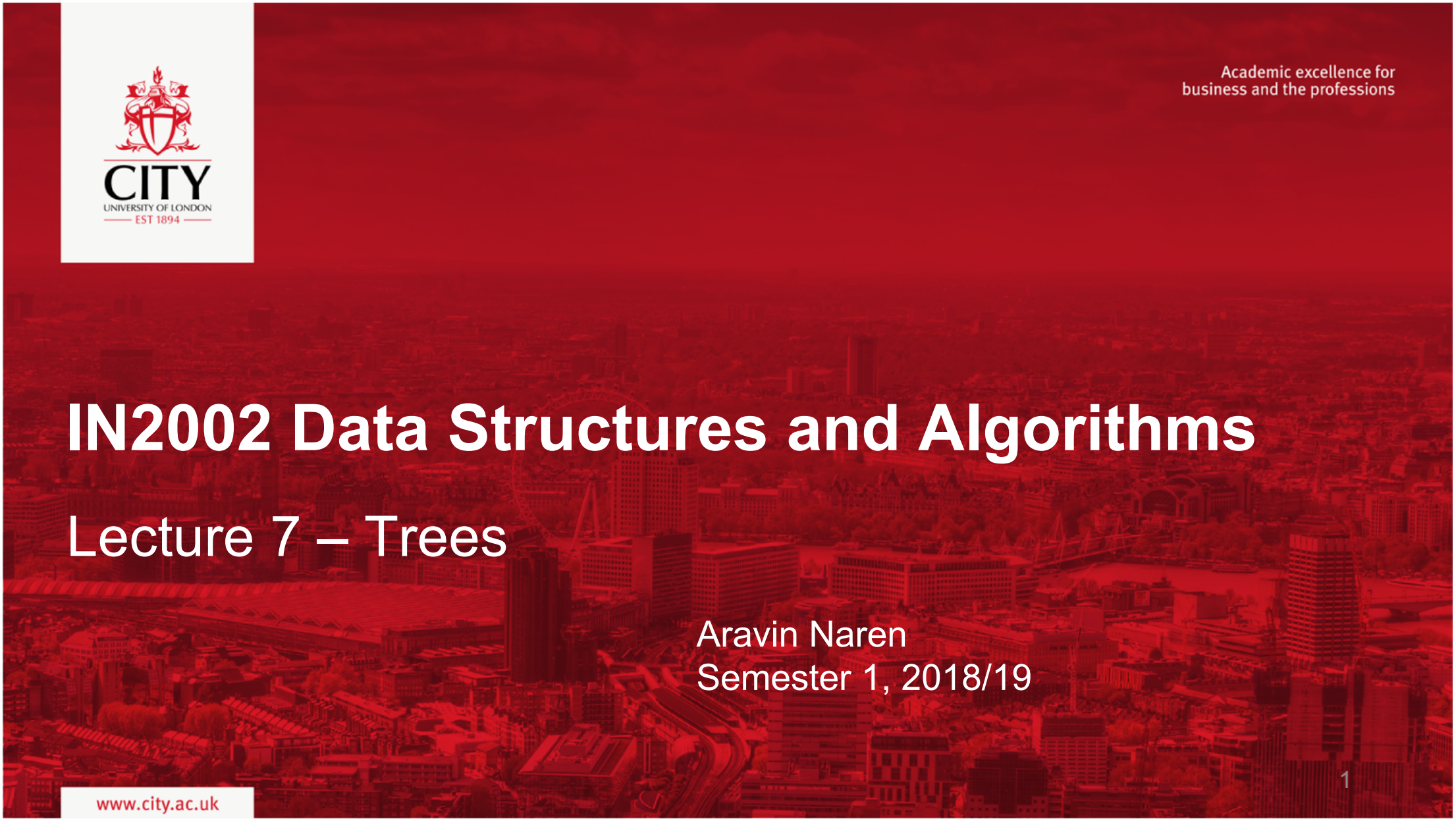

CITY
UNIVERSITY OF LONDON
EST 1894
Academic excellence for
business and the professions
IN2002 Data Structures and Algorithms
Lecture 7 – Trees
Aravin Naren
Semester 1, 2018/19
www.city.ac.uk
1

Learning Objectives

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

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

Trees*

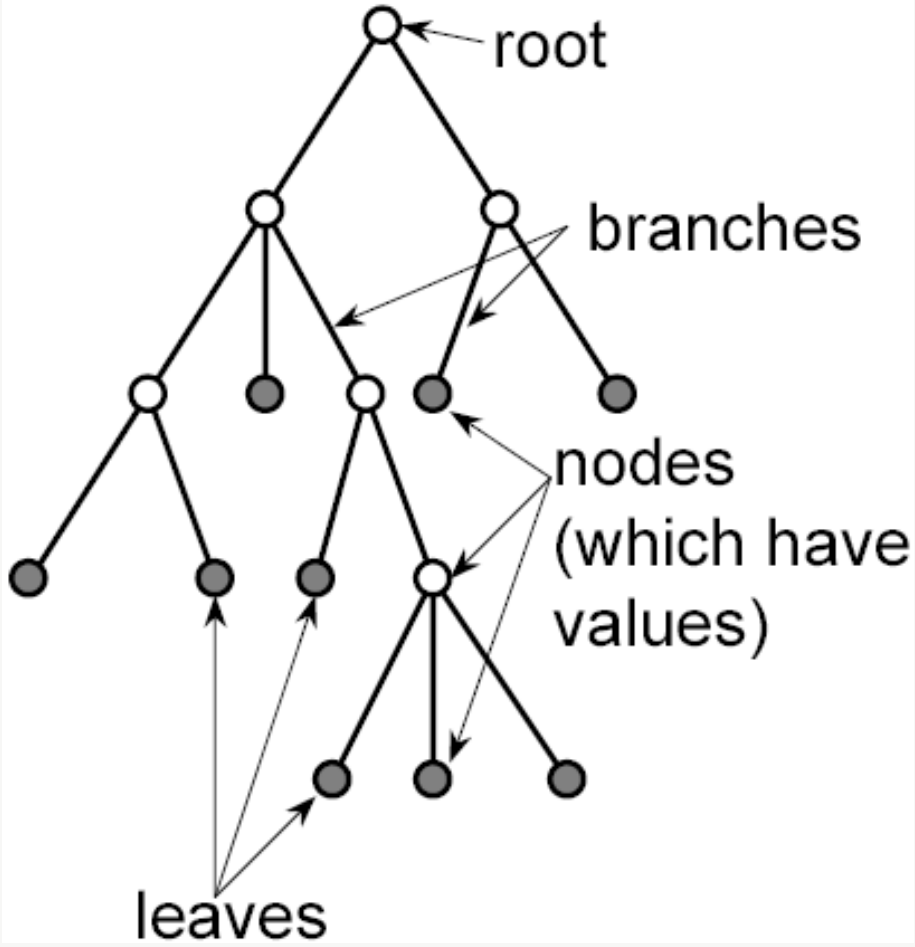
* Seen in this module lecture 3

Trees

- Trees are useful for hierarchical systems:
  - organisations
  - geographical regions
  - classes
  - language processing
- Artificial hierarchies can be created (e.g., heaps)
- Because of branching, trees can hold many keys without having great height
- Recursive definition is often used

