# Balanced Binary Search Tree

Bùi Tiến Lên

2022

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

# Contents

# Balanced Tree

- Rotations
- Strategies in Balancing Tree

# Introduction

**Balance** may be defined by:

- Comparing the numbers of nodes of the two subtrees
- Height balancing: comparing the heights of the twosub trees
- Null-path-length balancing: comparing the null-path-length of each of the two sub-trees
- Weight balancing: comparing the number of null sub-trees in each of the two sub trees

# Introduction (cont.)

## Concept 1

A binary tree is **balanced** if the difference in the numbers of nodes of both subtrees of any node in the tree either **zero** or **one**.

## Concept 2

A binary tree is **height-balanced** if the difference in height of both subtrees of any node in the tree either **zero** or **one**.

- A complete binary tree is is height-balanced

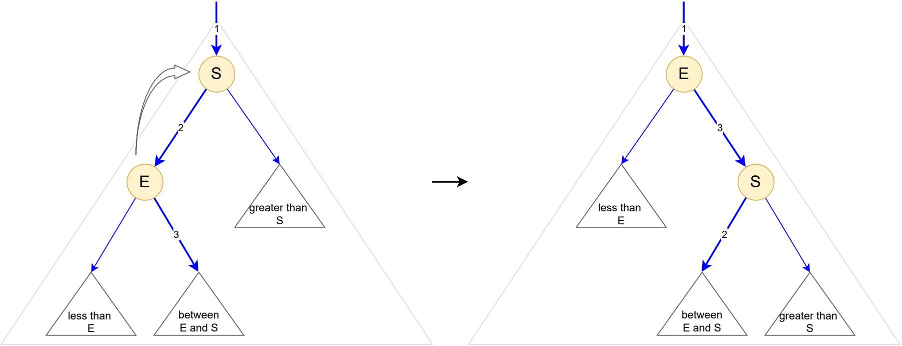
# Rotations

- A **rotation** allows us to interchange the role of the root and one of the root's children in a tree while still **preserving the BST ordering** among the keys in the nodes.
- There are two kinds of rotations: right rotation and left rotation

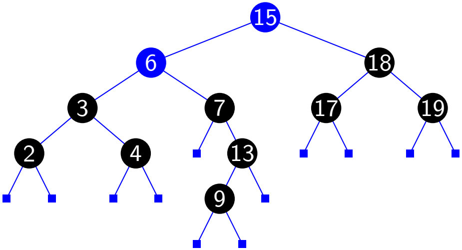
# Right rotation

A right rotation involves the root and the left child. The rotation puts the root on the right, essentially reversing the direction of the left link of the root:

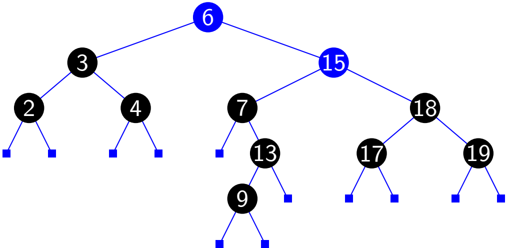
- Before the rotation, it points from the root to the left child
- After the rotation, it points from the old left child (the new root) to the old root (the right child of the new root)



# Example




- Make right rotation at 15

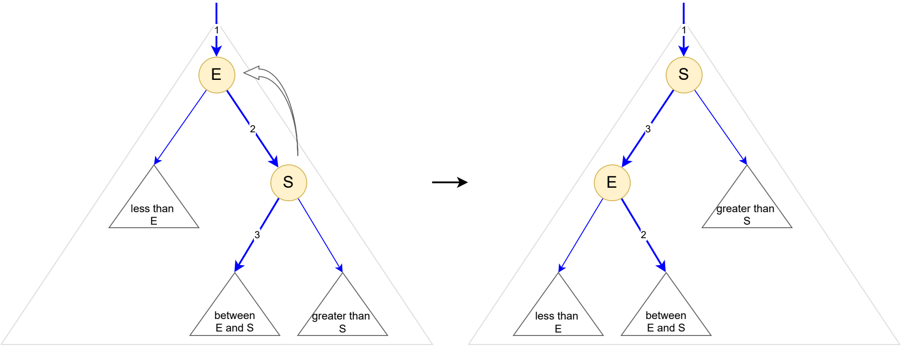


# Implementation

```
void rightRotate(link& h) {
  link x = h->left;
  h->left = x->right;
  x->right = h;
  h = x;
}
```

# Left rotation

A left rotation involves the root and the right child.



