R-4.1 Consider the insertion of items with the following keys into an initially empty AVL tree: 30, 40, 24, 58, 48, 26, 11, 13.

All the leaves (leaf) are empty.

R-4.3 Consider the insertion of items with the following keys into an initially empty splay tree: 0, 2, 4, 6, 8, 10, 12, 14, 16, 18.

All leaf's are empty.

R-4.6 What is the minimum number of nodes in an AVL tree of height 7?

\[ \text{n}(h) = 1 + \text{n}(h-1) + \text{n}(h-2) \quad \text{for} \quad h \geq 3 \]

\[ \text{n}(7) = 54 \]
R.4.7 What is the minimum number of nodes in a red-black tree of height 8?

\[ h_d = 2^d - 1 \]

\[ h = 2d + 1 \leq 2\log(n+1) + 1 \]

\[ d = 4, \quad \text{hence height} = 8 \]

\[ h_d = 2^4 - 1 \]

\[ h_d = 15 \]

R.4.15 Draw an example red-black tree that is not an AVL tree. Your tree should have at least 8 nodes, but no more than 16.

\[ \text{height 1} \quad \leftarrow \quad 1 \quad 4 \quad \rightarrow \quad \text{height 3} \]

\[ 3 - 1 = 2 \geq 1 \]

\[ \therefore \text{Not AVL} \]

C.4.1 Show that any n-node binary tree can be converted to any other n-node binary tree using \( O(n) \) rotations.

Left Rotate \((T, b)\)
One left rotation increases the number of nodes in the left chain by 1. As long as there is a right child or in a left chain, we may perform a right rotation on b.

Left rotations are well defined for binary tree at most $n-1$ left rotation are needed for putting all nodes on left chain.

Connecting $T_1$ to $T_2$.

Also $a_k$ left operation is invisible.

At most $n-1$ rotation needed to transform $T$ to $T_2$.

At most $2n-2$ rotations are needed from $T_1$ to $T_2$, which is $O(n)$.

\[c-4.3\]

\[n_h = 1 + n_{h-1} + n_{h-2} \geq 3\]

$k = 1, n_1 = 1$

\[F_3 - 1 = 1 \quad (\because F_3 - 1 = F_2 + F_1 - 1 = 2F_1 + F_0 - 1)\]

\[h = k\]

Suppose $n_k = F_{k+2} - 1$

Then $h = k + 1$, $n_h = 1 + n_k + n_{k-1}$

\[= 1 + F_{k+2} - 1 + F_{k+1} - 1\]

\[= F_{k+2} + F_{k+1} - 1\]

\[= F_{k+3} - 1\]

\[\therefore n_h = F_{k+2} - 1\]

\[\therefore n_h = F_{k+2} - 1\]

Show by induction, that the minimum number $n_h$ of internal nodes in an AVL tree of height $h$, as defined in the proof of Theorem 4.1, satisfies the following identity for $h \geq 1$.

\[n_h = F_{k+2} - 1\]

where $F_k$ denotes the Fibonacci number of order $k$, as defined in previous exercise.
Describe how to perform the operation `removeAllElements(k)`, which removes all elements with keys equal to k, in a balanced search tree T, and show that this method runs in time $O(c \log n)$, where $n$ is the number of elements stored in the tree and $s$ is the number of elements with key k.

```plaintext
sol

While True do
    T = balanced binary search tree
    k = current in question, key
    While True do
        v = search tree (k, T)
        If v is external node then
            return T
        If v has no external node child
            Let u be a node in T with key nearest to k
            v = u
        Let w be v's smallest height child
        Balance the +
        Remove w from T, by replacing
        v with w's siblinging +
        Balance the tree again (Z, T)
        notation technique
        which is used in AVL to
        rebalance

eight of tree = log n

two traversals $s$ times

O (c log n)
```
C-4.14 Define the height of the null (external nodes) to be 0, which means Height(Leaf) = 0, and for a specific node V, define Height(V) = 1 + \max(\text{Height}(\text{left child}(V)), \text{Height}(\text{right child}(V)))

The coloring strategy can be described as follows.

```
Black(V)
V. color <- black
If V is not external node
EndIf
ColorChildren(T.leftChild(V), T.rightChild(V))
```

```
Red(V)
V. color <- red
ColorChildren(T.leftChild(V), T.rightChild(V))
```

```
ColorChildren(x, y)
If (Height(x) < Height(y)) || (Height(y) is even)
  Black(x)
Else
  Red(x)
EndIf
If (Height(x) > Height(y)) || (Height(y) is even)
  Black(y)
Else
  Red(y)
EndIf
```

