complexity of Search, Insert and Delete method for 2-3 Tree are all θ(log(n)) and O(log(n)). 2-3 Tree is much easier to maintain the balance of the tree and have less layer than BST which means it executes faster while searching.

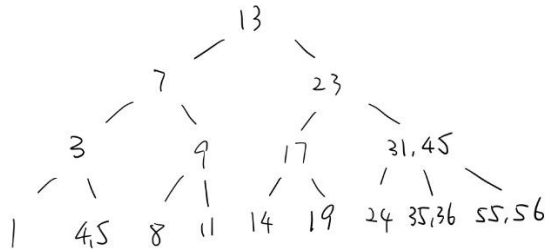
The time complexity of Search, Insert and Delete method for 2-3-4 Tree are all θ(log(n)) and O(log(n)). 2-3-4 Tree performs better of the running time. Although the time complexity is the same, 2-3-4 Tree has the least layers among the three trees which means it could run faster in searching.

e)

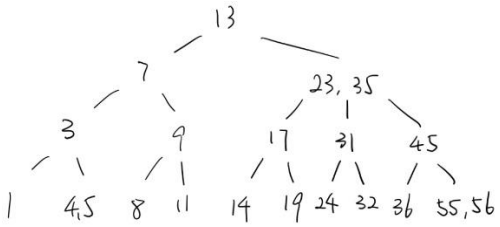
Insert 17:



Insert 36:

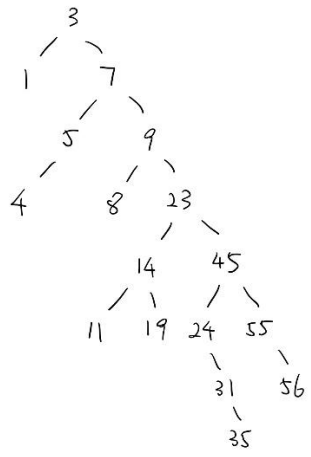


Insert 32:

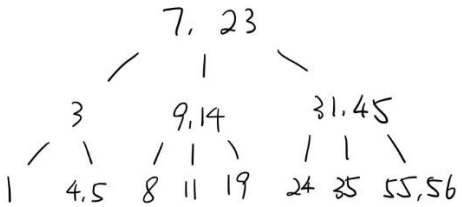


f)

Delete 13 in (a):

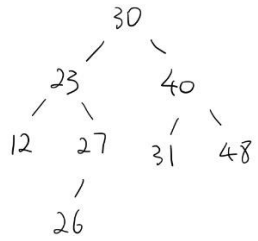


Delete 13 in (b):



g)
The height of (a) is 7, the height of (b) and (c) is 2.

8.
a)



b)
The maximum height of a binary search tree is n - 1. In this case, the input data could be an array in ascending or descending order.

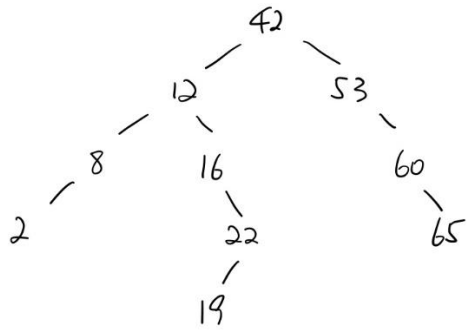
c)
All kinds of operations of the BST (insert, search, delete) has the same time complexity of O(n). In the worst situation, it's the same as that is discussed in question b. There are n layers in the BST.

9.