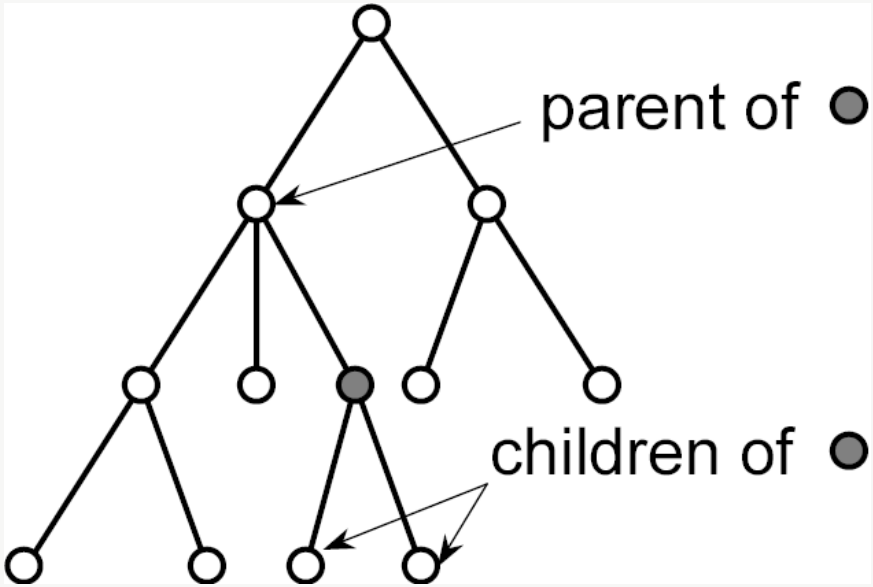
Tree terminology (reminder)

- **Trees** are composed of nodes and branches
- **Nodes** contain a label (data) and branches to other nodes
- **Root** is the top node
- **Leaves** are nodes that have no branches deriving from them



More tree terminology (reminder)

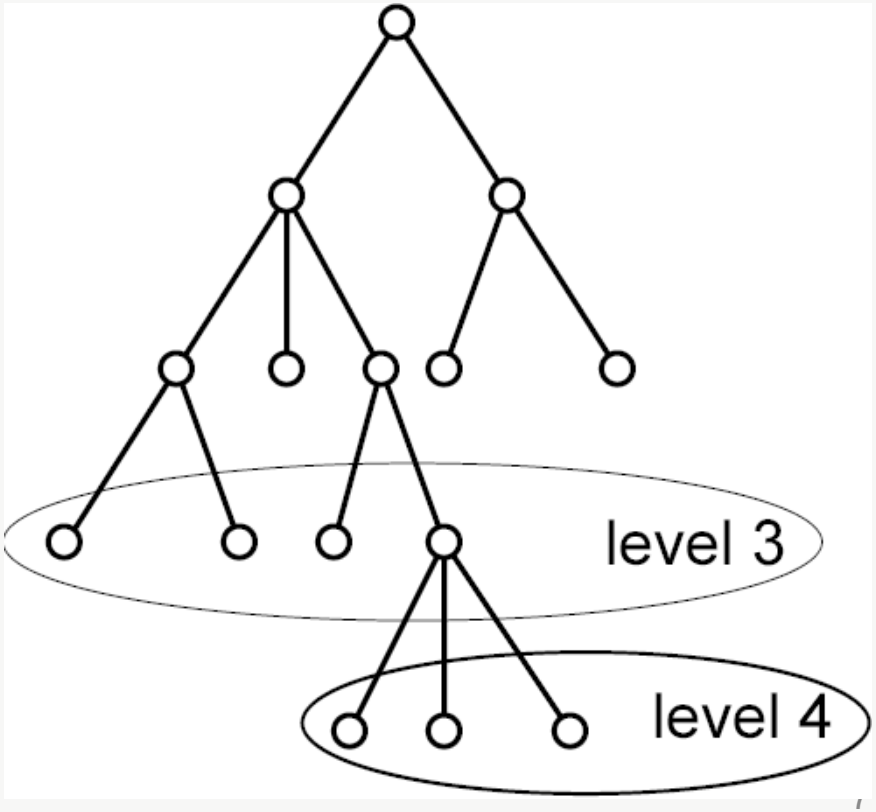
- The nodes n2,n3 at the end of a node n1's branches are its **children**, and n1 is the **parent** of n2,n3.
- The root is thus a node with no parents.
- The leaves are nodes with no children.



Tree properties (reminder)

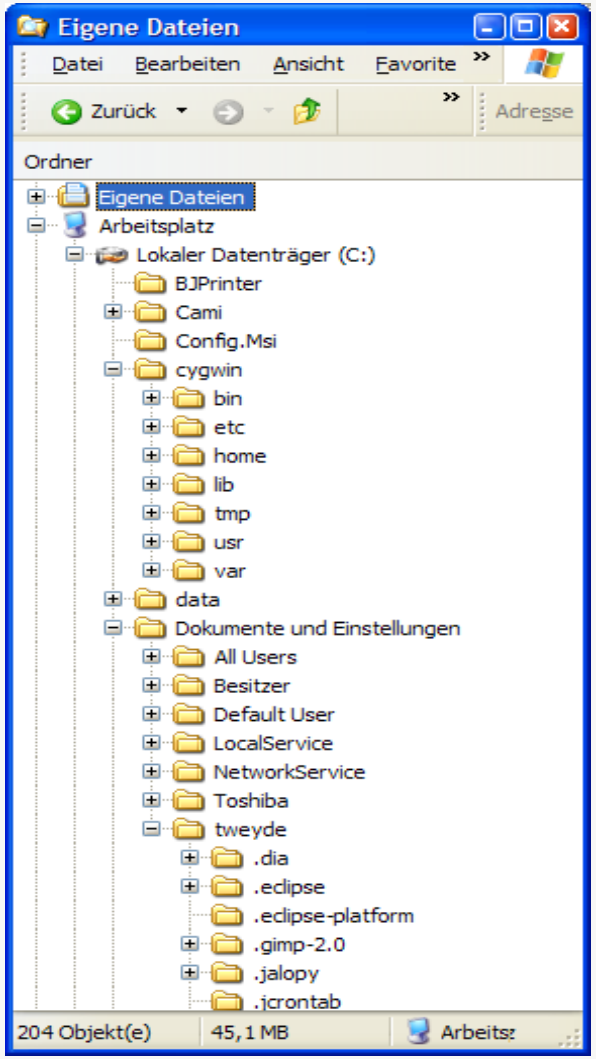
- The tree's **height** is the number of edges on the longest downward path between the root and a leaf
- The **depth** of a node is the number of edges from the node to the tree's root node. It refers to the distance to the root
- A node's **level** is its **depth + 1**
- A tree's **size** is the number of nodes it contains

Note that trees have no loops



A well-known example

Common file systems are organised as trees.