# Implementation

```
void leftRotate(link& h) {
  link x = h->right;
  h->right = x->left;
  x->left = h;
  h = x;
}
```

# Strategies in Balancing Tree

- **Global rebalancing**: an approach to producing better balance in BSTs is periodically to rebalance them explicitly.
  - costs at least linear time in the size of the tree
- **Local rebalancing**: balancing BSTs after each operation (insert, delete)

# DSW algorithm

The algorithm was proposed by Colin Day and later improved by Quentin F. Stout and Bette L. Warren.
Idea of algorithm:

1. Transform an arbitrary BST into a *linked-list*-like-tree called **backbone** or **vine** by rotations
2. Transform this tree into a **perfectly balanced tree** by rotations

# DSW algorithm (cont.)

$\textsc{createBackbone}(root)$

$\quad p := root$

$\quad$ **while** $p \neq null$?

$\quad\quad$ **if** $p$ has a left chid?

$\quad\quad\quad$ make right rotation at $p$

$\quad\quad$ **else**

$\quad\quad\quad p := p \rightarrow right$

# DSW algorithm (cont.)

CREATECOMPLETETREE(*root*)
- $n \leftarrow$ the number of nodes
- $m \leftarrow 2^{\lfloor \log_2(n+1) \rfloor} - 1$
- make $n - m$ left rotations starting from the top of backbone
- **while** $(m > 1)$
  - $m \leftarrow m/2$
  - make $m$ left rotations starting from the top of backbone

# Illustration

**Figure 1:** BST

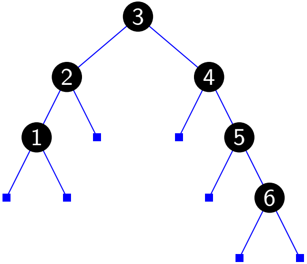



**Figure 2:** Backbone

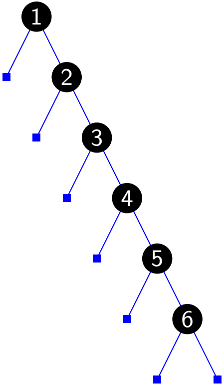
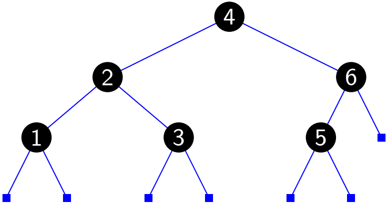



**Figure 3:** Perfect



# AVL Tree

- Insertion
- Deletion

# AVL Tree

## Concept 3

AVL tree

- proposed by two Soviet scientists G. M. Adelson-Velskii and E. M. Landis
- is BST tree which is **height-balanced**

$$\forall p : |\boldsymbol{height}(\boldsymbol{LeftSubtree}(p)) - \boldsymbol{height}(\boldsymbol{RightSubtree}(p))| \leq 1 \quad (1)$$

# The Height of an AVL Tree

Consider the worst case,

- To determine the maximum height that an AVL tree with $N$ nodes can have, we can instead ask what is the minimum number of nodes that an AVL tree of height $h$ can have (called AVL tree $F_h$).
- We have the recurrence relation

$$|F_h| = |F_{h-1}| + |F_{h-2}| + 1 \tag{2}$$

where $|F_0| = 1$ and $|F_1| = 2$

- Solve the equation, we have

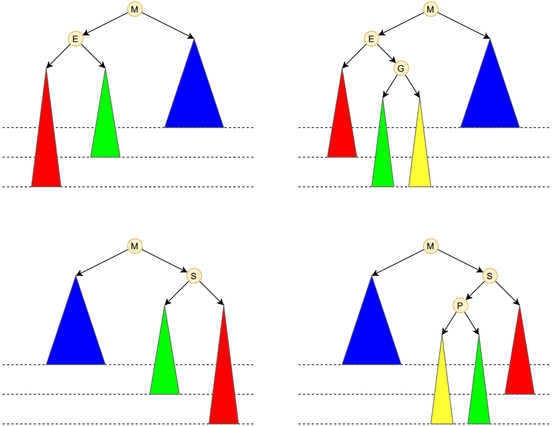
$$|F_h| + 1 \approx \frac{1}{\sqrt{5}} \left[ \frac{1+\sqrt{5}}{2} \right]^{h+2} \tag{3}$$

