Supervision work
Supervision 12

Tudor Avram
Homerton College, tma33@cam.ac.uk

9 Feb 2016

1 Elementary data structures

1.1 Exercise 1

The palindrome testing method :

```
public boolean PalindromeTest(String s) throws EmptyStackException{
          char[] input = s.toCharArray();
          int lng = s.length();
          int start;
          Stack<Character> myStack = new Stack<Character>();
          for (int i = 0; i < lng/2; i++) {
                    // we put the first half in the stack
                    myStack.push(input[i]);
          }
          if (lng % 2 == 0) start = lng/2;
          else start = lng/2+1;
          for (int i = start; i < lng; i++) {
                    char c = myStack.peek();
                    myStack.pop();
                    if ( c != input[i]) {
                              //our string is not a palindrome
                              return false;
                    }
          }
          // after the for loop is done, we can deduce that
          // our string IS a palindrome
          return true;
          }
```

It uses the following auxiliary classes :

(a) the *Stack* class :

```
package uk.ac.cam.tma33.s12;

import java.util.ArrayList;

public class Stack<T> {

private ArrayList<T> mStack;
private int mTop;

public Stack() {
        mStack = new ArrayList<T>();
        mTop = -1;
}

public boolean empty() {
        if (mTop < 0) return true;
        else return false;
}

public void push(T input) {
        mStack.add(++mTop, input);
}

public void pop() throws EmptyStackException{
        if (empty()){
                throw new EmptyStackException();
        }
        else {
                mStack.remove(mTop);
                // the top moves 1 position lower
                mTop--;
        }

}

public T peek() throws EmptyStackException{
        if (!empty()) {
                return mStack.get(mTop);
        }
        else {
                throw new EmptyStackException();
        }
}
}
```

(b) the *EmptyStackException* class :

```
package uk.ac.cam.tma33.s12;

public class EmptyStackException extends Exception{

public EmptyStackException() {
        super();
}

public EmptyStackException(String msg) {
        super(msg);
}

}
```

1.2 Exercise 2

ADVANTAGES : - easy removing of elements (i.e. just changing a reference)
- using just as much memory as needed and not having to resize the structure, as it would have been the case if we used an array.

DISADVANTAGES : - slow inserting (O(n) time) compared to an array-based stack (O(1) time)

1.3 Exercise 3

In order to analyse both strategies, we can take the example of inserting n elements into a stack.

Strategy 1 : resizing the array at every step
In this case, we need to go through the array at every step, in order to resize it. The number of operations can be expressed by the following function :
$f(n) = 1 + 2 + ... + (n-1) = \sum_1^{n-1} k = \frac{n*(n-1)}{2}$
In other words, we can say the the overall complexity of inserting the n numbers would be $O(n^2)$. Therefore, for a single $push()$ operation, the amortised cost would be $\boldsymbol{O(n)}$.

Strategy 2 : doubling the size of the array when needed
Using this strategy, we are going to increase the size of the array only at some of the calls of the $push()$ method and not at all of them, as previously. Function $g(n)$ describes how this works when calling $push()$ n times :
Let $p = [log(n)]$ (i.e. the integer part of the logarithm)
$g(n) = 1 + 2^2 + 2^3 + ... + 2^p = \sum_0^p 2^k = 2^{p+1} - 1 = 2 * n - 1$

In other words, the overall complexity of inserting n numbers to the stack using this strategy is $O(n)$. Therefore, for a single *push*() operation, the amortised cost would be $\boldsymbol{O(1)}$.

1.4 Exercise 4

The *Deque* class, implemented in java :

```java
package uk.ac.cam.tma33.s12;

public class Deque<T> {

private T mDeque[];
private int mFront, mRear;

//Constructor
@SuppressWarnings("unchecked")
public Deque() {
        mDeque = (T[]) new Object[2];
        mFront = 1;
        mRear = 0;
}

// Checkers
public int size() {
        if (!empty()) return (mRear - mFront + 1);
        else return 0;
}

public boolean empty() {
        return (mRear < mFront);
}

// Resize method for the array
@SuppressWarnings("unchecked")
private void resize() {
        int N = mDeque.length;
        T newDeque[] = (T[]) new Object[N*2];
        for (int i = 0; i < N; i++) {
                newDeque[i + N/2] = mDeque[i];
        }
        mDeque = newDeque;
        mFront += N/2;
        mRear += N/2;
}
```

```java
// Setters
public void pushFront(T input) {
        if (mFront - 1 < 0) {
                // we need to resize the array
                resize();
        }
        mDeque[--mFront] = input;
}

public void pushBack(T input) {
        if (mRear + 1 > mDeque.length) {
                // we need to resize the array
                resize();
        }
        mDeque[++mRear] = input;
}

// getters
public T getFirst() throws EmptyDequeException{
        if (empty()) {
                throw new EmptyDequeException();
        }
        else return mDeque[mFront];
}

public T getLast() throws EmptyDequeException{
        if (empty()) {
                throw new EmptyDequeException();
        }
        else return mDeque[mRear];
}

// removers
public void removeFirst() throws EmptyDequeException{
        if (empty()) {
                throw new EmptyDequeException();
        }
        else mFront++;
}

public void removeLast() throws EmptyDequeException{
        if (empty()) {
                throw new EmptyDequeException();
        }
        else mRear--;
}
```

```
}
```

For this implementation, I also declared the *EmptyDequeException* class :

```
package uk.ac.cam.tma33.s12;

public class EmptyDequeException extends Exception{

public EmptyDequeException() {
        super();
}

public EmptyDequeException(String msg) {
        super(msg);
}

}
```