Representation in Trees

Representation in trees

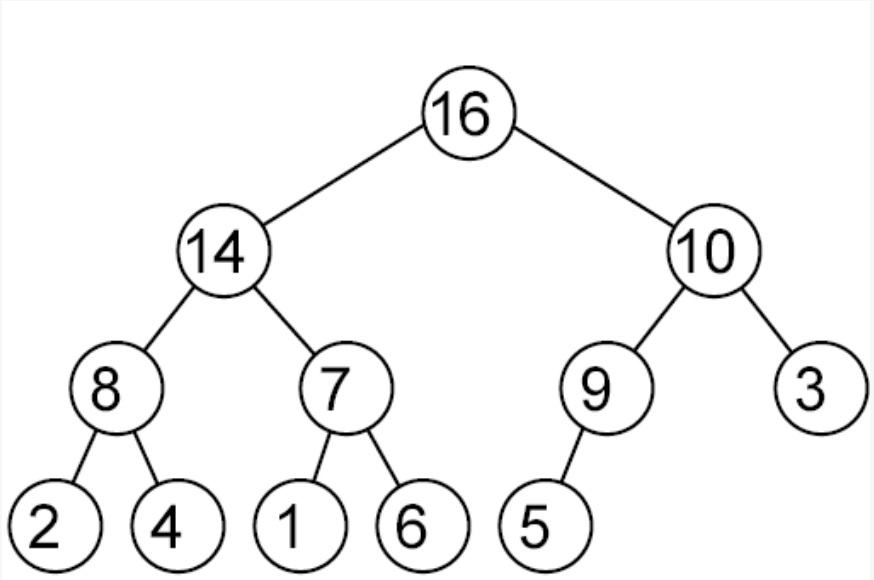
Nodes represent things.

Branches represent links between the nodes,
i.e., links between what nodes represent.

All this information is implicit in the tree.

A “pre-sorting” tree

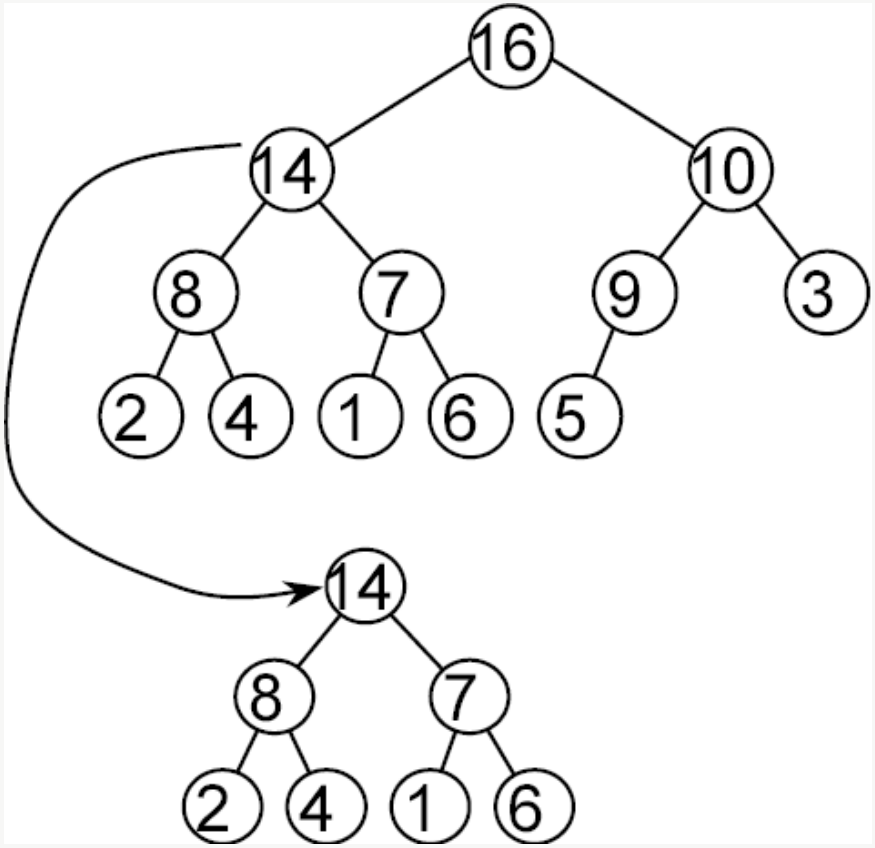
- This tree represents an abstract hierarchy: “greater than”
- Nodes represent numbers
- Branches represent ordinality (“greater than”)
- Does this tree look familiar?



About tree algorithms

- Any node in the tree can be considered the root of a subtree
- This allows a recursive definition:

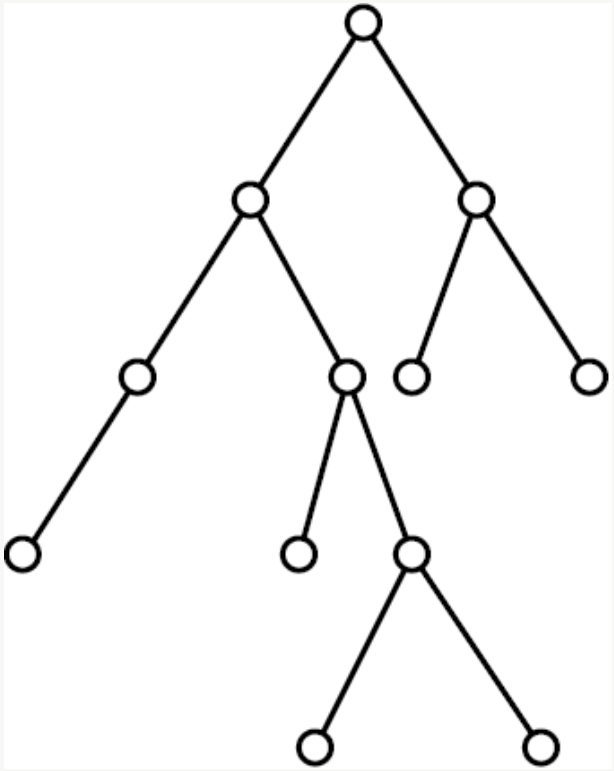
A tree is either empty or a tree-node with a label (data) and branches to other trees



Binary Trees

Building a binary tree

- **Binary tree:**
  - each node has at most two branches



Binary trees in Java

```
public class TreeNode {
    public int key;
    public TreeNode left, right;

    public TreeNode(int k, TreeNode l,TreeNode r) {
       key = k; left = l; right = r;
    }

    public TreeNode(int k) {
       this(k, null, null);
    }
}
```

Building a binary tree

```
TreeNode t =
     new TreeNode(5,
         new TreeNode(4),
         new TreeNode(6, new TreeNode(7), null));
```

Recursion in trees

Tree algorithms can be defined recursively, by evaluating the current tree and the trees branching from it

```
public int size(TreeNode t) {
    if (t == null)
        return 0;
    return size(t.left) + 1 + size(t.right);
}
public int height(TreeNode t) {
    if (t == null)
        return 0;
    return 1 + Math.max(height(t.left), height(t.right));
}
```

Tree Traversals

Tree traversals

Many applications involve visiting the nodes in a certain order.