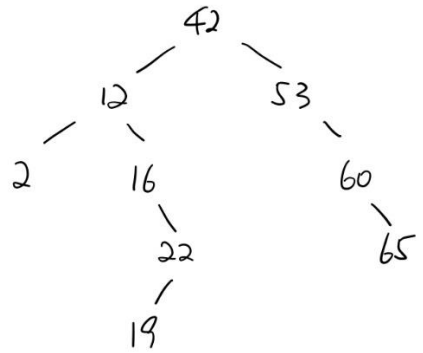
```
Node deleteNode(Node root, int valueToDelete) {
   if root = null
      return node
   if root.value < valueToDelete
      deleteNode(root.right, valueToDelete)
   if root.value > valueToDelete
      deleteNode(root.left, valueToDelete)
   else
      if (isLeafNode(root))
         return null

      if (root.right == null)
         return root.left
      if (root.left == null)
         return root.right

      else
         maxValue = findMaxInLeftSubtree(root)
         root.value = maxValue
         removeDuplicateNode(root)
         return root
}
```

```
Node max(Node x) {
    if(x.right == null) return x;
    else return max(x.right);
}

void DeleteMax() {
    root = DeleteMax(root);
}

Node DeleteMax(Node x) {
    if(x.right == null) return x.left;
    x.right = DeleteMax(x.right);
    x.count = 1 + size(x.right) + size(x.left);
    return x;
}
```

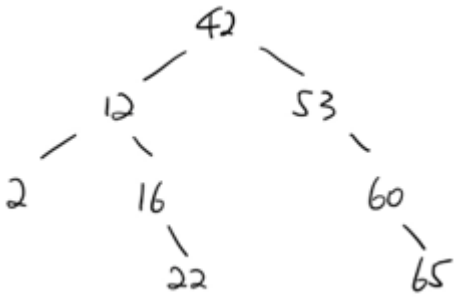
Delete 57:



Delete 8:

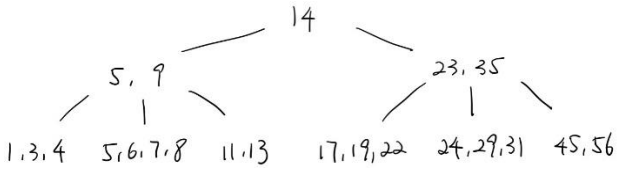


Delete 19:



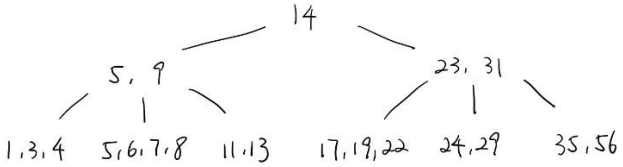
10.

a)

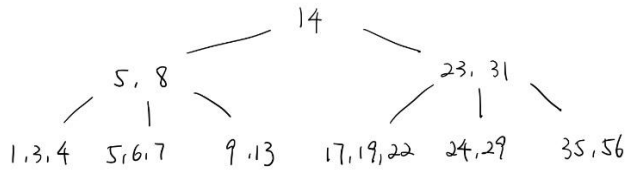


b)

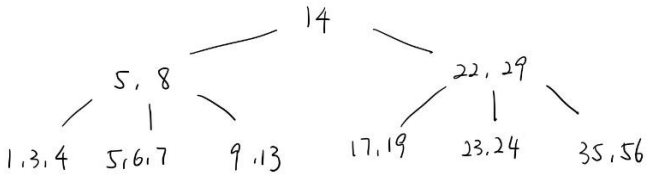
First, delete 45. The node then has only one key which is 56. So 35 is then moved to the node. Then the node which used to contain 23 and 35 has only one key. Then 31 is moved to the node. The result is shown in the following figure:



Then, delete 11. The node then has only one key which is 13. So 9 is then moved to the node. Then the node which used to contain 5 and 9 has only one key. Then 8 is moved to the node. The result is shown in the following figure:



Then, delete 31. The node then has only one key which is 23. So 29 is then moved to the node. Then the node which used to contain 24 and 29 has only one key. Then 23 is moved to the node. Then the node which used to contain 23 and 29 has only one key. Then 22 is moved to the node. The result is shown in the following figure:



d)