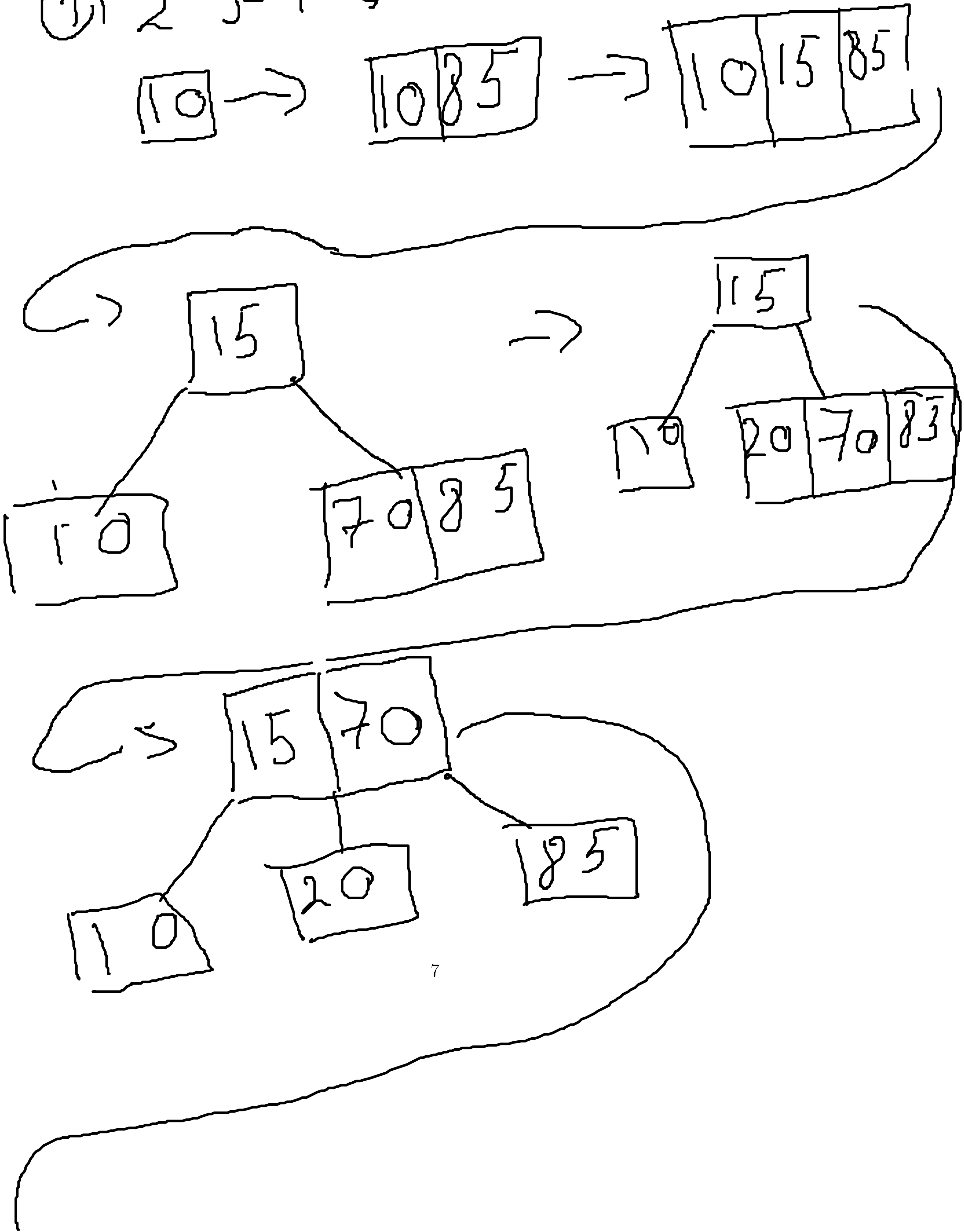
2 BST, Red-Black and 2-3-4 trees

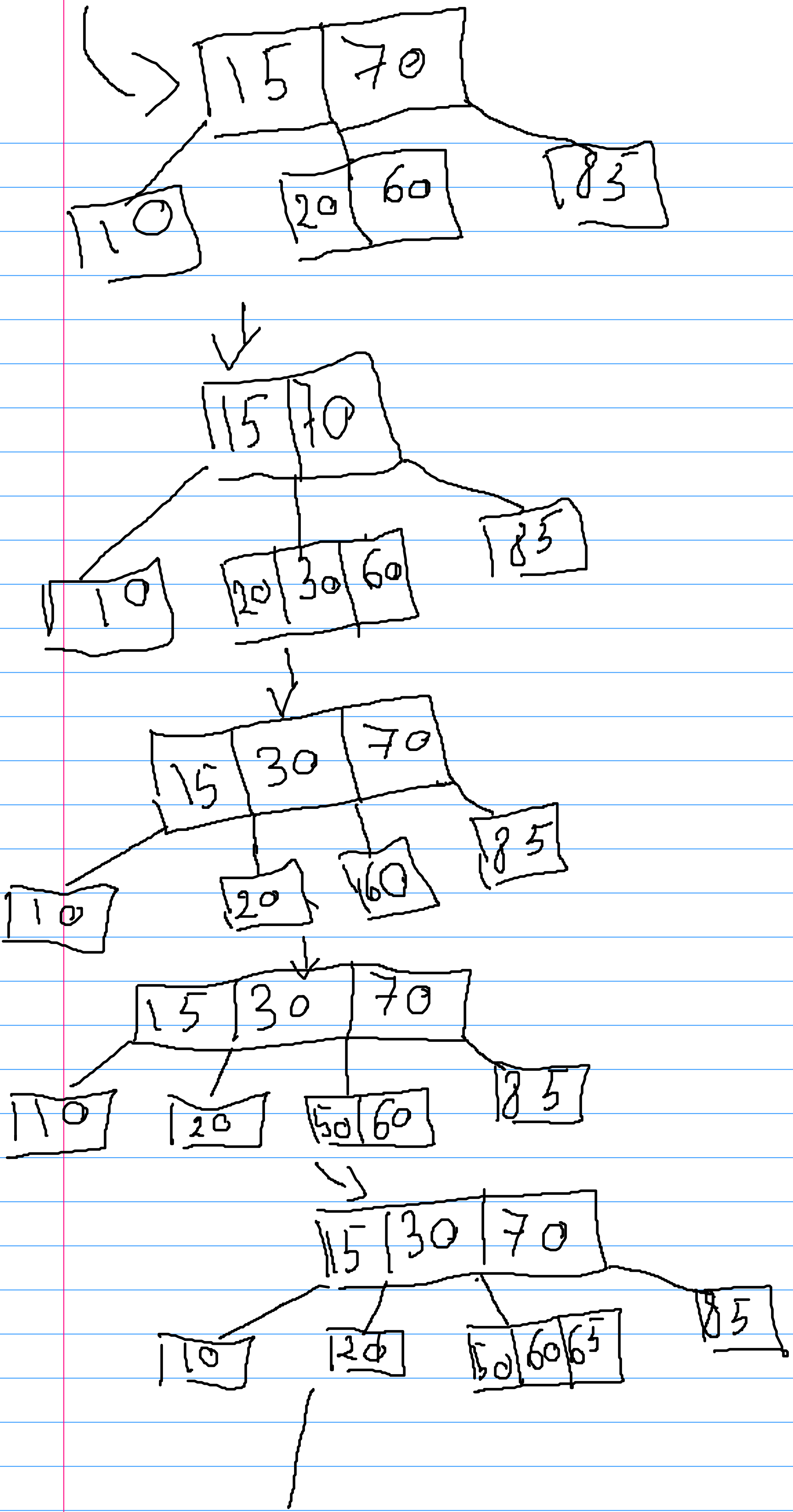
2.1 Exercise 6

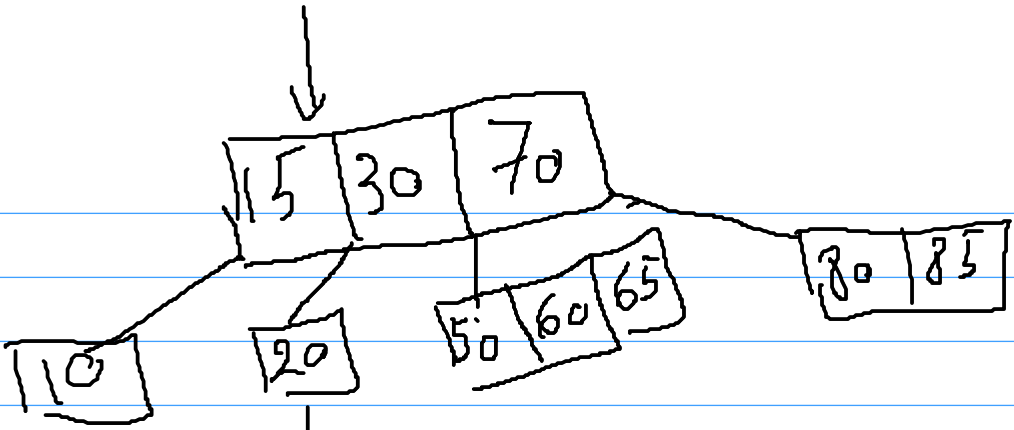
A solution to keep the tree balanced would be to keep a pair of the key and a list of values, instead of a pair that consists of the key and one value. This way, when we find a duplicate key, we insert the value in the list IFF it is not already there. A downside for this implementation would be the additional cost from searching the list.
