Academic excellence for
business and the professions

IN2002 Data Structures and Algorithms

Lecture 8 – Advanced Trees

Aravin Naren
Semester 1, 2018/19

Learning Objectives

- Understand and be able to use the data structures binary search trees, splay trees and B-trees

- Be able to understand, apply and develop algorithms to handle the data structures above. Including:
  - Access to keys in trees
  - Balancing trees
  - Adjusting trees

Balanced Trees

Remember…Analysis of binary search trees

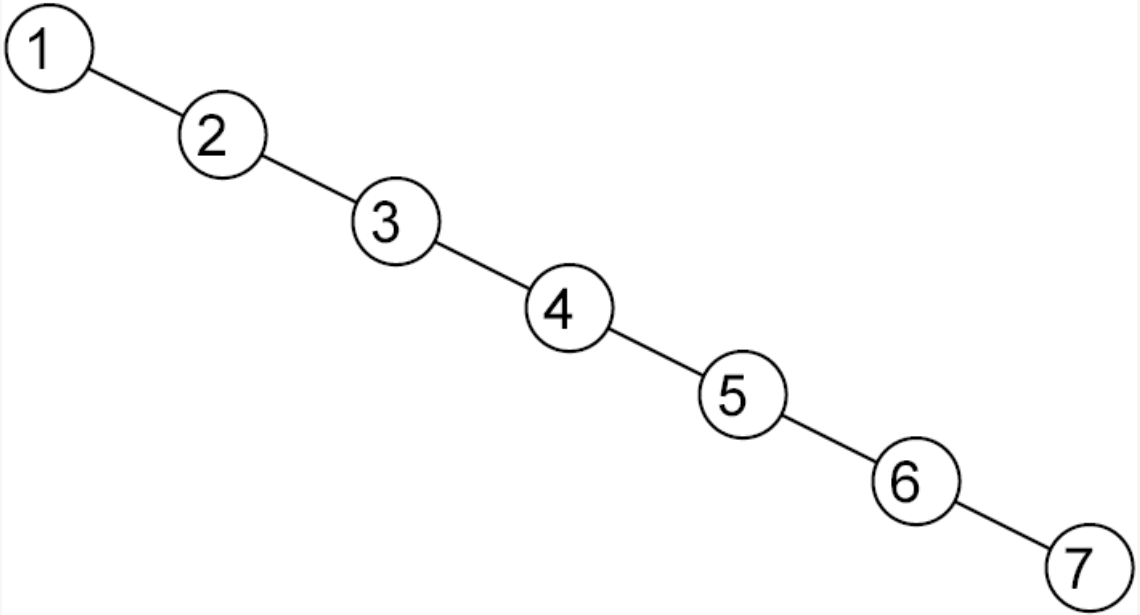
- Search, delete and insert have the height of the tree as the worst case for running time.
- Search, delete and insert take *O(log n),* where *n* is the number of nodes in the tree, if the tree is close to perfectly balanced.
- Otherwise, they can be as bad as *O(n)*
- **Problem**: insertion and deletion make the tree less balanced.
- **Solution**: keep the tree approximately balanced, without making insertion and deletion too expensive

The importance of trees being balanced

- To access a node in a binary search tree, the path is visited from the root to the node.
- This means that the maximum number of nodes visited is equal to the height of the tree.
- In a perfectly balanced tree, this is log(n)+1.

A not-so-balanced tree

- The maximum number of levels of a tree with *n* node is *n*.



Should the tree be kept balanced?

The **height *h* of the tree** will be between **log(*n*)+1 and *n***, so the time complexity of access to the tree will be between O(log n) and O(n)

Ideally, the tree should be kept **perfectly balanced**, with ***h* = log (*n*) + 1**. Doing this after insertion or deletion takes O(n) time, which defeats the purpose of efficiency.

Solution: keep the tree balanced enough to obtain O(log n) height.

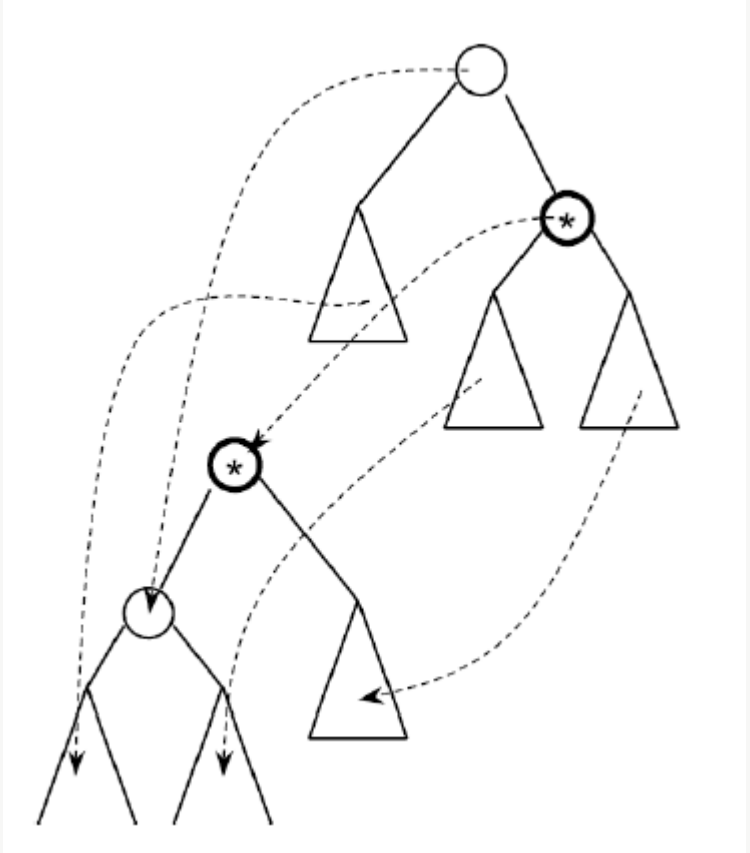
Some approximate balance definitions

Weight-balanced trees:
each node has 1/k <= *size(right)/size(left)* <= k
AVL trees (height-balanced):
each node has -1 <= *height(right) - height(left)* <= 1
(AVL-balance)
Red-Black trees: each node is either red or black, and no red node can have a red child.
Each full path from the root contains the same number of black nodes.