- The height of an AVL tree in the worst case

$$h \approx 1.44 \log_2 |F_h| = 1.44 \log_2 N \tag{4}$$

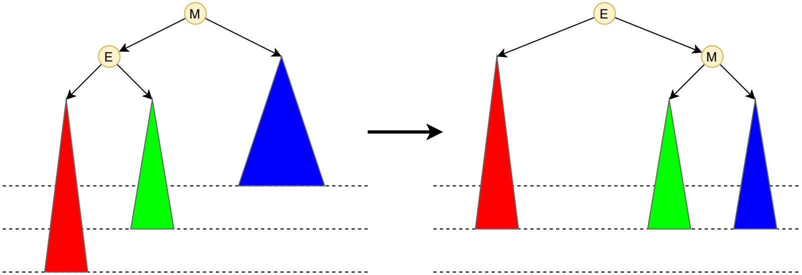
# Rebalancing technique

- After an insertion/deletion, we may find a node whose new balance violates the AVL condition.
- Four cases: LL imbalance, LR imbalance, RR imbalance, RL imbalance



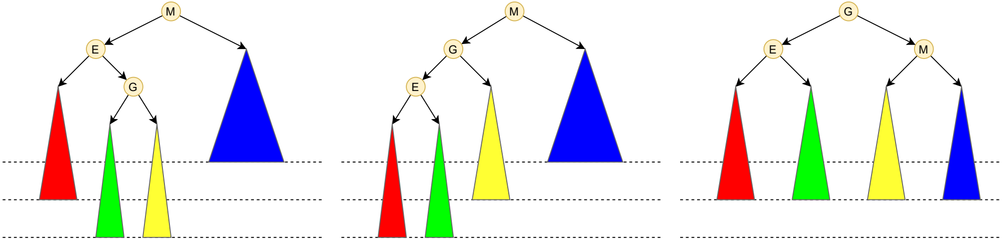
# Rebalancing technique (cont.)

- Case LL imbalance is corrected by executing a single right rotation at the node with the imbalance.



# Rebalancing technique (cont.)

- Case LR imbalance is corrected by executing a double rotations



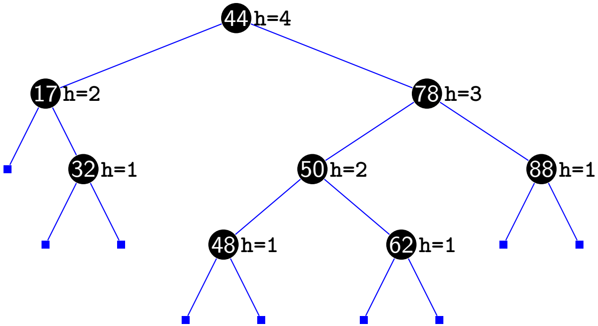
# Insertion

$\textsc{Insert}(\mathit{root}, \mathit{key})$

- **if** $\mathit{root} = \mathit{null}$
  - $\mathit{root} :=$ **new** $\textsc{Node}(\mathit{key})$
  - **return**
- **if** $\mathit{root} \rightarrow \mathit{key} = \mathit{key}$
  - **return**
- **if** $\mathit{root} \rightarrow \mathit{key} < \mathit{key}$
  - $\textsc{Insert}(\mathit{root} \rightarrow \mathit{right}, \mathit{key})$
- **if** $\mathit{root} \rightarrow \mathit{key} > \mathit{key}$
  - $\textsc{Insert}(\mathit{root} \rightarrow \mathit{left}, \mathit{key})$
- **if** unbalanced at *root*? rebalance at *root*

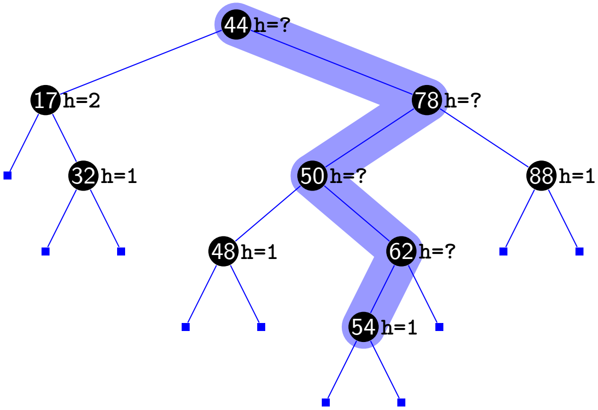
# Illustration

- An AVL tree



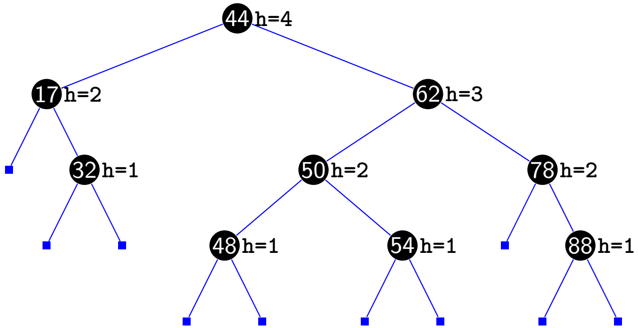
# Illustration (cont.)