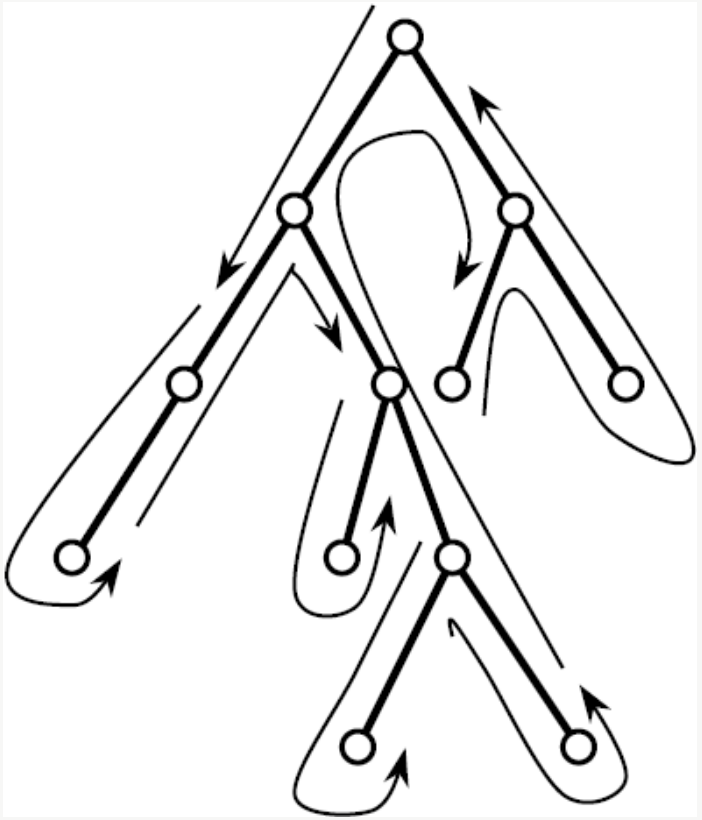
Let's say we have procedure
`void visit(TreeNode p);`
which, for example, prints the node's label.
This can be done in different sequences.

For simplicity, we will define the traversals for binary trees.

Depth First Traversals

Depth First Traversal

- Nodes are visited going as far as possible through the first branch
- Once a leaf is found, the algorithm backtracks until it can find an unvisited branch



Recursive definition

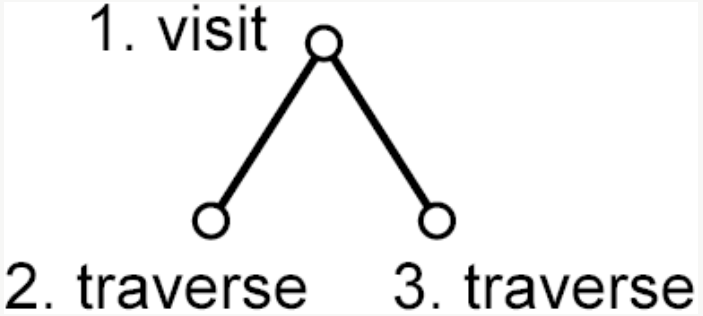
- When a depth-first traversal is in a node, it must:
  - **visit** that node, (V)
  - traverse the **left** subtree (L)
  - traverse the **right** subtree (R)

- However, this could be done in any order:
  - if visit is done first (VLR), then it is a **preorder** traversal
  - if visit is done in between (LVR), then it is **inorder** traversal
  - if visit is done last (LRV), then it is **postorder** traversal

Preorder traversal

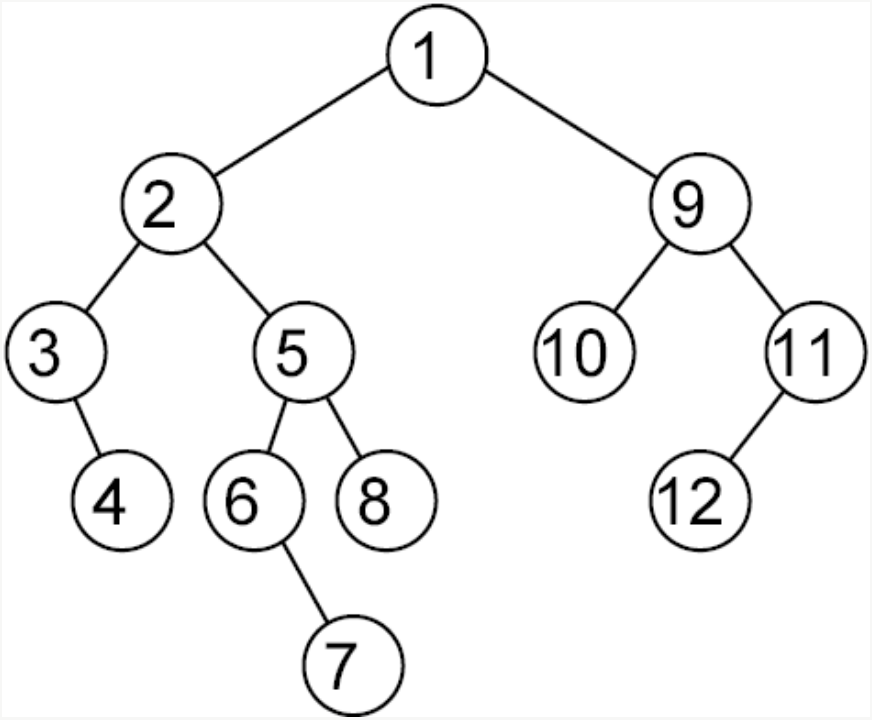
The sequence is:

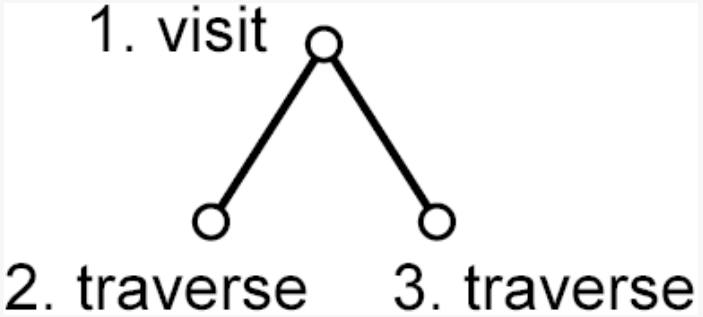
- visit the current node first
- traverse the left subtree
- traverse the right subtree



<u>Function preorder (node):</u>
IF node ≠ null THEN
    visit node
    preorder (left child of node)
    preorder (right child of node)

… preorder traversal





(stack ADT)