Another viable approach would be to also keep a count for every $(key, value)$ pair. This way, when we want to insert a value and we get to a node with the same key and value, we increase the count for that specific pair. If the keys are different, or the keys are equal and the values are different, we continue to one of the node's children.

2.2 Exercise 7

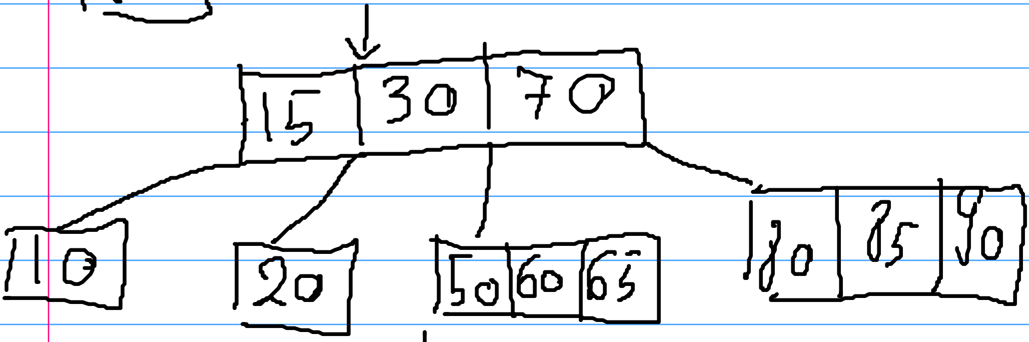
① 2-3-4 tree



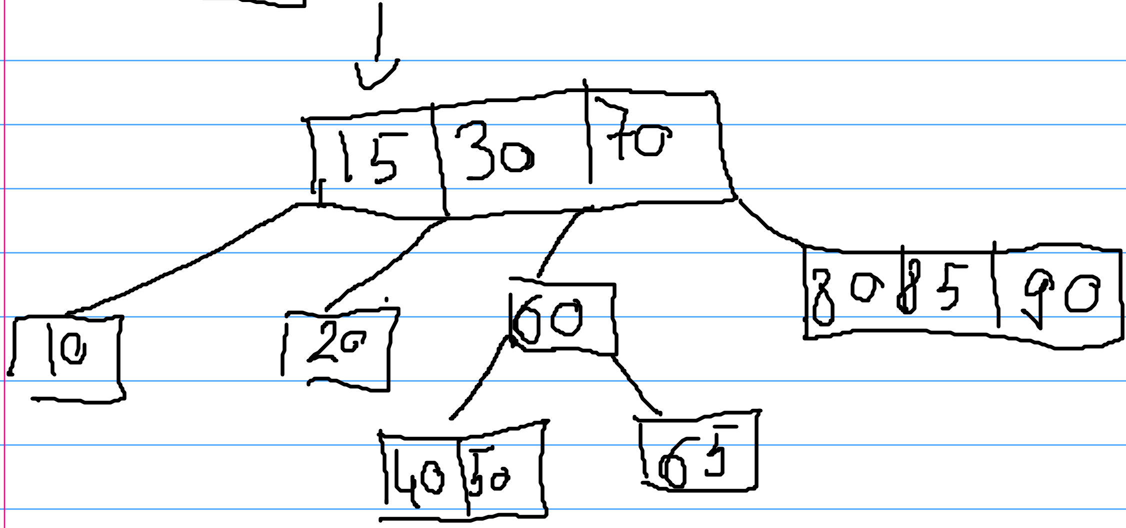

15 | 70
10
20 | 60
85
15 | 70
10
20 | 30 | 60
85
15 | 30 | 70
10
20
60
85
15 | 30 | 70
10
20
50 | 60
85
15 | 30 | 70
10
20
50 | 60 | 65
85