We begin the coloring procedure by calling Black(root), then the AVL tree transforms to the Red-Black Tree.
C-9.14 (Continue) According the above procedure, define h as the height of the AVL tree. We claim that

For \( n \geq 0 \), if \( n = 2n \), \( \text{Black}(n) \) generates a red-black tree with black-depth \( n+1 \);

(2) If \( h = 2n+1 \), \( \text{Red}(n) \) sets all paths with black-depth \( n+1 \). \( \text{Black}(n) \) creates a red-black tree with depth \( n+2 \).

Now we prove this assertion by induction.

① Base Case: When \( n = 0 \), \( h = 2n = 0 \), the tree only contains one leaf (degenerated case).

\( \text{Black}(0) \) generates one black node, and the depth is 1.

When \( n = 0 \), \( h = 2n+1 = 1 \), the root is an internal node with 2 children, both of which are marked by black. \( \text{Red}(1) \) sets all paths with black-depth 1. \( \text{Black}(1) \) generates a red-black tree with depth 2. So the base case holds.

② Suppose the hypothesis is true for \( n \), we show that it holds for \( n+1 \).

For \( h = 2(n+1) = 2n+2 \), there are 2 circumstances to consider:

(i) both subtrees have odd heights \( 2n+1 \) then

(i1) \( \text{Color}(n+1) \) calls \( \text{Red}(n) \) for each children.

(i2) Both children have depth \( n+1 \) (by hypothesis).

(i3) For parents, \( \text{Black}(n) \) adds a black node with depth \( n+2 \) (by hypothesis).

(ii) subtrees have heights \( 2n+1 \) and \( 2n \) respectively.

(ii1) \( \text{Color}(n+1) \) calls \( \text{Red}(n) \) and \( \text{Black}(n) \).

(ii2) For odd height \( 2n+1 \), \( \text{Red}(n) \) leads to black-depth \( n+1 \) (by hypothesis).

(ii3) For even height \( 2n \), \( \text{Black}(n) \) yields a tree with black-depth \( n+1 \) (by hypothesis).

(ii4) For parents, \( \text{Black}(n) \) adds a black node with depth \( n+2 \).

For \( h = 2(n+1) + 1 = 2n+3 \), there are also 2 circumstances. With similar argument,

We can prove that it is also true.

③ Hence completes the proof.
A-4.2: The main data structure we can use to solve this problem could be any balanced search trees. For example, we can implement it by AVL tree.

At first, we can define the following class and global variables. We have several settings listed below.

- **Setting 1**: Every tree node represents the information of an employee;
- **Setting 2**: Every "name" variable in the class "treeNode" can be mapped into a key with integer value, where the alphabetic order is exactly the same when compared with the way of adopting integer value comparison;
- **Setting 3**: All of those tree nodes are stored in an AVL tree, which has height $O(\log n)$, where $n$ is the size of the tree;
- **Setting 4**: Each Time inserting an employee, add $num$ by 1; every time removing a employee, decrease $num$ by 1;
- **Setting 5**: When Friday arrives, add $budget1$ by more share of stock being promised, and assign it to $budget2$. Then update every employee’s share to $share + (budget2 - budget1)/num$, and let $y = (budget2 - budget1)/num$;

```java
class treeNode{
    treeNode(String a){ // Constructor
        this.name = a;
        this.share = 0.0;
    }
    String name; // Employee’s Name
    double share; // Current Share
}
double budget1; // Old Budget for Employer
double budget2; // Updated Budget for Employer
int num; // The size of the AVL tree
int y; // The more shares for a specific Friday
```

Now we show that how the above implementation can achieve the requirements.

- Inserting or removing a node in an AVL tree spends $O(\log n)$ time, where $n$ is the size of the tree. This fact suffices to show that the first requirement is satisfied;
- According to setting 2, every "name" variable can perfectly map to a corresponding key without compromising the order of tree nodes. So Inorder traversal can easily list all nodes in alphabetical order in $O(n)$ time, and the relevant information, like "name", "share", is also accessible;
- When Friday arrives, we can compute $y$ by $(budget2 - budget1)/num$, which can be processed in constant time.