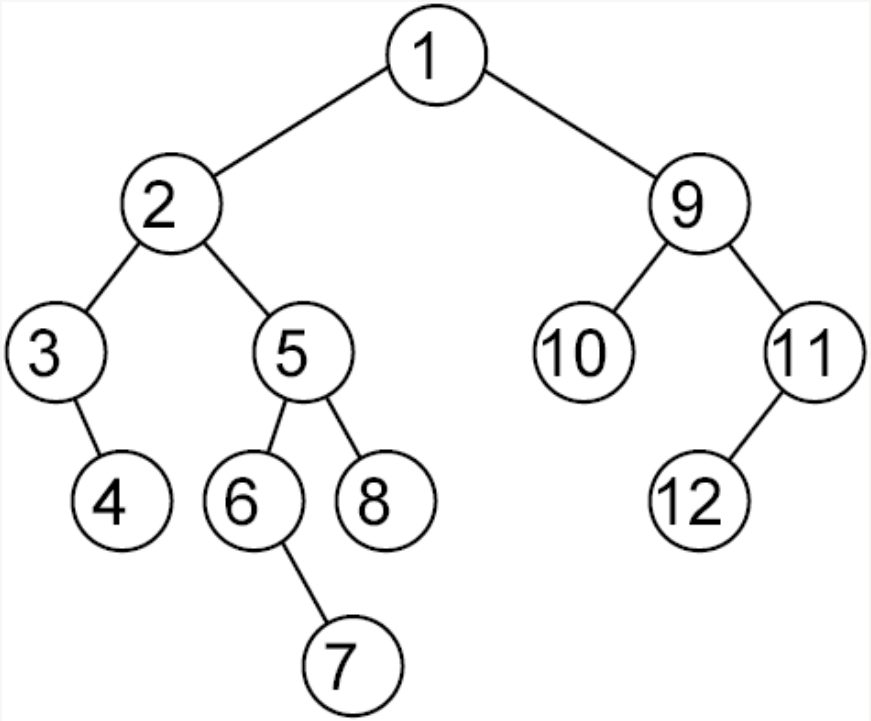
```
public interface Stack<T> {
    // Is the stack empty?
    boolean isEmpty();

    // Push one element onto the stack
    void push(T elt);

    // Remove and return the most recently pushed
    // element not already popped.
    // Precondition: ! isEmpty()
    T pop();
}
```

... iterative preorder traversal

The trick is to realise that the preorder actually keeps a stack



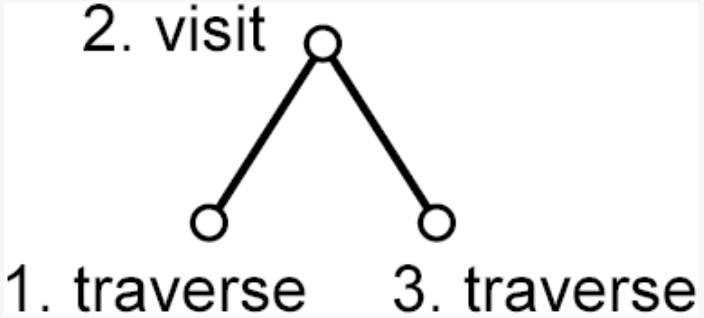
... iterative preorder traversal

```
void iterativePreorder(TreeNode p) {
    if (p != null){
        Stack<TreeNode> stack = new StackImpl<TreeNode>();
        stack.push(p); // add root
        do {
            p = stack.pop();
            visit(p);  // visit after popping
            if (p.right != null)
                stack.push(p.right); // push right
            if (p.left != null)
                stack.push(p.left); // push left
        } while (!stack.isEmpty()); // ready for next level
    }
}
```

Inorder traversal

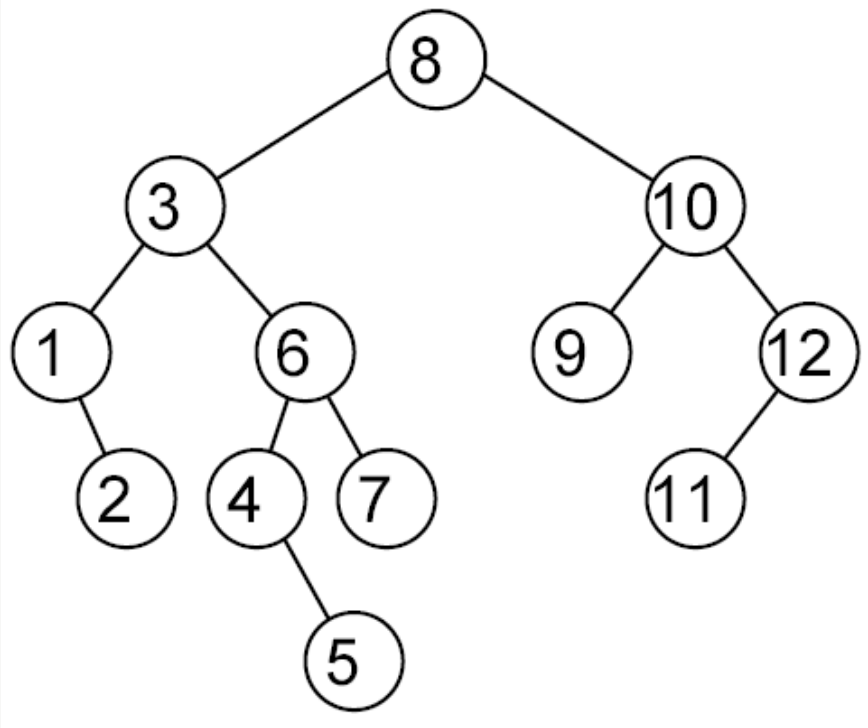
The sequence is:

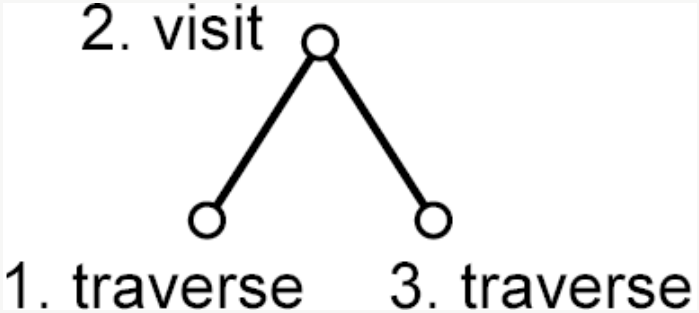
- traverse the left subtree
- visit the current node
- traverse the right subtree



<u>Function inorder (node):</u>
IF node ≠ null THEN
inorder (left child of node)
visit node
inorder (right child of node)