Inorder: 1,3,4,5,5,6,7,8,9,11,13,14,17,19,22,23,24,29,31,35,45,56

Preorder: 14,5,9,1,3,4,5,6,7,8,11,13,23,35,17,19,22,24,29,31,45,56

Postorder: 1,3,4,5,6,7,8,11,13,5,9,17,19,22,24,29,31,45,56,23,35,14

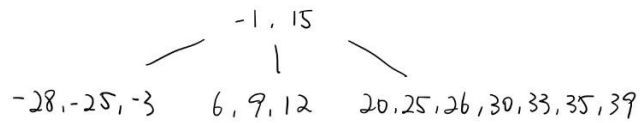
e)

```
class Record {
    private int[] key;
    private int[] val;
    private Node[] node;
```

```
    public Record(int[] key, int[] val, Node[] node) {
        this.key = key;
        this.val = val;
        this.node = node;
    }
    public Record(int[] key){
        this.key = key;
    }
    public int[] getKey() {
        return key;
    }
    public void setVal(int[] val){
        this.val = val;
    }
    public int[] getVal() {
        return val;
    }

    public void setKey(int[] key) {
        this.key = key;
    }
    public Node[] getNode() {
        return node;
    }
    public void setNode(Node[] node) {
        this.node = node;
    }
}
```

11.

a)



b)

The order of this btree is 8

c)

The right child of the root which contains 20,25,26,30,33,35,39 has seven keys which means:

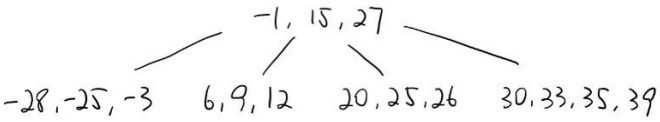
$2t-1>=7$

The left nodes of the root contain -28,-25,-3 and 6,9,12 have three keys which means:

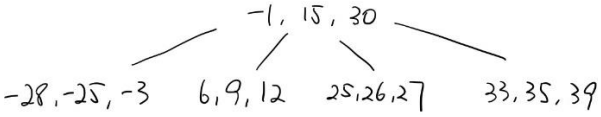
$t-1<=3$

Then we can get minimum degree t=4.

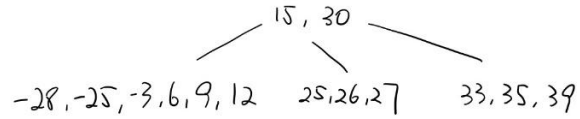
d)
Insert 27:



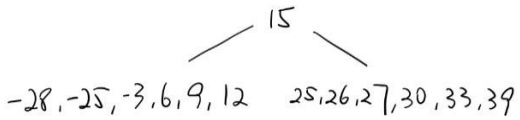
e)
Delete 20:



Delete -1:



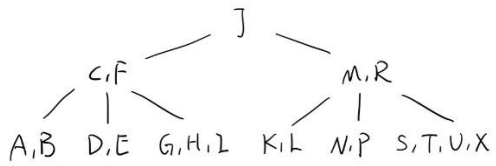
Delete 35:



12.
a)
1024/100 = 10. So 10 records fit onto a block.

b)
1000000/10 = 100000. So 100000 blocks are required to store the entire file.

13.
a)



b)
The minimum degree for this BTree is t=3.

c)
The height of BTree is log(n). In this case, the height is 2.
Time complexity for BTree is O(log(n))

Space complexity for BTree is O(log(n))

14.

A)

Single Left Rotation (LL Rotation):

In LL Rotation every node moves one position to left from the current position.

Single Right Rotation (RR Rotation):

In RR Rotation every node moves one position to right from the current position.

Left Right Rotation (LR Rotation):

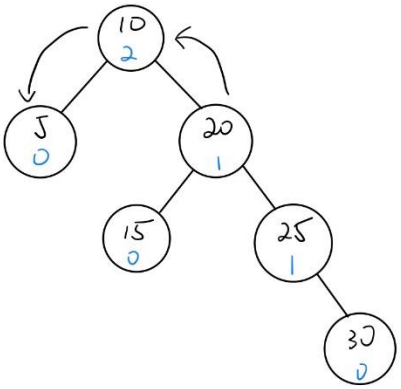
The LR Rotation is combination of single left rotation followed by single right rotation. In LR Rotation, first every node moves one position to left then one position to right from the current position.