Balancing Trees

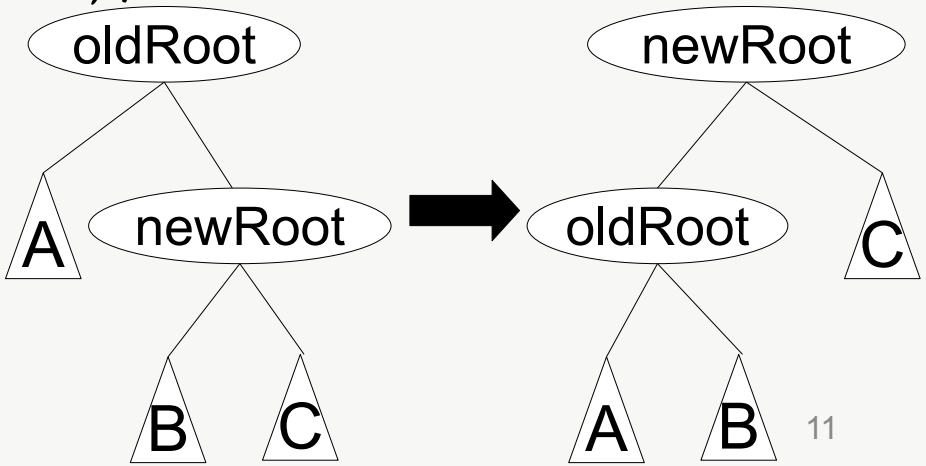
Tree Rotations

- Rotation is a basic operation in many tree re-balancing schemes
- Rotations keep the search tree structure
- Rotations can be applied anywhere in the tree



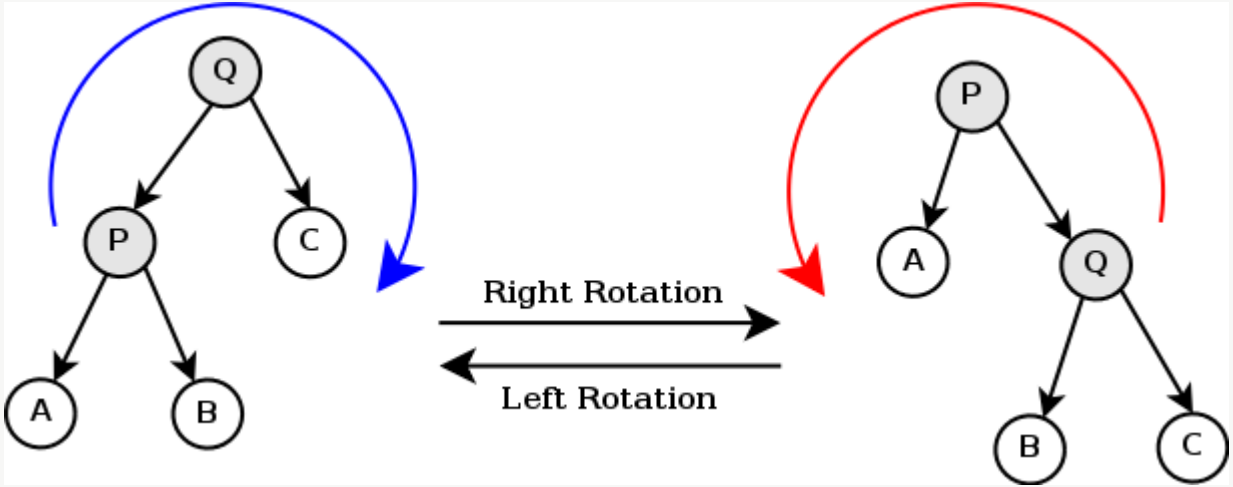
Tree Rotation in Java

```
public TreeNode rotateLeft(TreeNode oldRoot) {
    TreeNode newRoot = oldRoot.getRight();
    if(newRoot != null){
       oldRoot.setRight(newRoot.getLeft());
       newRoot.setLeft(oldRoot);
    }
    return newRoot;
    // needs to checked
    // by caller
}
```