… inorder traversal

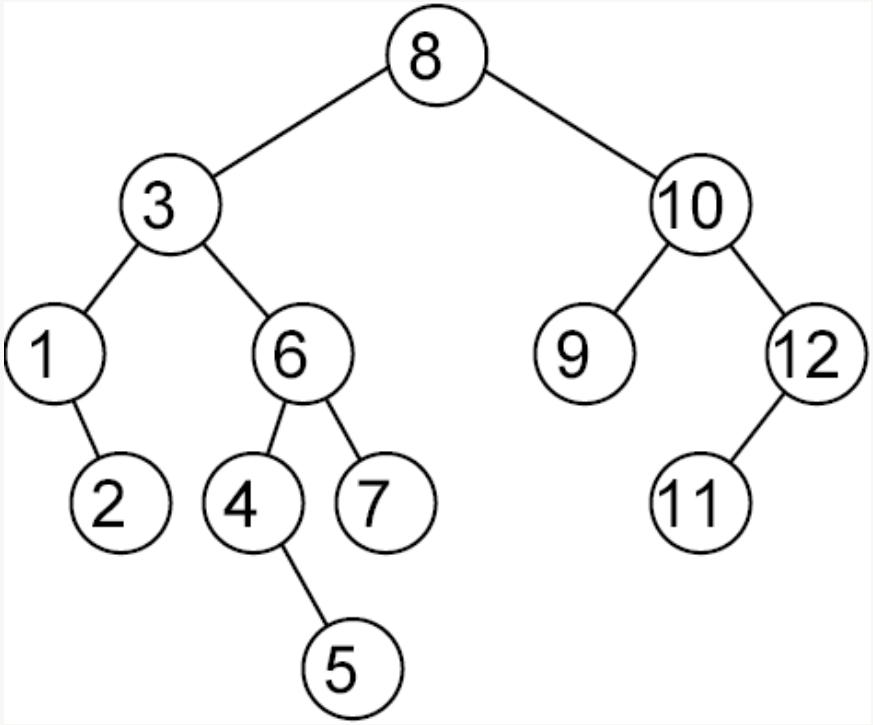




... iterative inorder traversal

```
void iterativeInorder(TreeNode p) {
    Stack<TreeNode> stack = new StackImpl<TreeNode>();
    for(;;) {
        while (p != null) { // go as far as
            stack.push(p);   // possible down
            p = p.left;      // on the left side
        }
        if (stack.isEmpty())
            break;    // stop when stack is empty
        p = stack.pop();    // leftmost unvisited node
        visit(p);    // visit
        p = p.right;    // continue on right side
    }
}
```

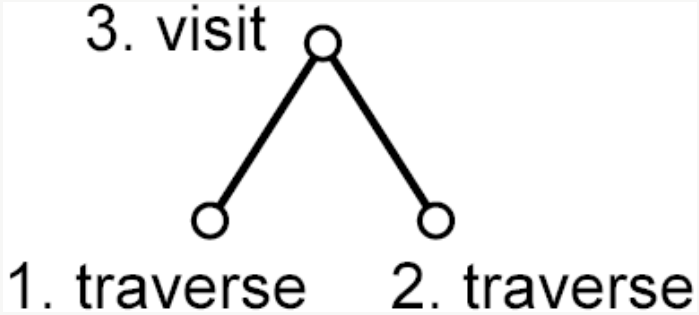
... iterative inorder traversal



Postorder traversal

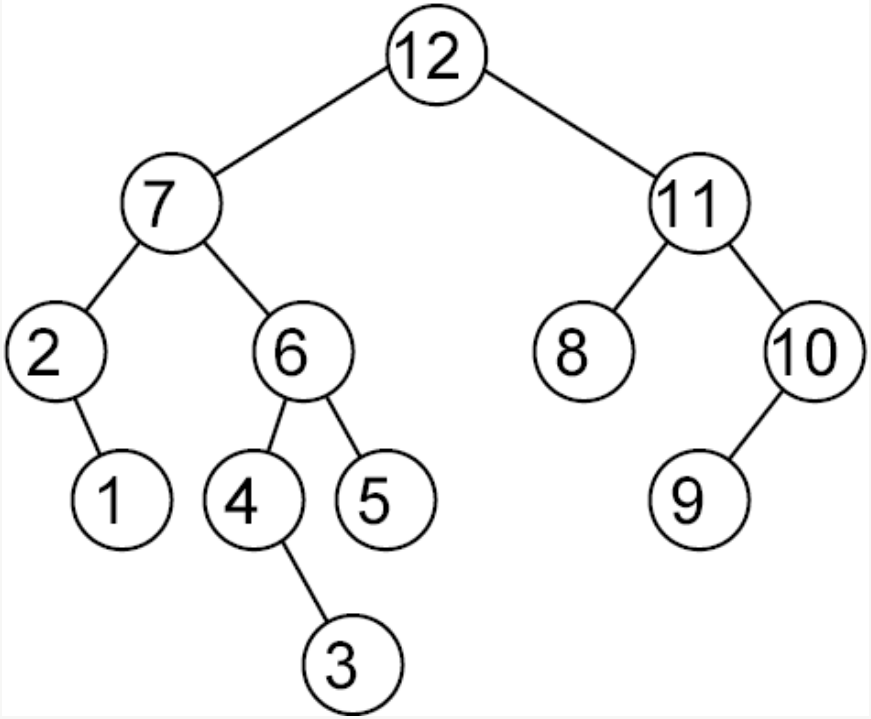
The sequence is:

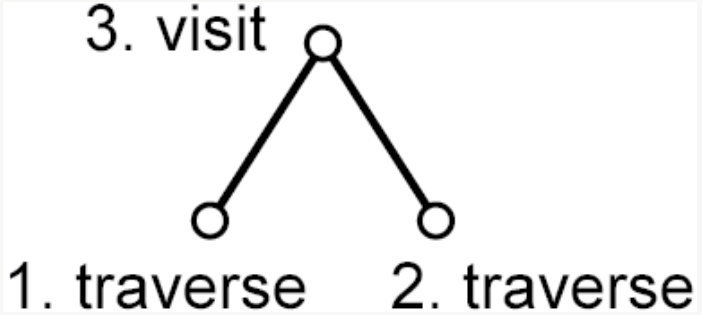
- traverse the left subtree
- traverse the right subtree
- visit the node



<u>Function postorder (node):</u>
IF node ≠ null THEN
postorder (left child of node)
postorder (right child of node)
visit node

… postorder traversal

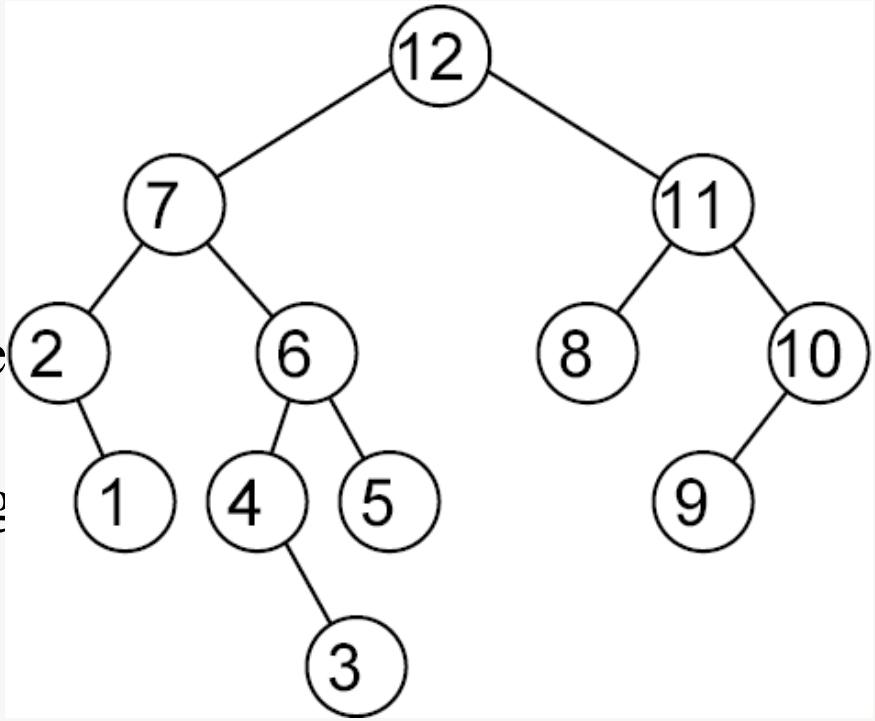




Iterative postorder traversal

We only store the parent nodes, not the current ones.