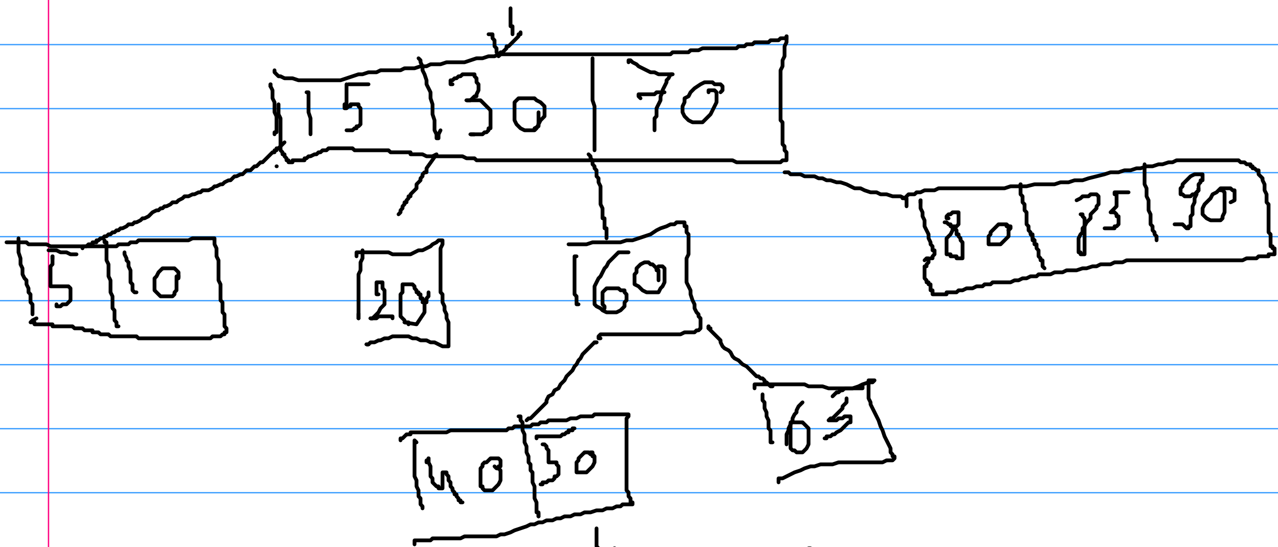
15 | 30 | 70
10
20
50 | 60 | 65
80 | 85



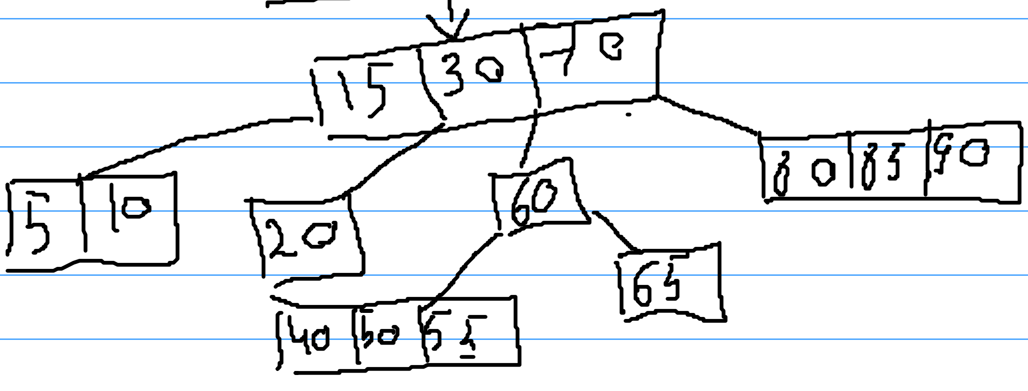
15 | 30 | 70
10
20
50 | 60 | 65
80 | 85 | 90



15 | 30 | 70
10
20
60
40 | 50
65
80 | 85 | 90



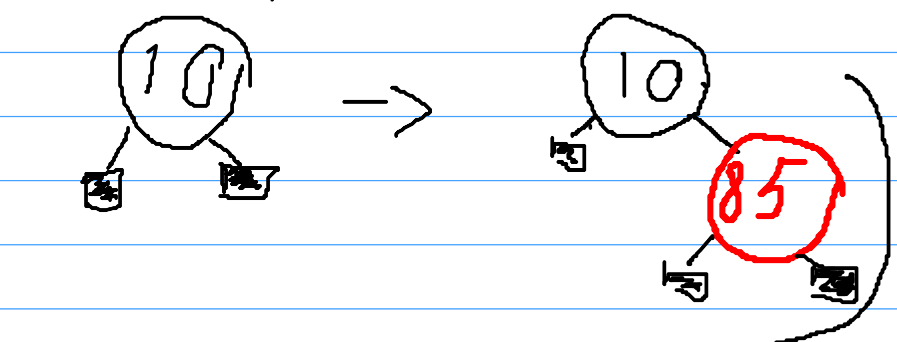
15 | 30 | 70
5 | 10
20
60
40 | 50
65
80 | 85 | 90