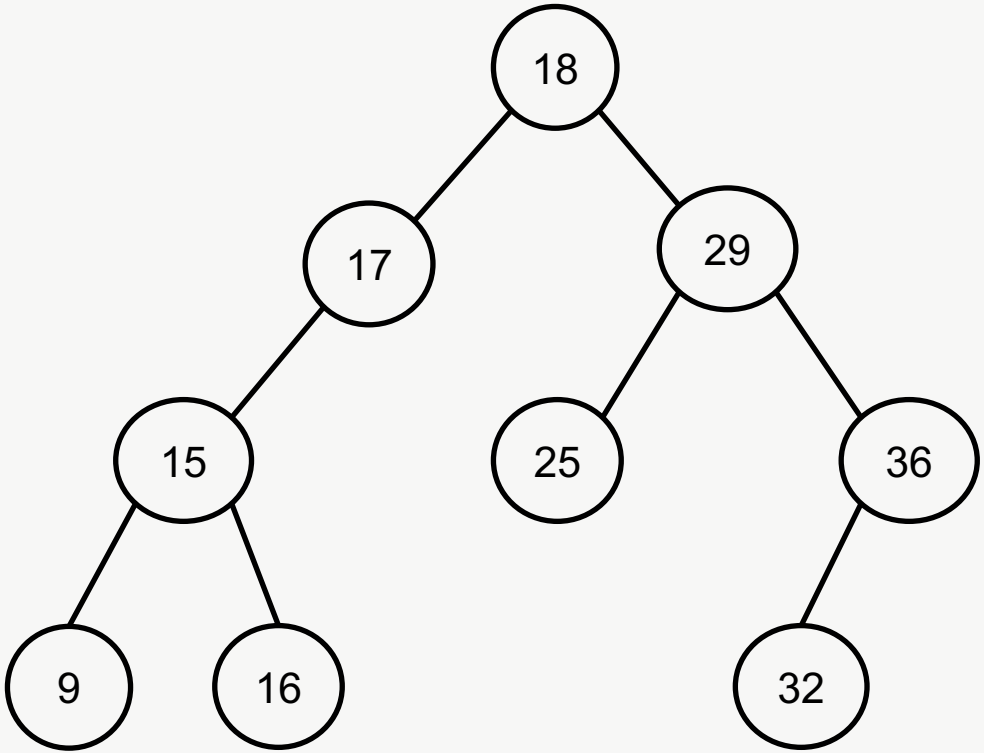


Left and Right Tree Rotations

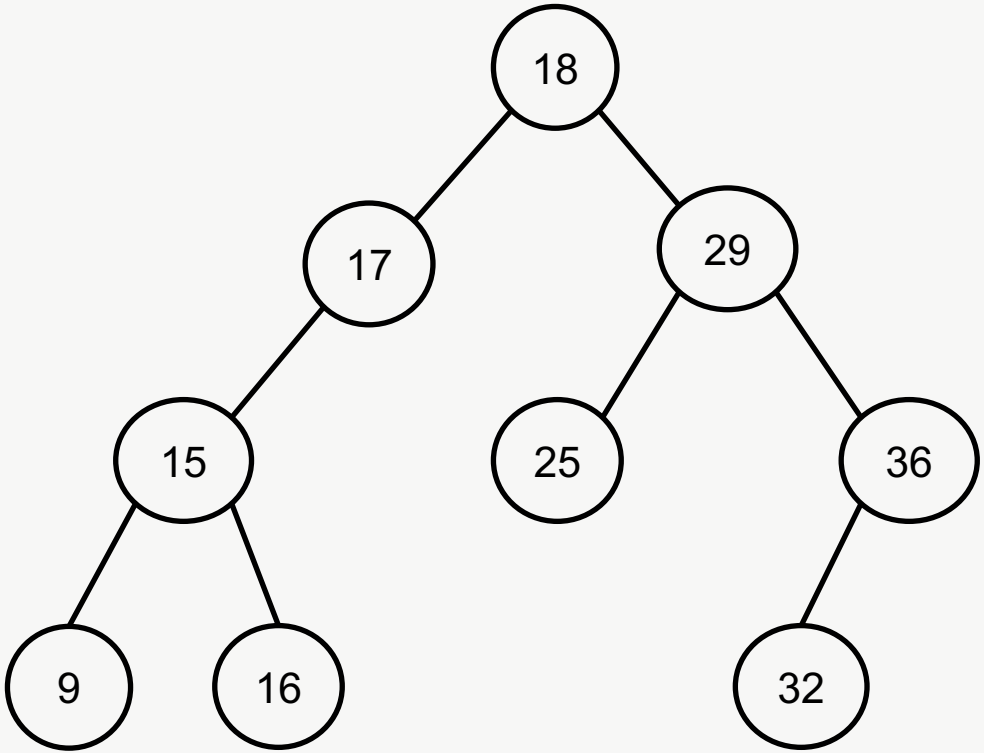
- Rotations can be applied in two directions



Left Tree Rotation Example



Right Tree Rotation Example



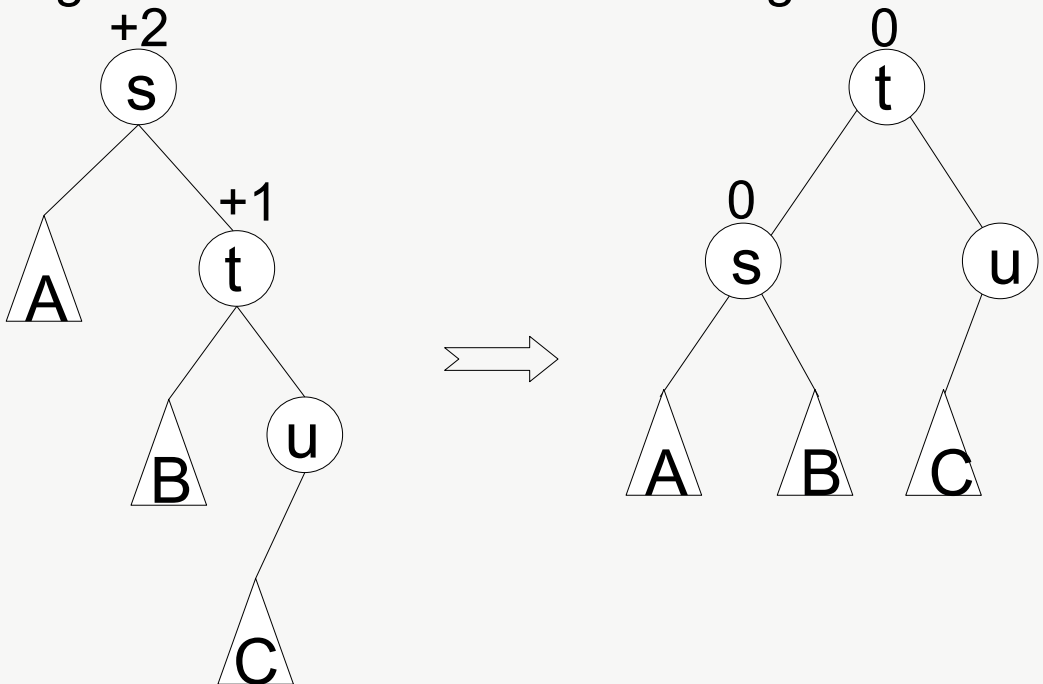
Re-balancing an AVL tree

If insertion invalidates the AVL condition (-1 <= *height(right) - height(left)* <= 1), use rotations to restore it:

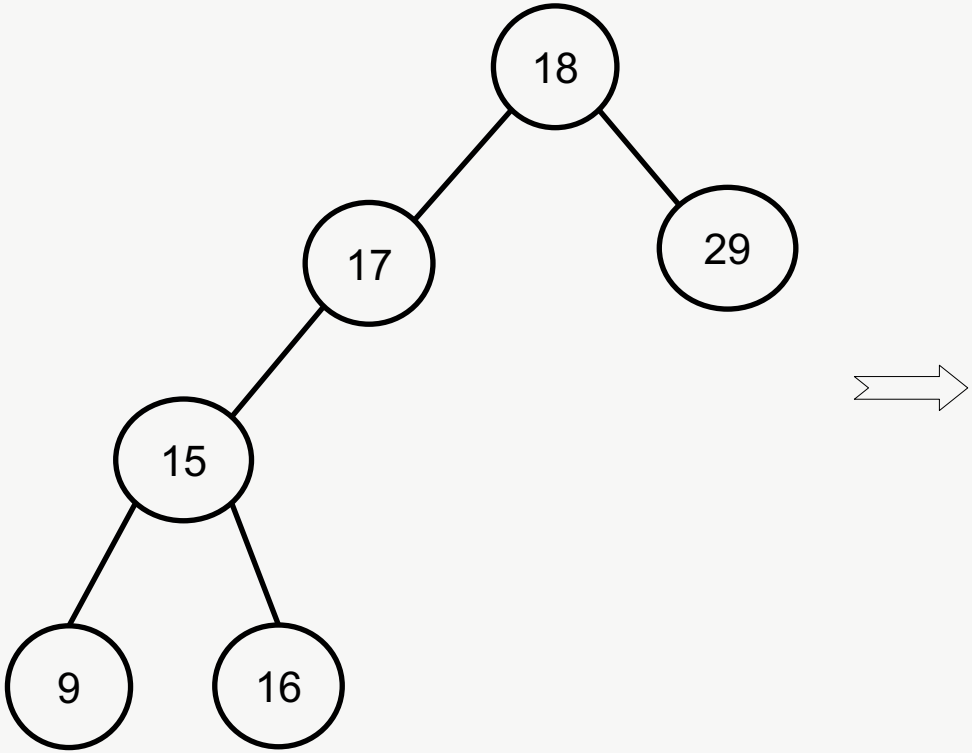
- For insertion, at most two O(1) rotations are required (but finding the position still takes O(log n))
- Deletions may require rotating any nodes in the path to the changed node, O(log n)
- No re-balancing at lookup-only access

Rebalancing by Single Rotation

If root has AVL-balance < -1 or AVL-balance > 1 and
child balance on higher subtree side has same sign or is 0