- Insert node 54 into the tree → node 78 become unbalanced



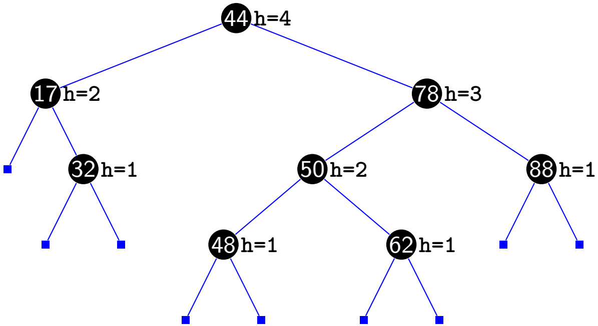
# Illustration (cont.)

- Case RL imbalance $\rightarrow$ rebalance by making double rotations



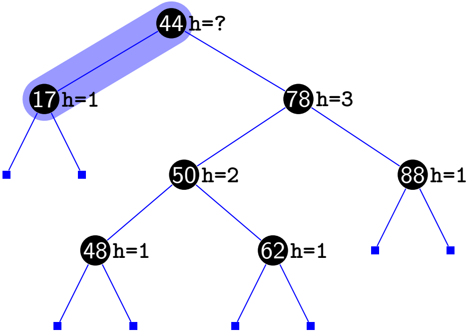
# Illustration

- An AVL tree



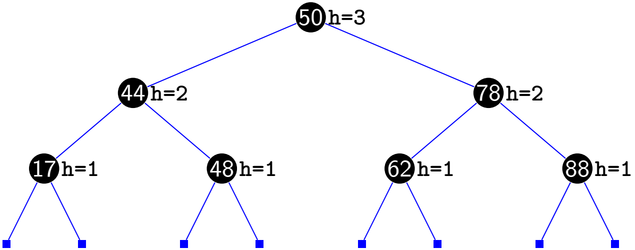
# Illustration (cont.)

- Delete node 32 from the tree → node 44 become unbalanced



# Illustration (cont.)

- Case RL imbalance → rebalance by making double rotations



# Red-Black Tree

- Insertion
- Deletion

# Red-Black Tree

## Concept 4

A red–black (**RB**) tree is a special type of binary search tree that must statsify

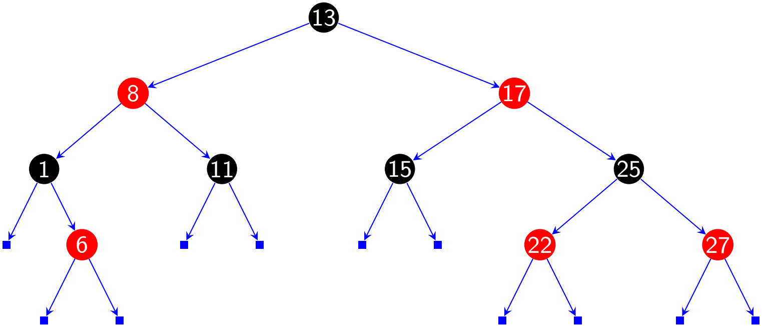
1. Each node is either **red** or **black**.
2. The root is **black**. (sometimes omitted)
3. If a node is **red**, then both its children are **black**.
4. Every path from a given node to any of its descendant null link has the same number of **black** nodes (balance criteria).

## Concept 5

A left-leaning red–black (**LLRB**) tree (leveraging Andersson's idea AA tree) is a variant of the red–black tree that has only left red children

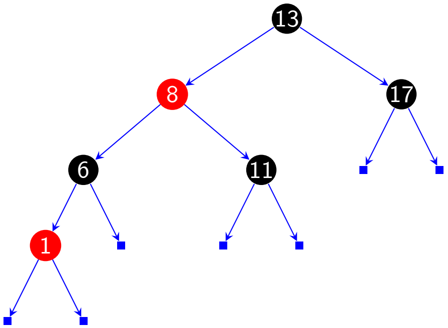
# Example

- A red–black tree



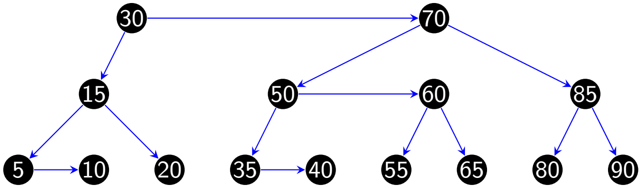
# Example (cont.)