Right Left Rotation (RL Rotation):

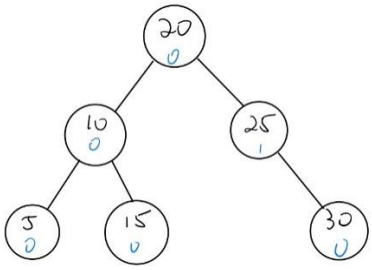
The RL Rotation is combination of single right rotation followed by single left rotation. In RL Rotation, first every node moves one position to right then one position to left from the current position.

In this case:

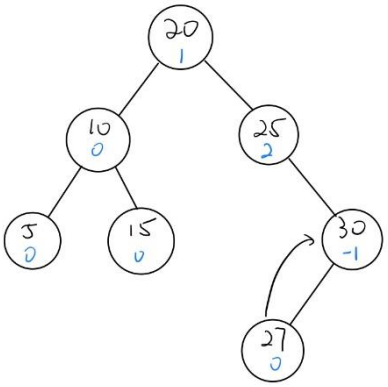
Insert 30:



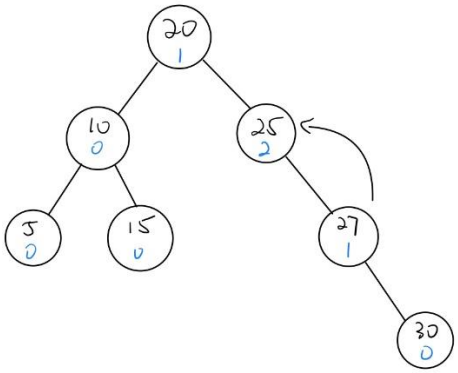
A single left rotation is needed:



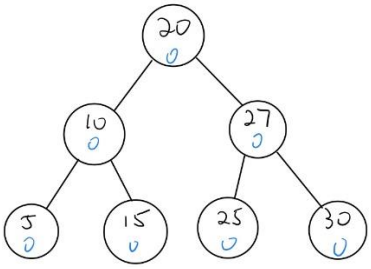
Insert 27:



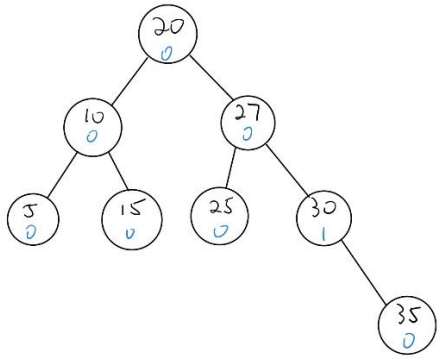
double rotation part 1:



double rotation part 2:



Insert 35:



B)

The following implementation uses the recursive BST insert to insert a new node. In the recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node.

1) Perform the normal BST insertion.

2) The current node must be one of the ancestors of the newly inserted node. Update the height of the current node.

3) Get the balance factor (left subtree height – right subtree height) of the current node.

4) If balance factor is greater than 1, then the current node is unbalanced and we are either in Left Left case or left Right case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.

5) If balance factor is less than -1, then the current node is unbalanced and we are either in Right Right case or Right-Left case. To check whether it is Right Right case or not, compare the newly inserted key with the key in right subtree root.

D)

No need to make change. The record class is designed for binary tree. Since the AVL tree is a binary tree, the record class can fit the AVL tree well.

15.

Yes, it works correctly.

The "printable" method is described as synchronized. In the main method of the test class, the threads, t1 and t2, are both defined basing on the same object obj1. So the same "printable" method of the same object is called. Then the synchronized can work correctly.