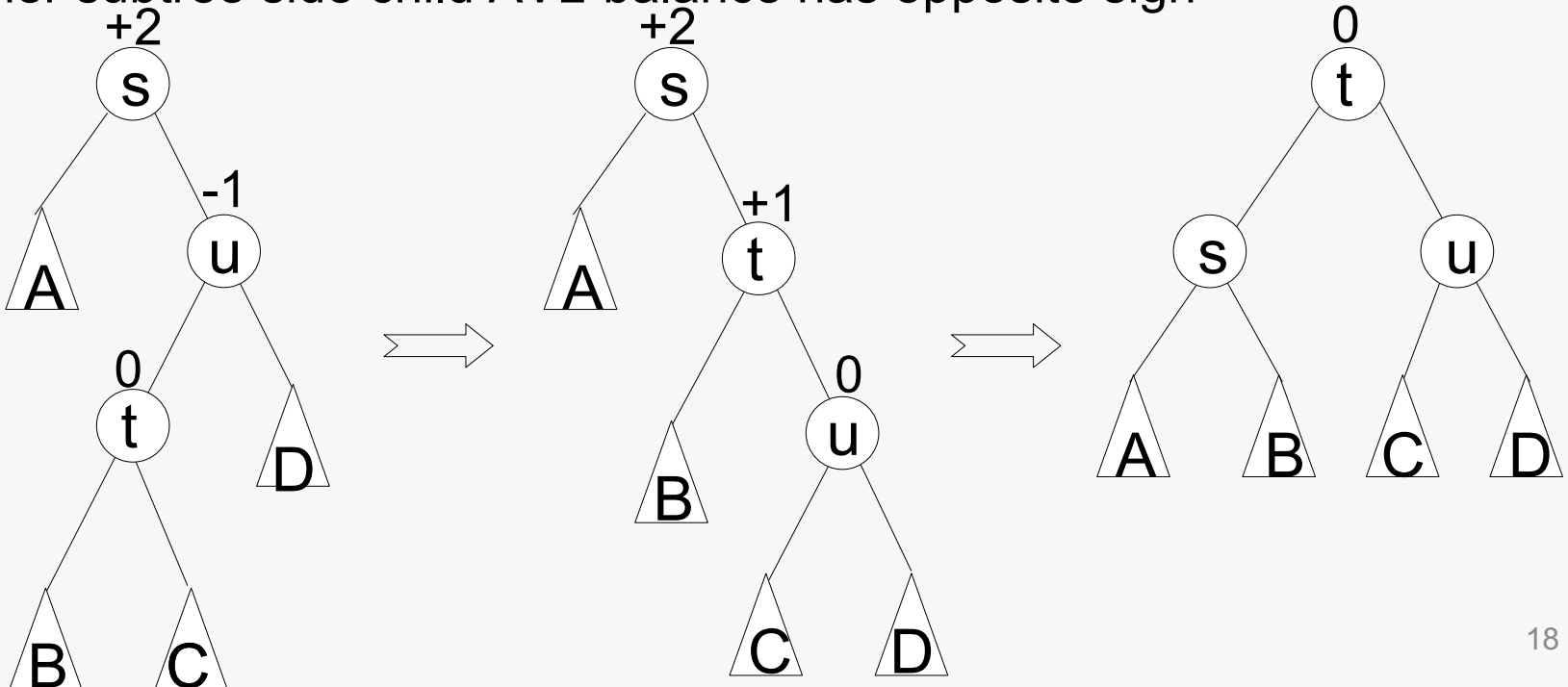


Rebalancing by Single Rotation Example

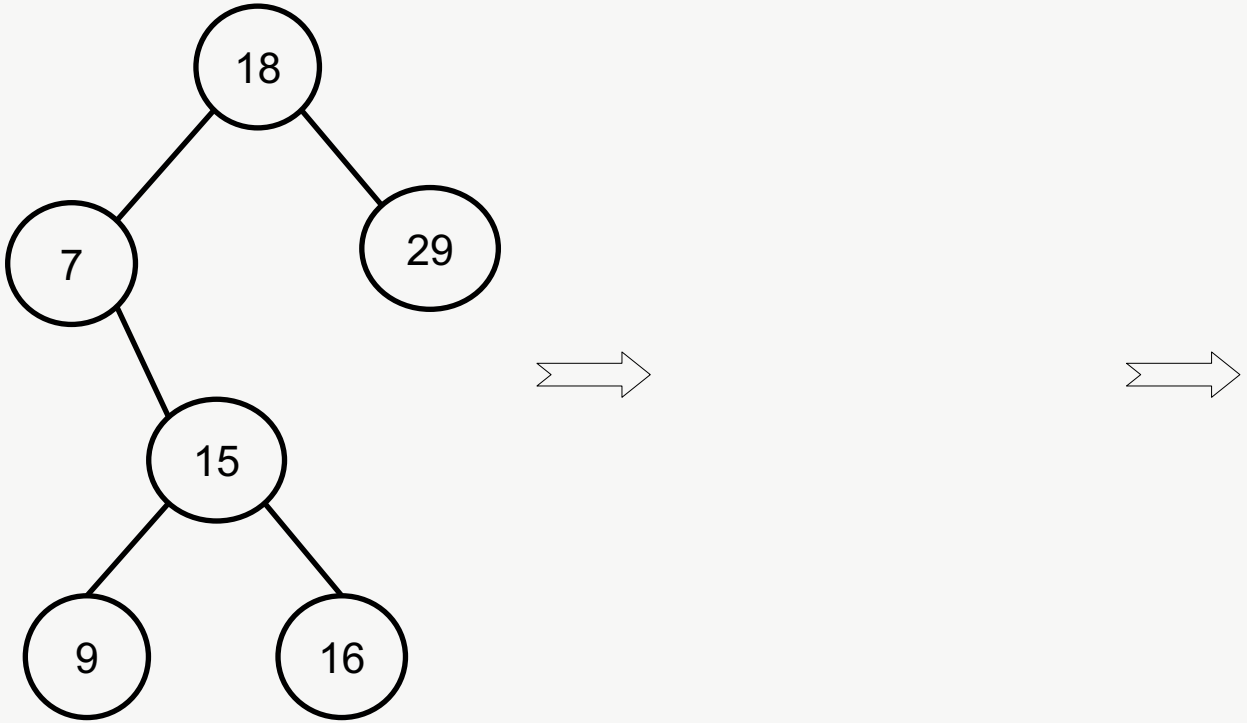


Rebalancing by Double Rotation

If root has AVL-balance < -1 or AVL-balance > 1 and higher subtree side child AVL-balance has opposite sign



Rebalancing by Double Rotation Example



AVL Tree Rebalancing Animation

- Inserting 12, 23, 200, 17, 15, 20, 18
- Then deleting 15, etc.

AVL Tree

Splay Trees

Splay Trees

- A splay tree rearranges itself after each access
- Moves the accessed node to root (splay operation)
- Frequently accessed keys get closer to the root
- No need for additional information to be stored

Splay step operations

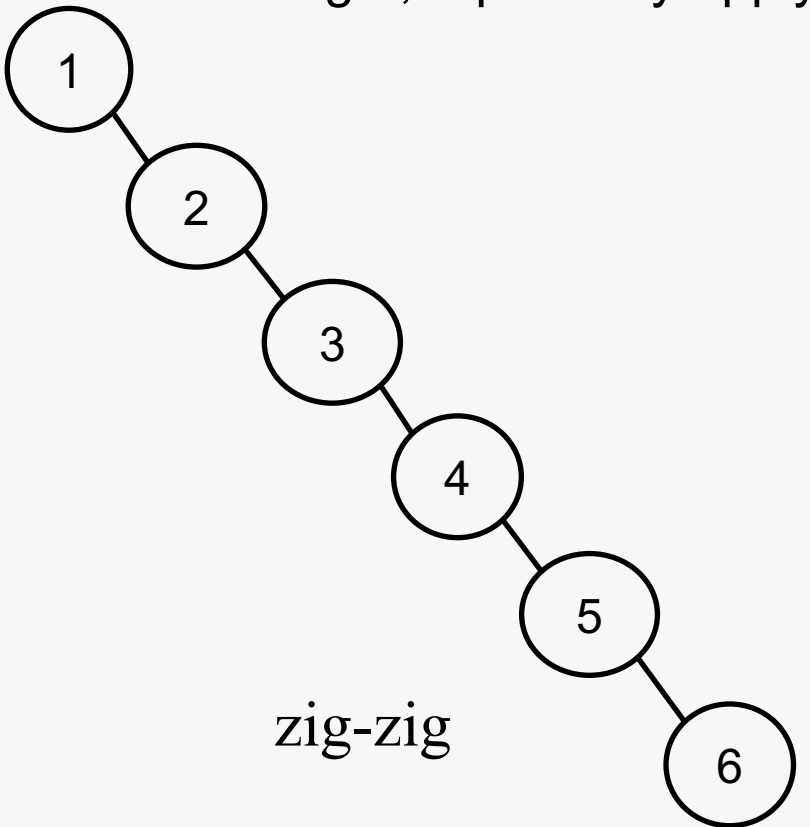
There are three cases (plus a symmetric one), and maximally two rotations per splay:

- Until accessed node is the root:
- If the accessed node's parent is the root (zig), rotate the accessed node up.
- If the accessed node is a left child and its parent is a left child as well (zig-zig), rotate the parent up and then rotate the accessed node up.*
- Otherwise (zig-zag), rotate to the accessed node up twice (standard double rotation).