- A left-leaning red–black tree



# Example (cont.)

- AA tree



# The Height of a RB Tree

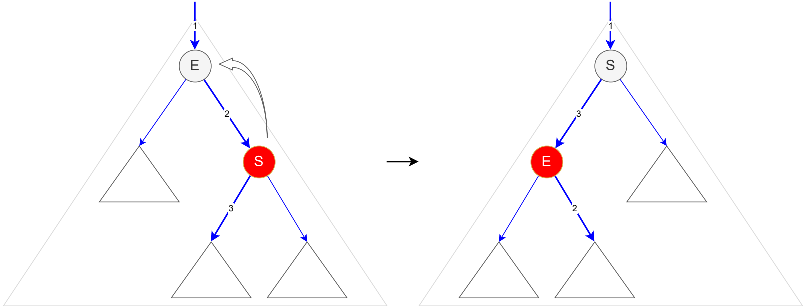
**Theorem 1**

*The height of a red-black BST with $N$ nodes is no more than $2\log_2 N$. It means that the height of an RB tree in the worst case*

$$h \leq 2\log_2 N \tag{5}$$

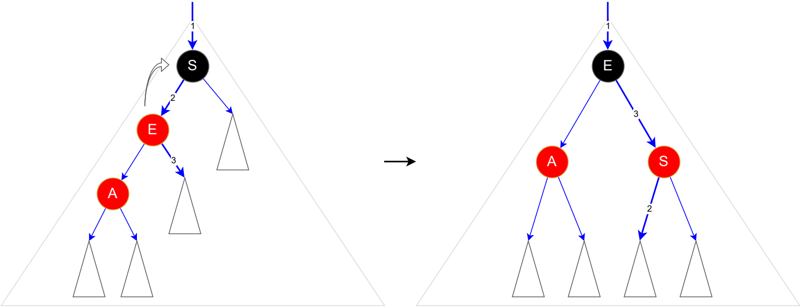
# Operations

- Case 1: Left rotation to orient a right red node to left red node.



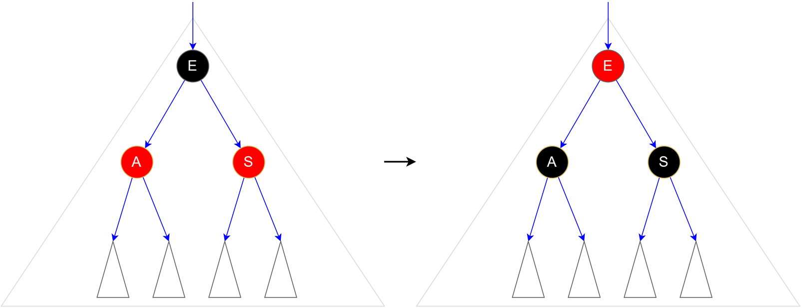
# Operations (cont.)

- Case 2: Right rotation.



# Operations (cont.)

- Case 3: Color flip.



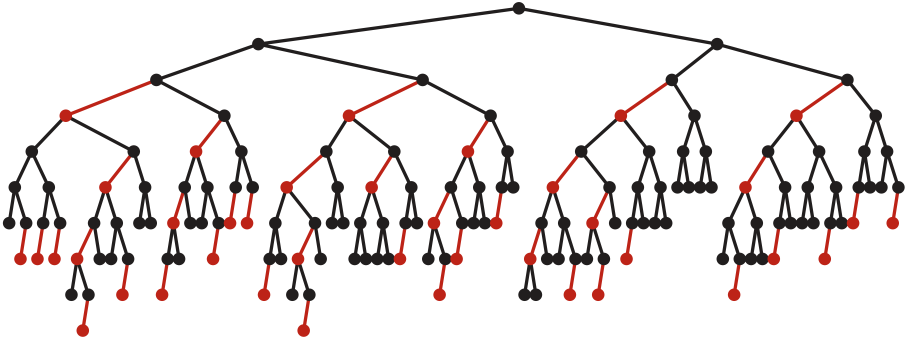
# Insertion

$\textsc{Insert}(root, key)$

- **if** $root = null$
  - $root := \textbf{new } \textsc{Node}(key)$ *red node*
  - **return**
- **if** $root \rightarrow key = key$
  - **return**
- **if** $root \rightarrow key < key$
  - $\textsc{Insert}(root \rightarrow right, key)$
- **if** $root \rightarrow key > key$
  - $\textsc{Insert}(root \rightarrow left, key)$
- **if** $\textsc{isRed}(root \rightarrow right)$ **and not** $\textsc{isRed}(root \rightarrow left)$ $\textsc{rotateLeft}(root)$
- **if** $\textsc{isRed}(root \rightarrow left)$ **and** $\textsc{isRed}(root \rightarrow left \rightarrow left)$ $\textsc{rotateRight}(root)$
- **if** $\textsc{isRed}(root \rightarrow left)$ **and** $\textsc{isRed}(root \rightarrow right)$ $\textsc{flipColors}(root)$