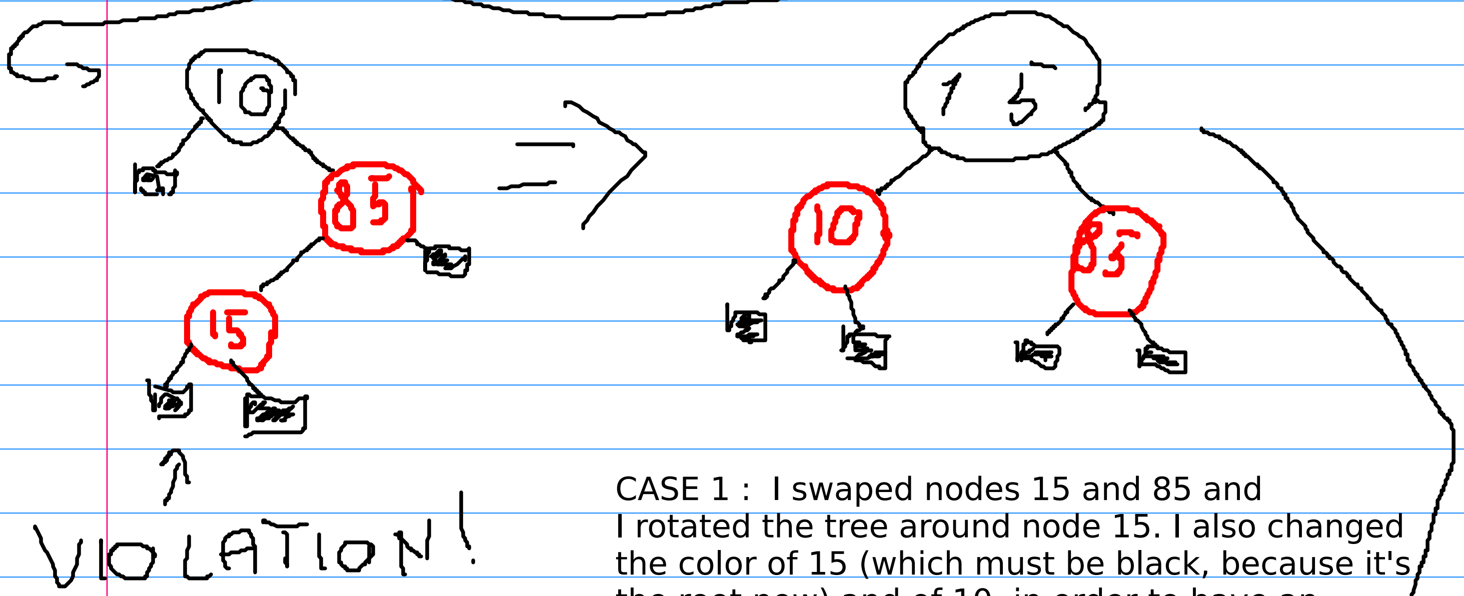


15 | 30 | 70
5 | 10
20
60
40 | 50 | 55
65
80 | 85 | 90

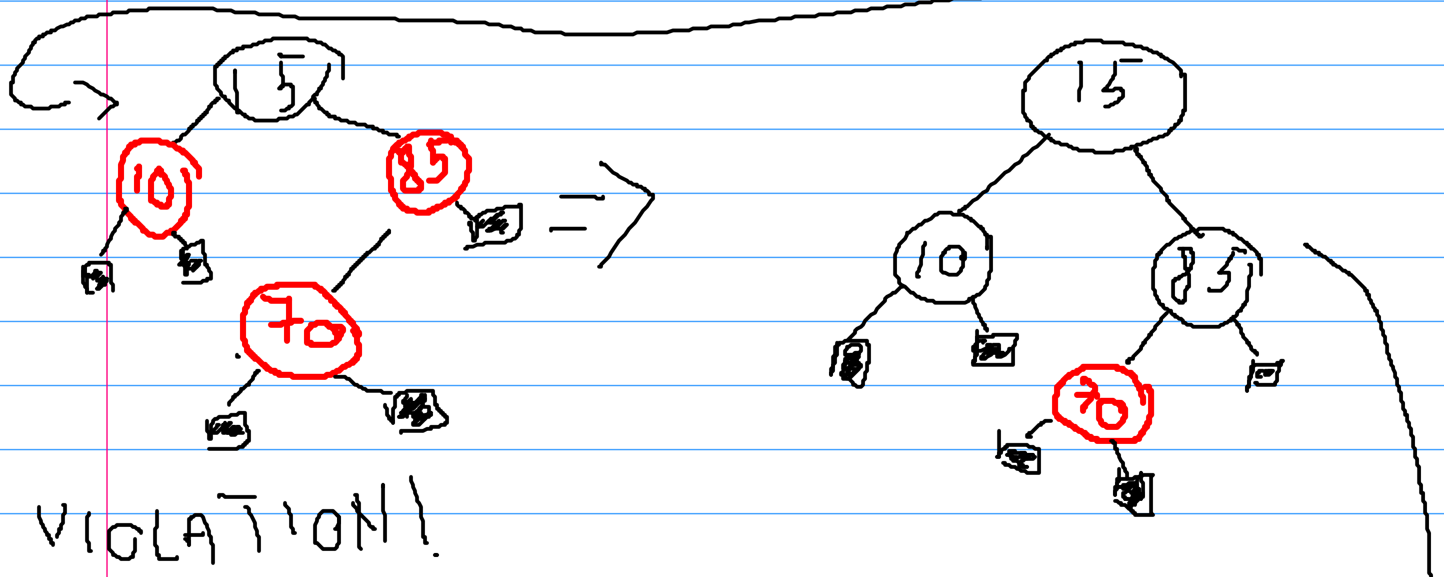
② Red-black trees

▨ = Nil node



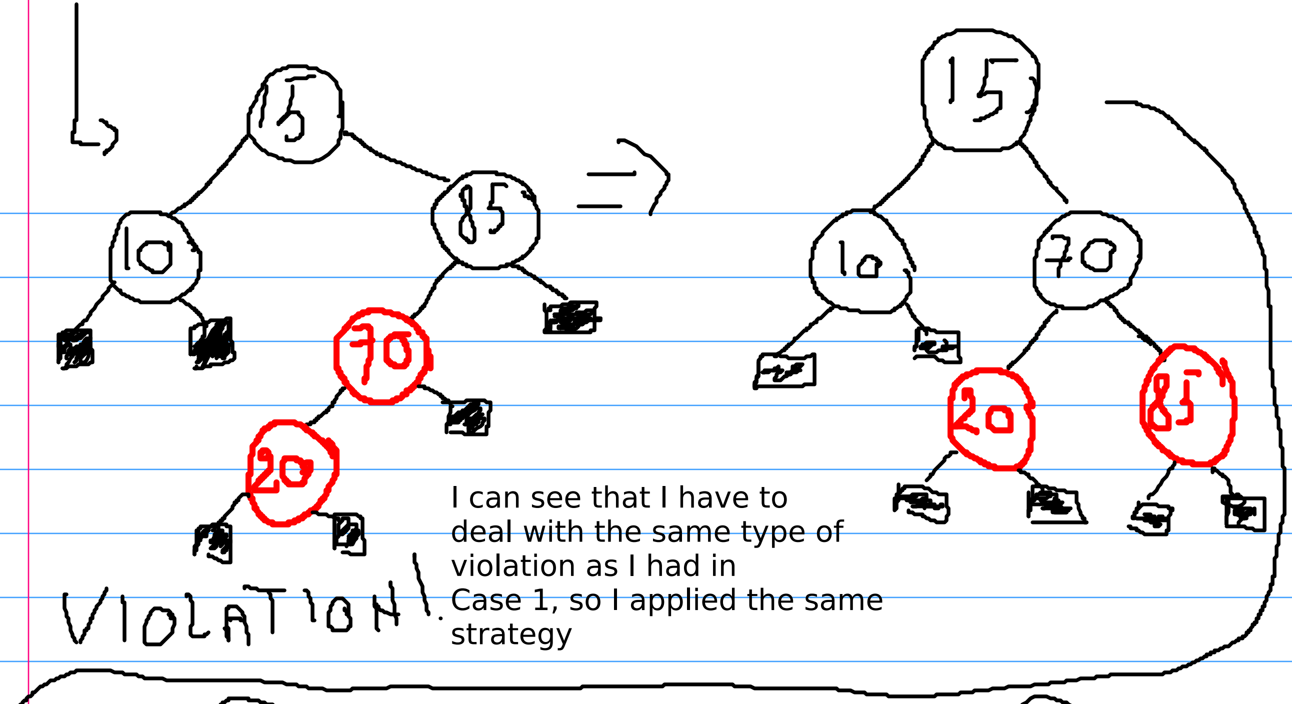


CASE 1 : I swaped nodes 15 and 85 and
I rotated the tree around node 15. I also changed
the color of 15 (which must be black, because it's
the root now) and of 10, in order to have an
equal number of black nodes on each path.