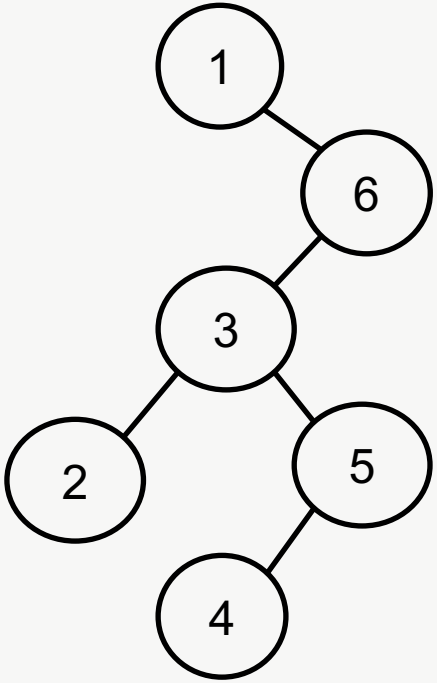
* Also applies to right child and right child

An example splay operation

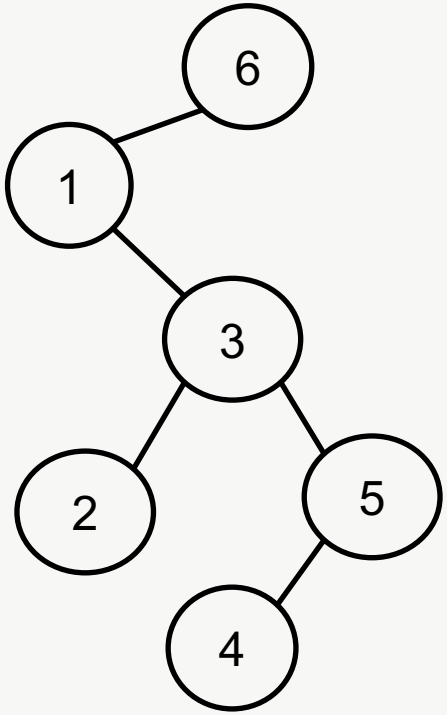
After accessing 6, repeatedly apply `splay(6)`:



Example splay (cntd.)



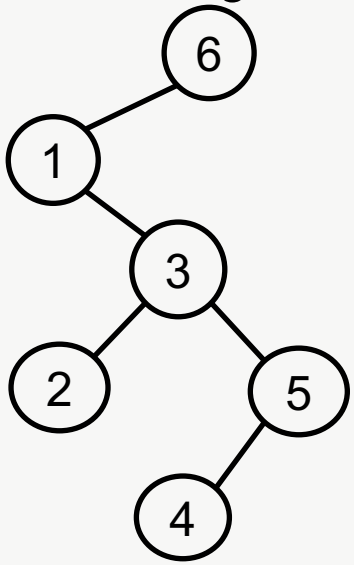
zig



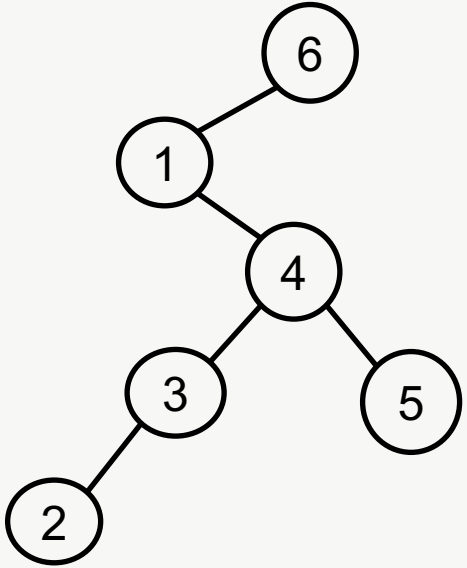
After the third `splay`, 6 is at the root of the tree.

Example splay zig zag

After accessing 4, repeatedly apply `splay(4)`:

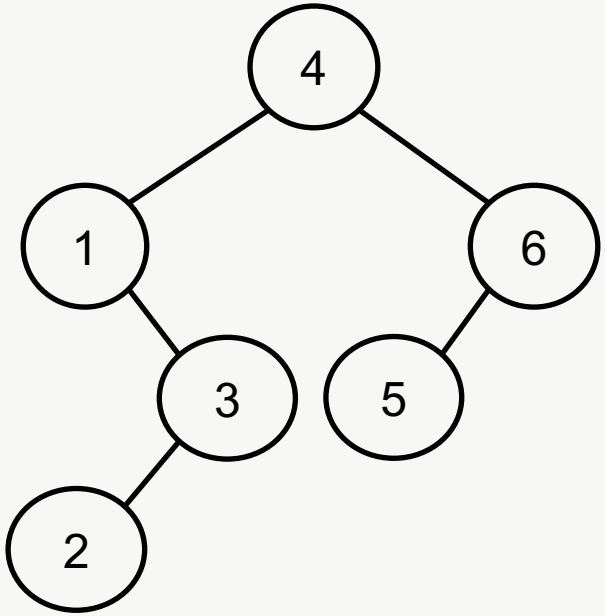


zig-zag



zig-zag

Example splay zig zag (cntd.)



After the second `splay`, 4 is at the root of the tree.

Note that 6 is still closer to the root than initially.

Splay trees are self-organising

There are three cases (plus a symmetric one), and maximally two rotations per splay:

- Until accessed node is the root:
- If the accessed node's parent is the root (zig), rotate the accessed node up.
- If the accessed node is a left child and its parent is a left child as well (zig-zig), rotate the parent up and then rotate the accessed node up.*
- Otherwise (zig-zag), rotate to the accessed node up twice (standard double rotation).

* Also applies to right child and right child

Splay Tree Animation

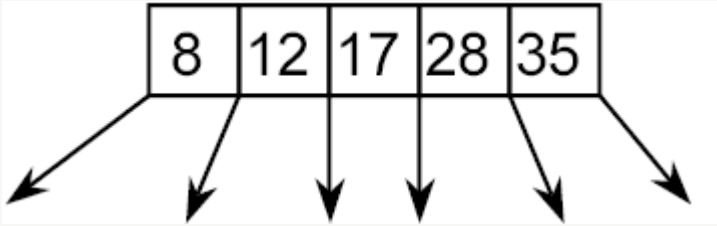
- Inserting 1, 2, 3, 4, 5, 6
- Then finding 1, 4, etc.
- Then deleting …

Splay tree

Multi-Way Search Trees

Multi-way search trees

Each node has *m* subtrees, interleaved with *m-1* keys stored in ascending order