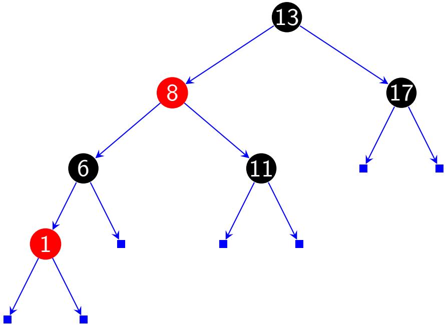
# Example

- Typical left-leaning red-black BST built from random keys



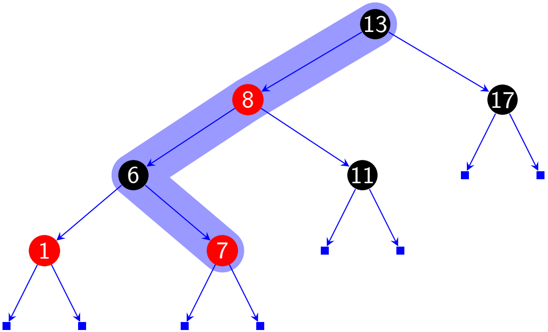
# Illustration

- A left-leaning red–black tree



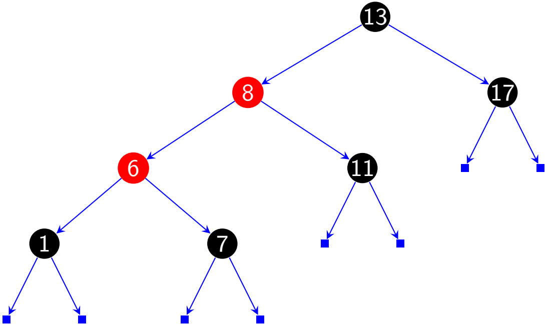
# Illustration (cont.)

- Insert node 7 into the tree



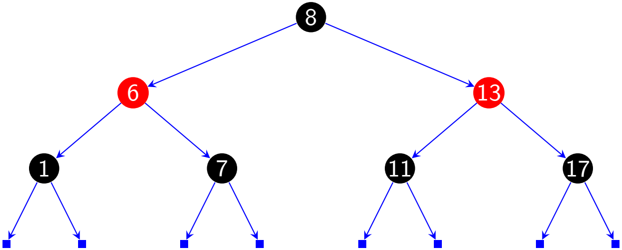
# Illustration (cont.)

- Flip color at 6 into the tree



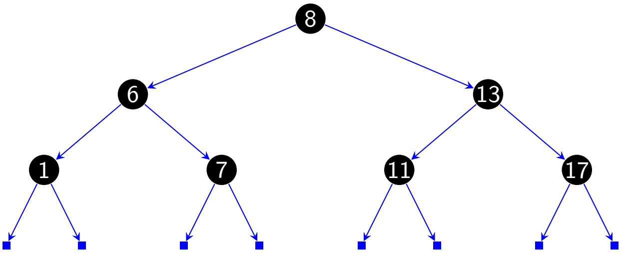
# Illustration (cont.)

- Right rotation at 13 into the tree



# Illustration (cont.)

- Flip color at root at change root to black color



# Cost summary for symbol-table implementations

| implementation | worst case | | | average case | | | key |
|---|---|---|---|---|---|---|---|
| | search | insert | remove | search hit | insert | remove | |
| **unordered list** | $N$ | 1 | $N$ | $N/2$ | 1 | $N/2$ | equal |
| **ordered list** | $N$ | $N$ | $N$ | $N/2$ | $N/2$ | $N/2$ | compare |
| **ordered array** | $\log_2 N$ | $N$ | $N$ | $\log_2 N$ | $N/2$ | $N/2$ | compare |
| **BST** | $N$ | $N$ | $N$ | $c\log_2 N$ | $c\log_2 N$ | $\sqrt{N}$ | compare |
| **AVL** | $c_a\log_2 N$ | - | - | $\log_2 N$ | - | - | compare |
| **RB** | $c_r\log_2 N$ | - | - | $\log_2 N$ | - | - | compare |
| **goal?** | | | | | | | |

**Note**: $c = 1.39, c_a = 1.44, c_r = 2.0$

# Optimal binary search trees

- Static
- Dynamic

# Introduction

## Concept 6

An **optimal binary search tree** (optimal BST), sometimes called a weight-balanced binary tree, is a binary search tree which provides the smallest possible search time (or expected search time) for a given sequence of accesses (or access probabilities)

- Optimal BSTs are generally divided into two types: **static** and **dynamic**
  - In the static optimality problem, the tree cannot be modified after it has been constructed.
  - In the dynamic optimality problem, the tree can be modified at any time, typically by permitting tree rotations.