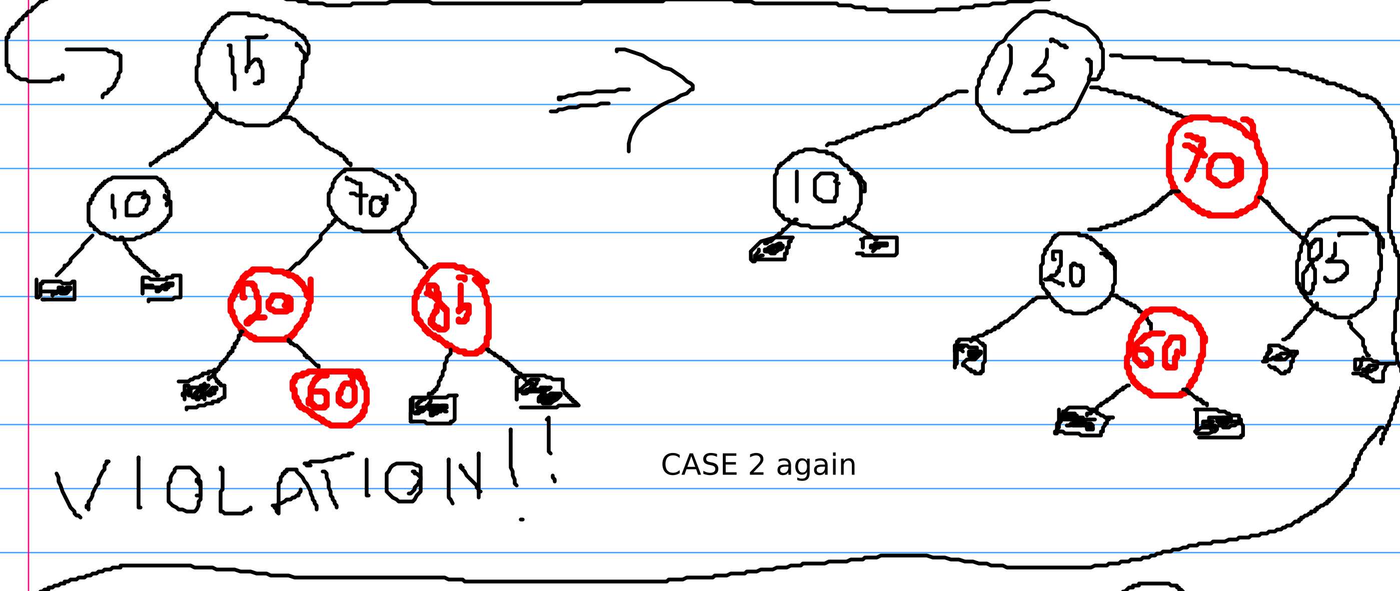


Case 2 : We have a RED "uncle".

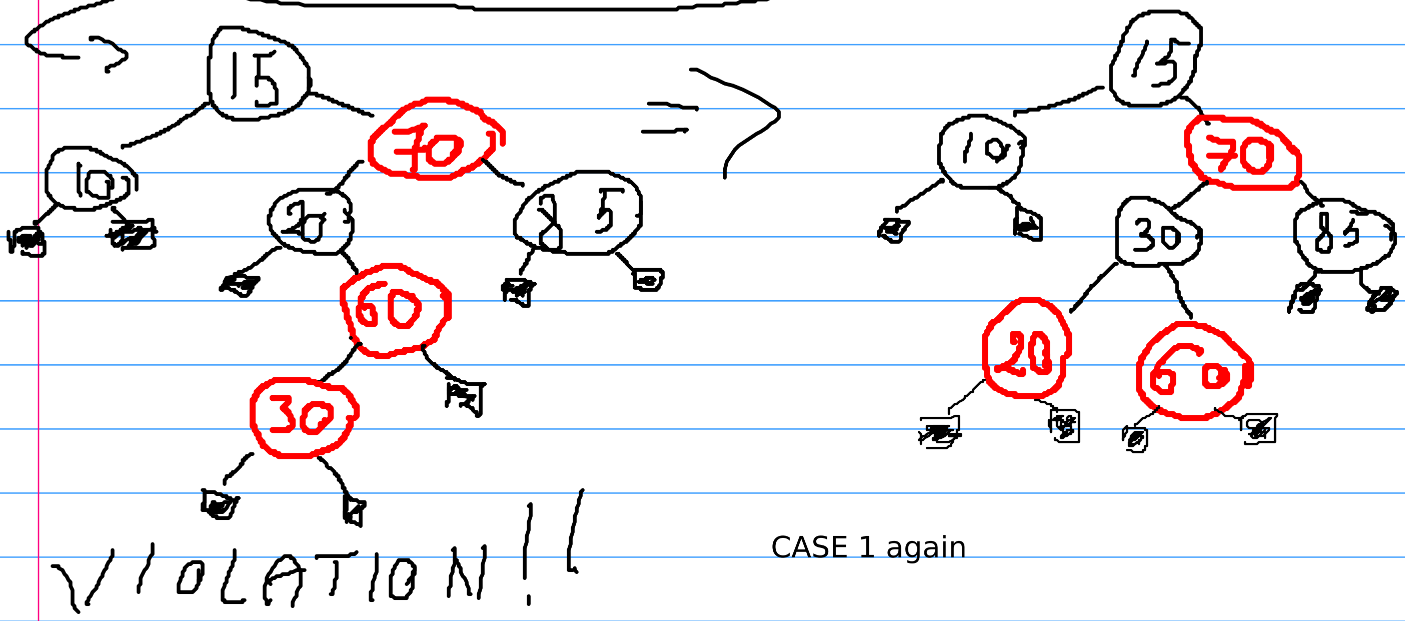
Solution : - Changing the color of the Parent,
Grandparent and uncle
- Checking further for other violations