All keys in subtrees to the left of key *k* are smaller than *k*

All keys in subtrees to the right of key *k* are greater

Large *m* is useful for external storage (hard disk)

Small *m* is better for internal storage (RAM)

... and binary search trees are multi-way trees with *m = 2*

B-Trees

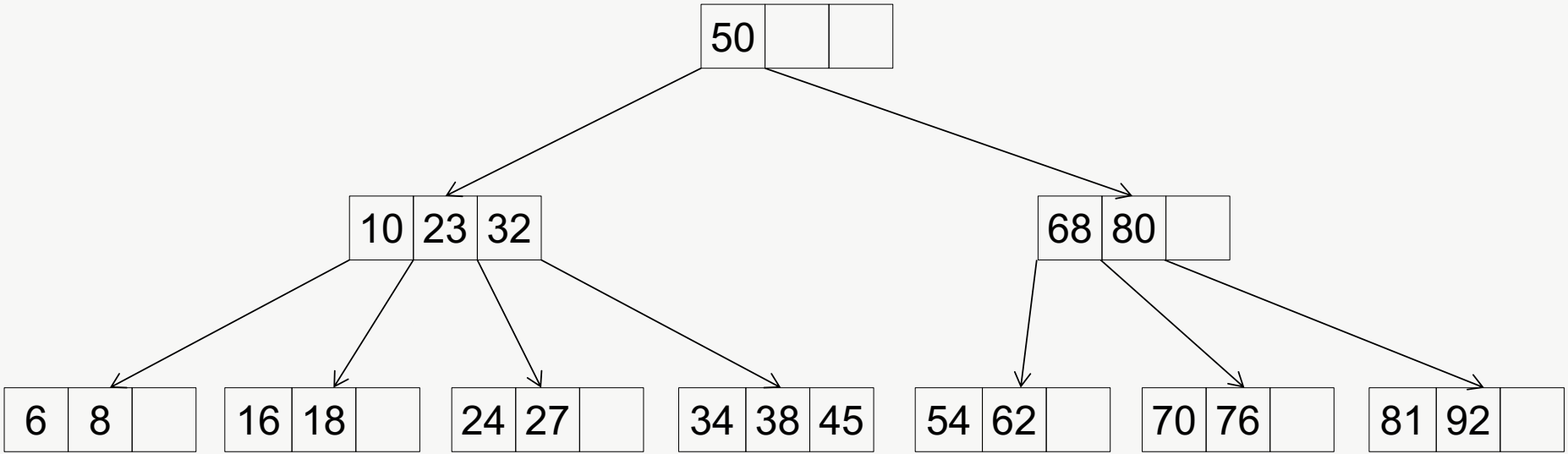
A B-tree of order *m* is a multi-way search tree for which:

- the root has at least one key
- every non-root node has *k* subtrees and *k-1* keys with $m/2 + 1 <= k <= m$
- leaves have all subtrees empty
- non-leaf nodes have non-empty subtrees
- all leaves are at the same level

The last condition guarantees that B-trees are approximately balanced.

For hard disk storage, one node is one HD block.

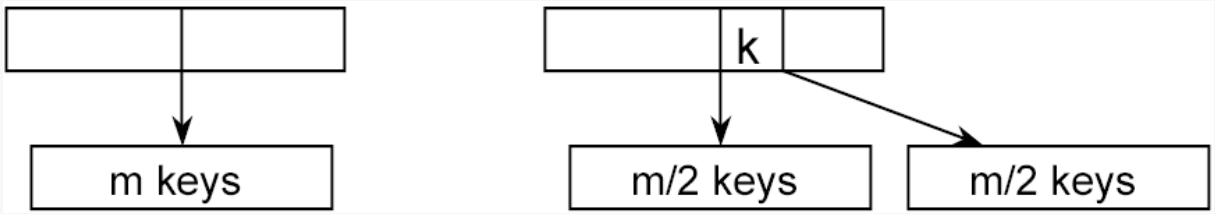
A B-tree of order 4 and height 3



Insertion into a B-tree

Always insert new keys into a leaf

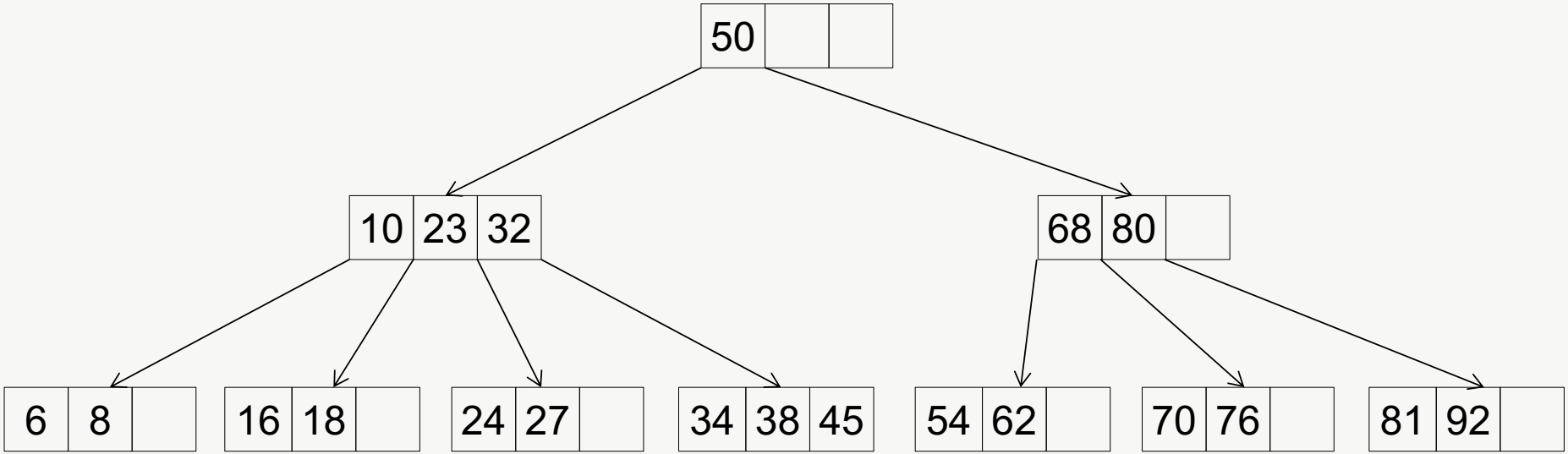
If the node overflows, split it:



- where k is the middle key of the overflowed node
- if the parent node overflows, repeat
- it the root overflows, create a new root on top and split

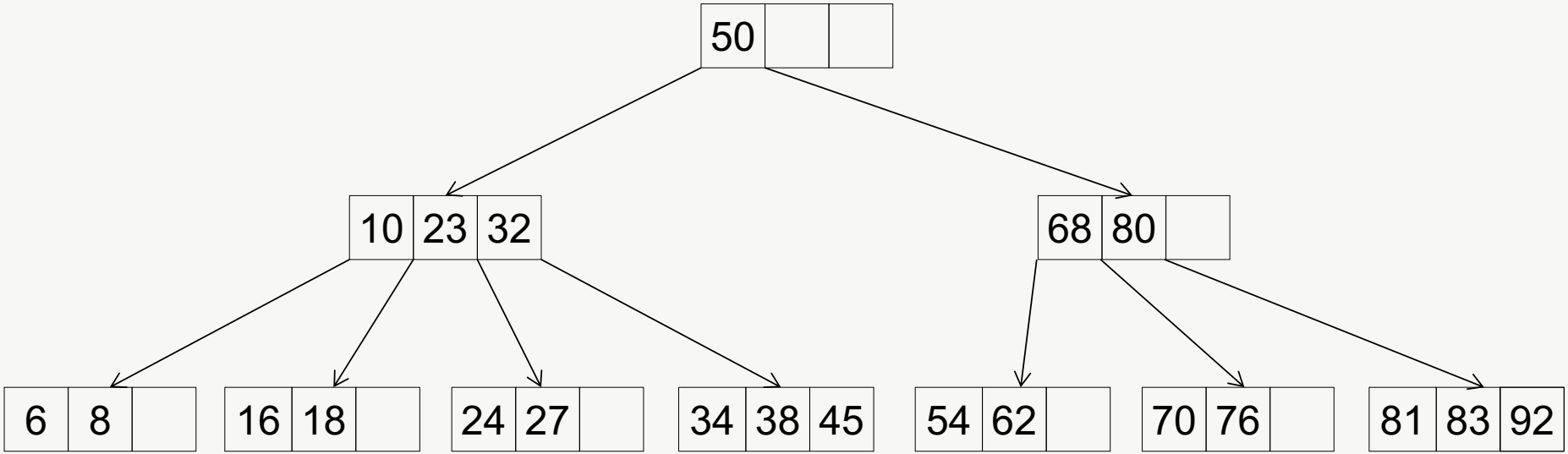
Insertion into a B-tree Example

Insert 83 then 90



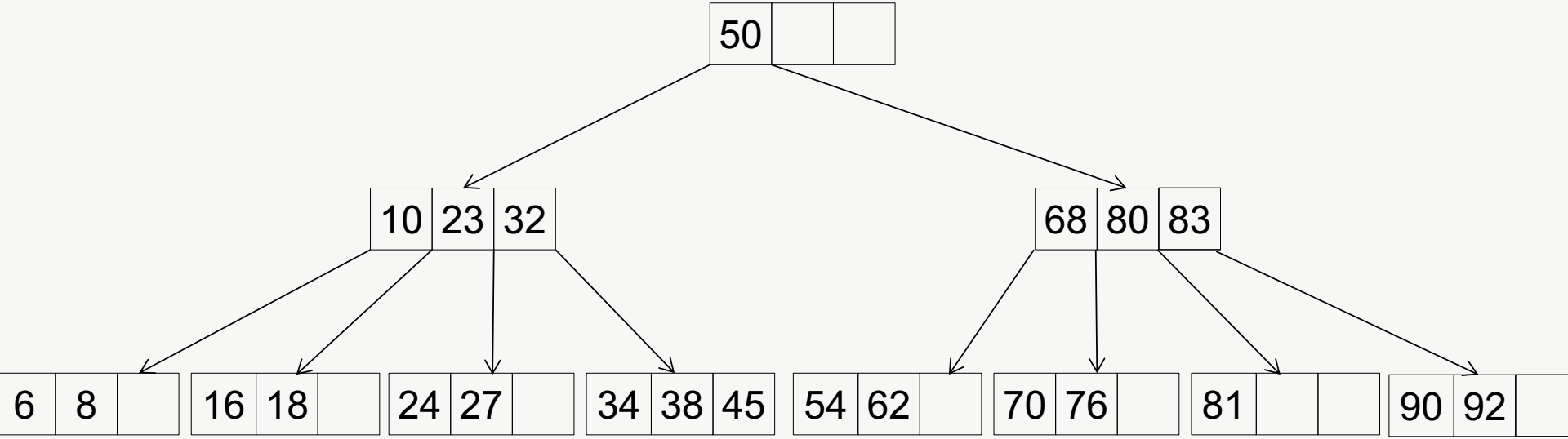
Insertion into a B-tree Example (cntd.)

Insert 83 (then 90)



Insertion into a B-tree Example (cntd.)

Insert 90



Deletion from a B-tree

- If the key to be deleted is in a non-leaf node, swap it with a neighbouring key in a leaf and delete it
- If deletion leaves a node half empty, redistribute keys with a sibling node
- If sibling node is only half full, combine with it (reverse of splitting)
- If the root node ends up with no keys and one pointer, its child becomes the new root

B-Tree Animation

- Inserting …
- Then finding ...
- Then deleting …

B-Tree

Reading

- Weiss: Sections 18.4, 18.8, 21.1-21.3, 21.5-21.7

- Drozdek: Sections 6.7, 6.8, and 7.1.1

Next session: Graphs

Drozdek: Sections 8.1, 8.2 and 8.3 OR Weiss: Sections 13.1-13.4

City, University of London
Northampton Square
London
EC1V 0HB
United Kingdom

T: +44 (0)20 7040 8406
E: ug-compsci@city.ac.uk
www.city.ac.uk/department-computer-science