# Introduction

- Suppose that we are designing a binary search tree for a program to translate text from English to Vietnamese, we want words that occur frequently in the text to be placed nearer the root.

## Problem

Given a sequence of of $n$ distinct keys in sorted order $(k_1 < k_2 < \cdots < k_n)$ and their frequencies

$$\mathcal{D}=\begin{array}{|c|c|c|c|c|}\hline \text{key} & k_1 & k_2 & \dots & k_n \\ \hline \text{frequency} & f_1 & f_2 & & f_n \\ \hline \end{array}$$

What binary search tree has the **lowest search cost**?

# Search Cost

- Cost of search for key $k_i$

$$\textsc{Cost}(k_i) = \textsc{Depth}(k_i) \tag{6}$$

  where $\textsc{Depth}(root) = 1$

- Denote $\textsc{ExpectCost}(l, r)$ be expected cost of search for a BST tree containing $\{k_l, ..., k_r\}$ given $\mathcal{D}$

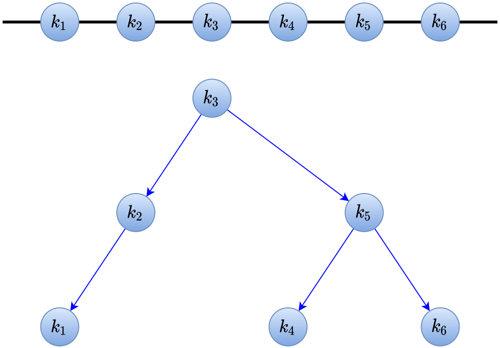
$$\textsc{ExpectCost}(l, r) = \sum_{i=l}^{r} \textsc{Cost}(k_i) f_i \tag{7}$$

# Search Cost (cont.)

- Compute the expected cost for the following binary search tree given

$\mathcal{D}=$

| key | $k_1$ | $k_2$ | $k_3$ | $k_4$ | $k_5$ | $k_6$ |
|---|---|---|---|---|---|---|
| frequency | 0.1 | 0.2 | 0.1 | 0.3 | 0.2 | 0.1 |



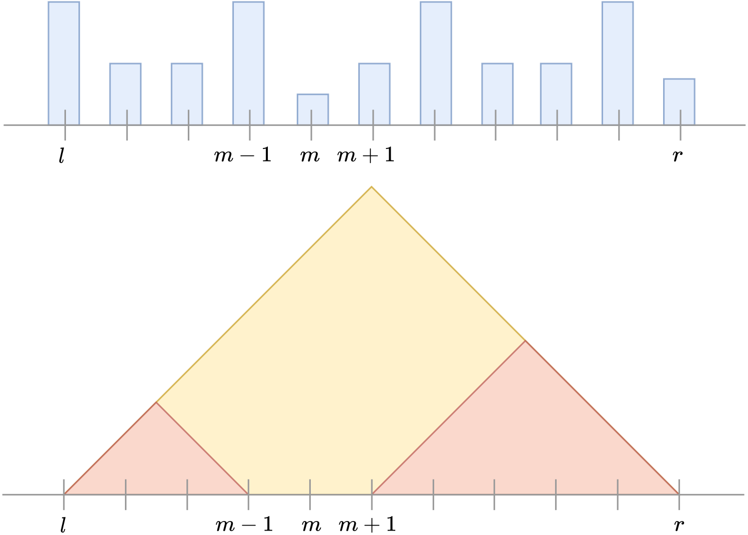
# Optimal Search Cost

- Denote $\textsc{OptimalCost}(l, r)$ be optimal cost of search for a BST tree $T$ containing $\{k_l, ..., k_r\}$ given $\mathcal{D}$
- Denote $\textsc{OptimalCost}(l, m, r)$ be optimal cost of search for a BST tree $T$ containing $\{k_l, ..., k_r\}$ and $k_m$ be the root node given $\mathcal{D}$
- We have

$$\begin{aligned}\textsc{OptimalCost}(l, m, r) = &\sum_{i=l}^{r} f_i \\ &+ \textsc{OptimalCost}(l, m-1) \\ &+ \textsc{OptimalCost}(m+1, r) \end{aligned} \tag{8}$$

$$\textsc{OptimalCost}(l, r) = \min_{m \in \{l, ... r\}} \{\textsc{OptimalCost}(l, m, r)\} \tag{9}$$