We need to store with the nodes information on whether we are traversing the left or right branch.



Analysis

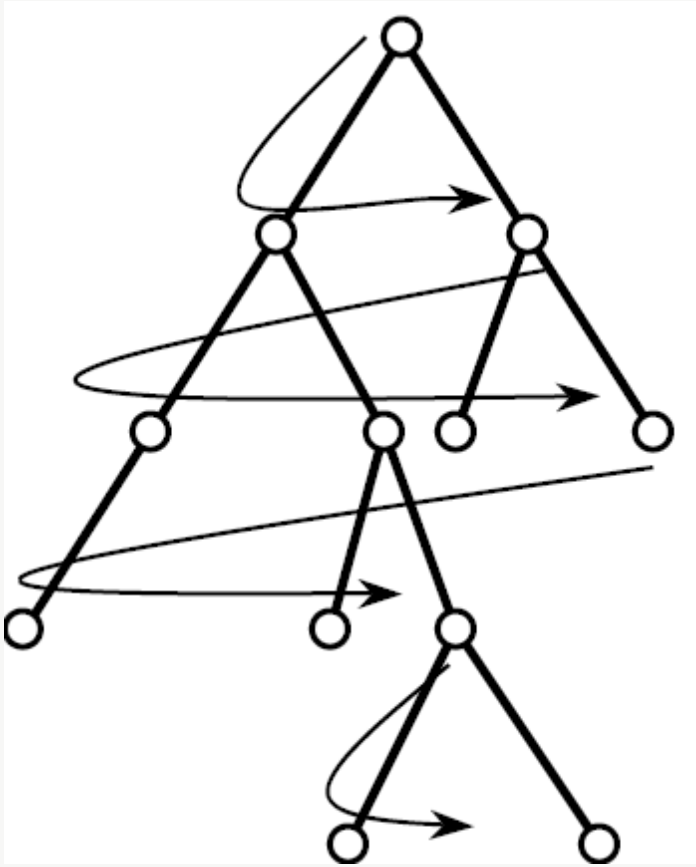
- In all cases the memory complexity equals the depth of the tree.
- In balanced binary trees the depth is O(log n), and so is the memory complexity.
- This memory complexity occurs in the recursion stack and in the explicit stack. The recursive algorithm is simpler than the iterative one.
- The choice of pre/in/post-order depends on how interested you are in the nodes that are not leaves

Breadth First Traversals

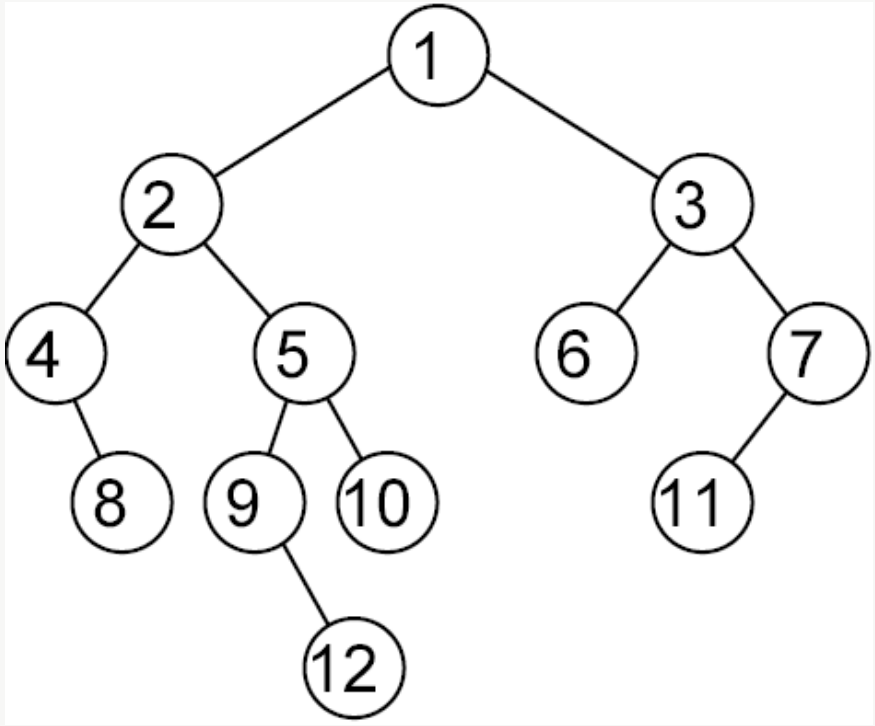
Breadth First Traversal

If you are not so interested in leaves, you may want to visit the nodes closer to the root first (also called level-order). In this case, you traverse the tree by levels:

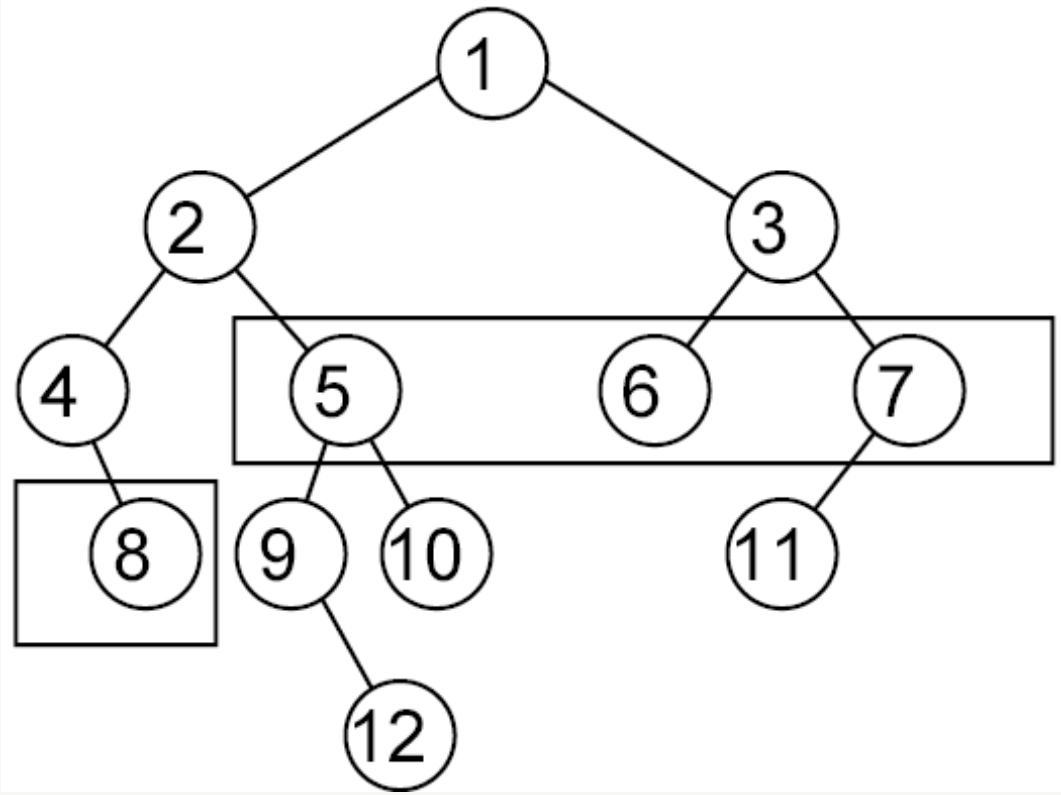
- Use a queue of nodes
- The queue contains initially the root only
- Until the queue is empty,
  - Remove and visit the first node in the queue
  - Add its children at the back of the queue