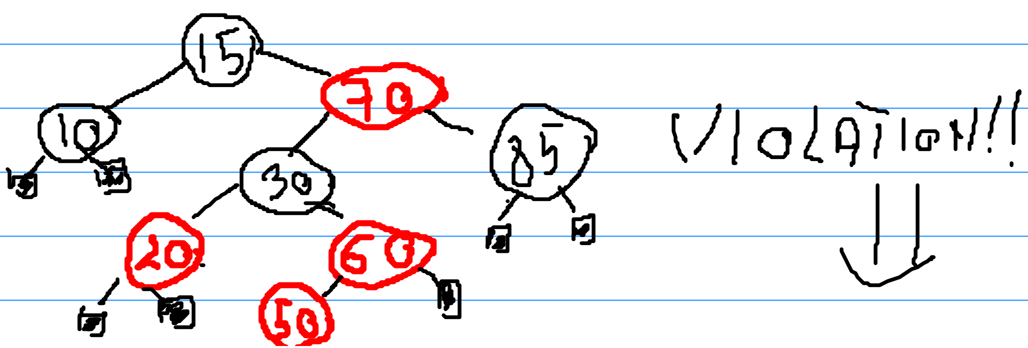
I can see that I have to
deal with the same type of
violation as I had in
Case 1, so I applied the same
strategy



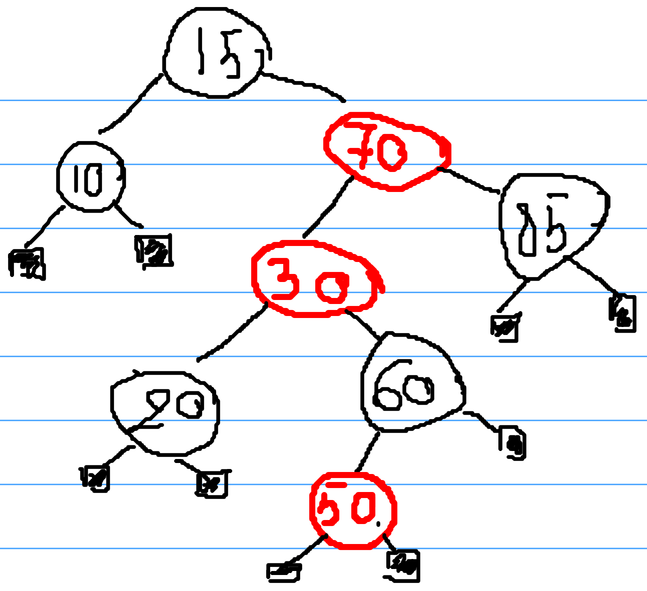
CASE 2 again



CASE 1 again



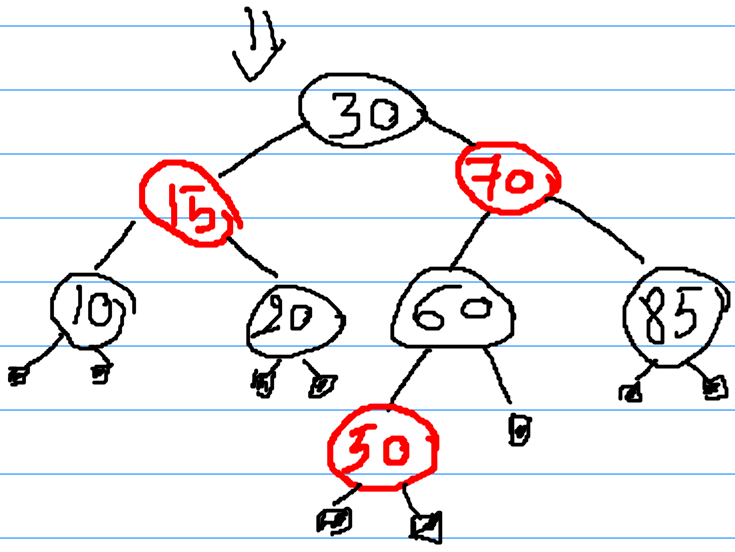
First, I apply the strategy from CASE 2 :