# Optimal Search Cost (cont.)



# Splay tree

## Concept 7

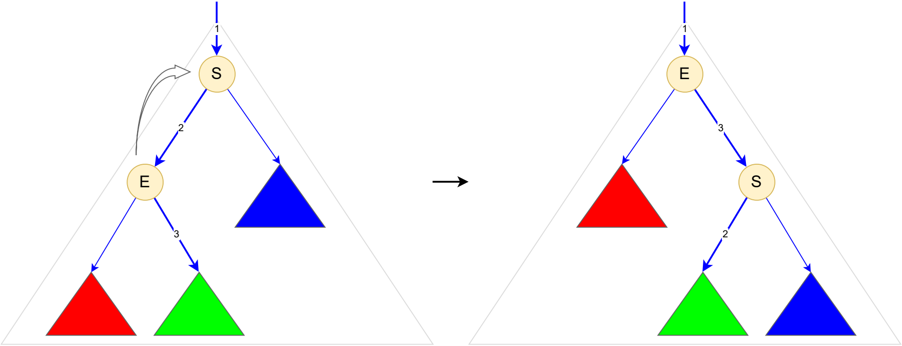
A **splay tree** is a binary search tree with the additional property that recently accessed elements are quick to access again.

- All normal operations (*insert*, *look-up*) on a binary search tree are combined with one basic operation, called **splaying**.
- For many sequences of non-random operations, splay trees perform better than other binary search trees.

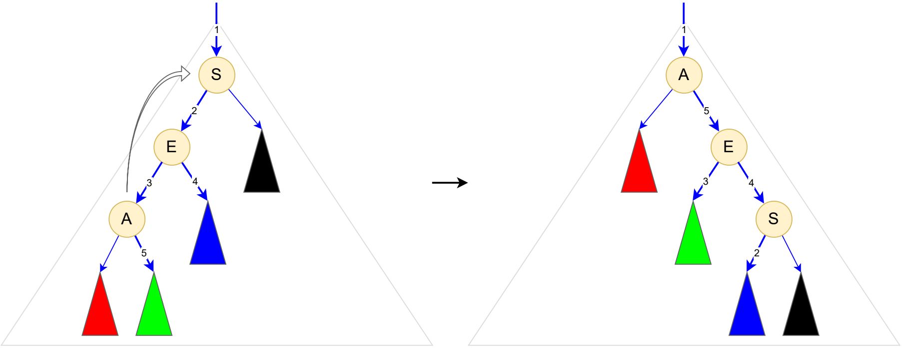
# Splaying

- When a node $x$ is accessed, a splay operation is performed on $x$ to move it to the root.
- To perform a splay operation we carry out a sequence of *splay steps*, each of which moves $x$ closer to the root.
- There are three types of splay steps, each of which has two symmetric variants:
  - zig step
  - zig-zig step
  - zig-zag step

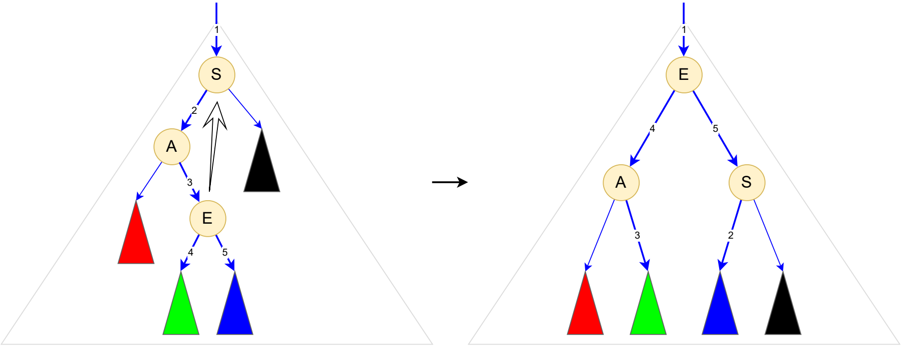
# Zig step



# Zig-zig



# Zig-zag



# Workshop

# Quiz

1. What is an AVL tree?

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

2. What is a Red-black tree?

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .

# Exercises

- Programming exercises in [Cormen, 2009, Sedgewick, 2002]

# References


Cormen, T. H. (2009).
*Introduction to algorithms*.
MIT press.

Sedgewick, R. (2002).
*Algorithms in Java, Parts 1-4*, volume 1.
Addison-Wesley Professional.

Walls and Mirrors (2014).
*Data Abstraction And Problem Solving with C++*.
Pearson.