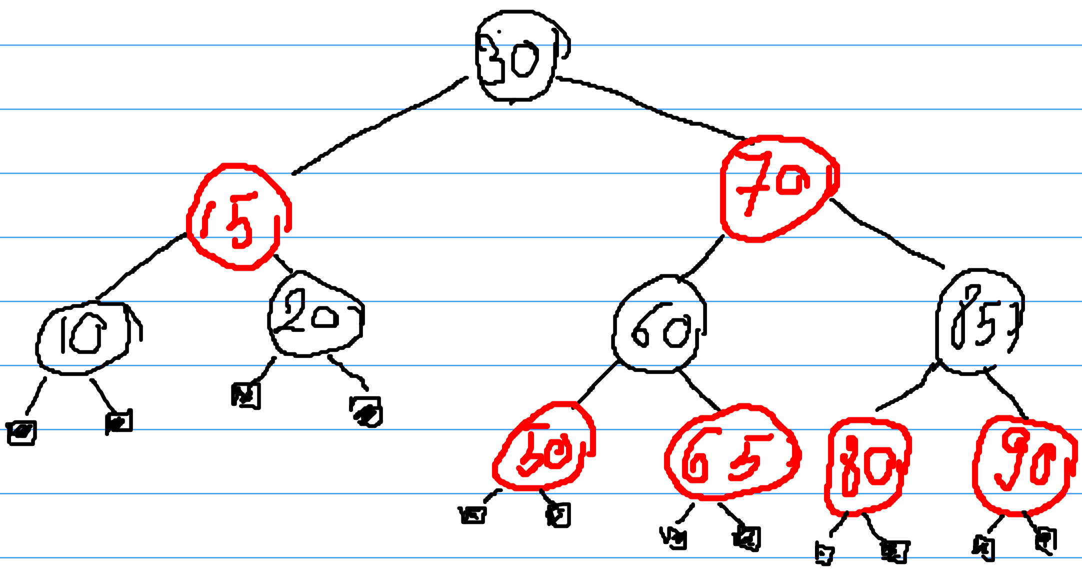


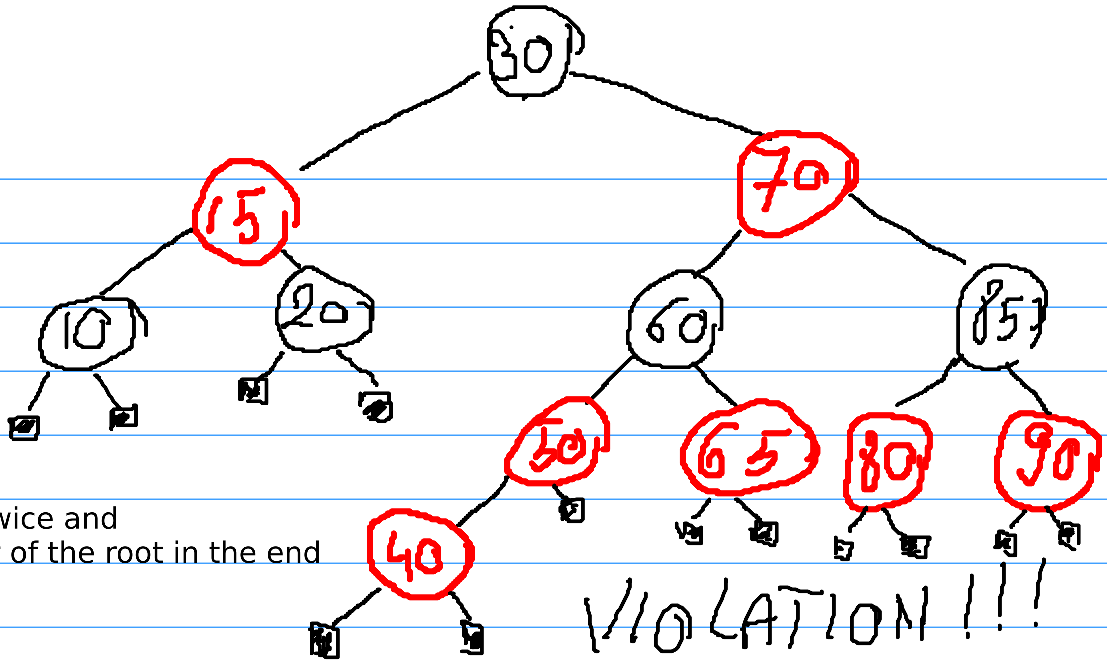
STILL VIOLATION!!

$\Downarrow$

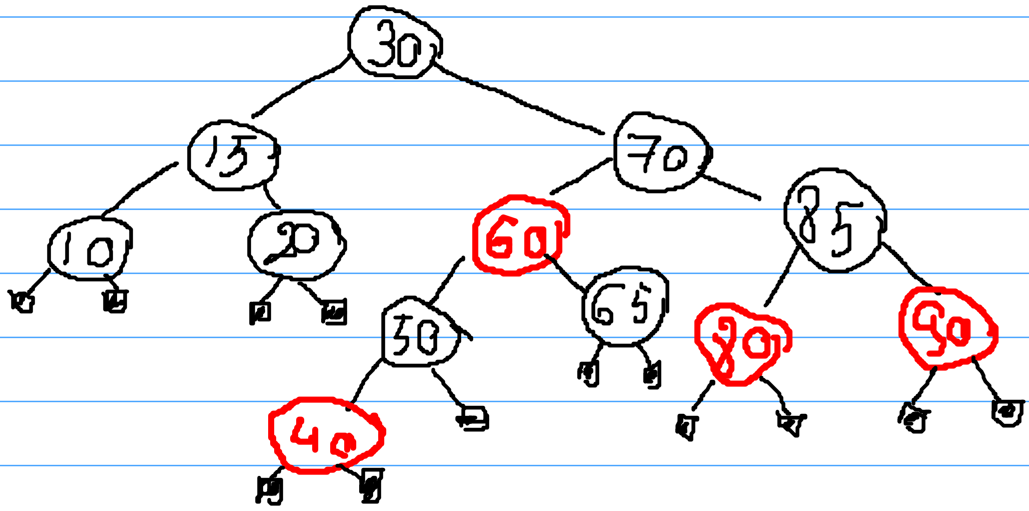
We apply the strategy from CASE 1 one more time.

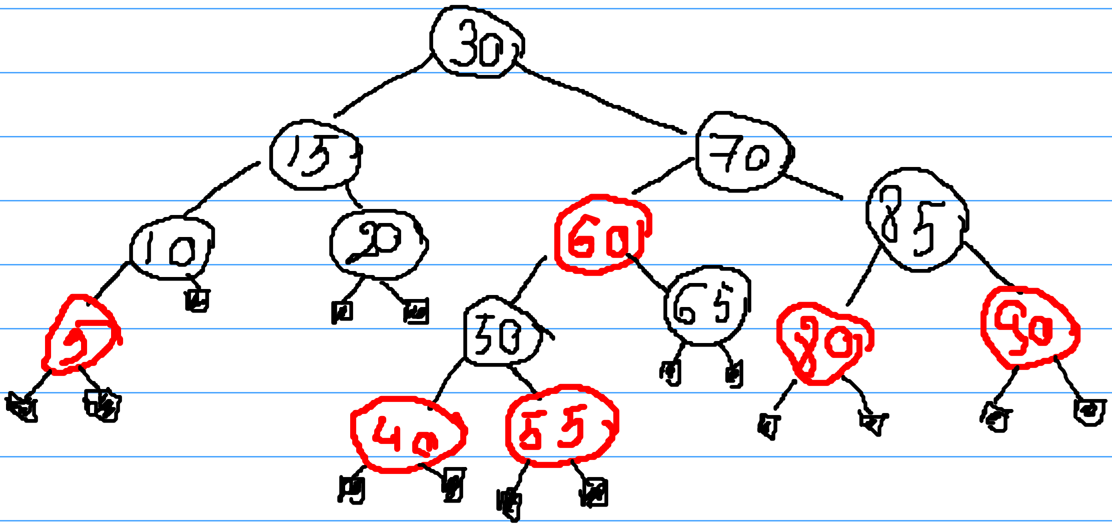






I apply CASE 2 twice and
change the color of the root in the end


30
15
70
10
20
60
85
50
65
80
90
40


30
15
70
10
20
60
85
5
50
65
80
90
40
55