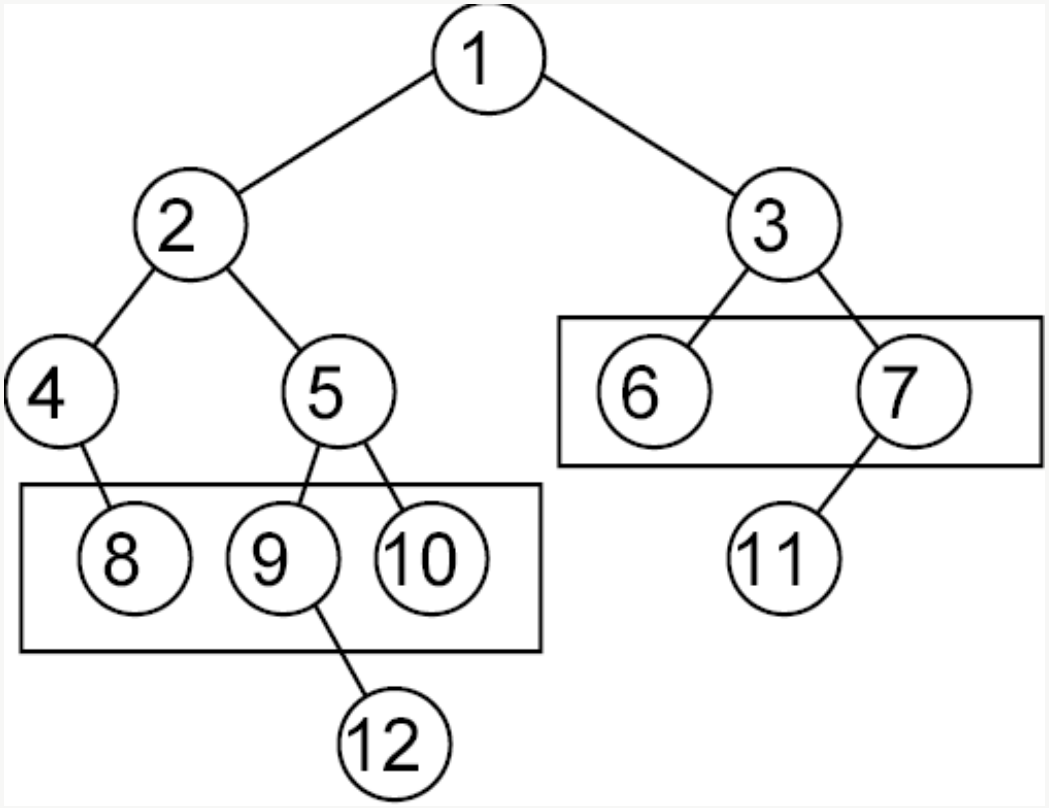
Stepping through the breadth-first traversal



The queue after visiting node 4



The queue after visiting node 5



(queue ADT)

As with stacks, the Queue interface can be used directly:

```
public interface Queue<T> {
   boolean isEmpty();
   void enqueue(T elt);
   T dequeue();
}
```

Breadth-first traversal – Java

```
void breadthFirst(TreeNode p) {
    if (p != null){
        Queue<TreeNode> queue = new QueueImpl<TreeNode>();
        queue.enqueue(p); // add root
        while (!queue.isEmpty()) {
            p = queue.dequeue(); // remove
            visit(p); // visit
            if (p.left != null) // add left child
                queue.enqueue(p.left);
            if (p.right != null) // add right child
                queue.enqueue(p.right);
        }
    }
}
```

Analysis of breadth-first

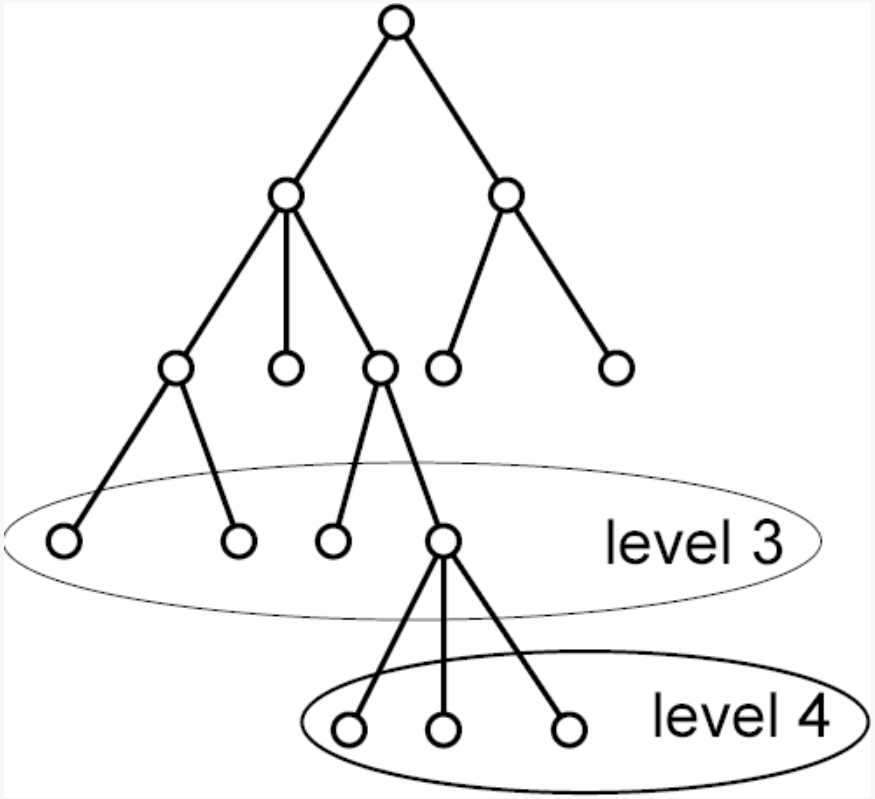
- The queue will contain at any one time the rest of the current level and part of the next level
- The queue will be at its largest at the level with the most nodes
- In a binary, perfectly balanced tree, this will occur at the lowest level. The size would be of O(n).

Binary Search Trees

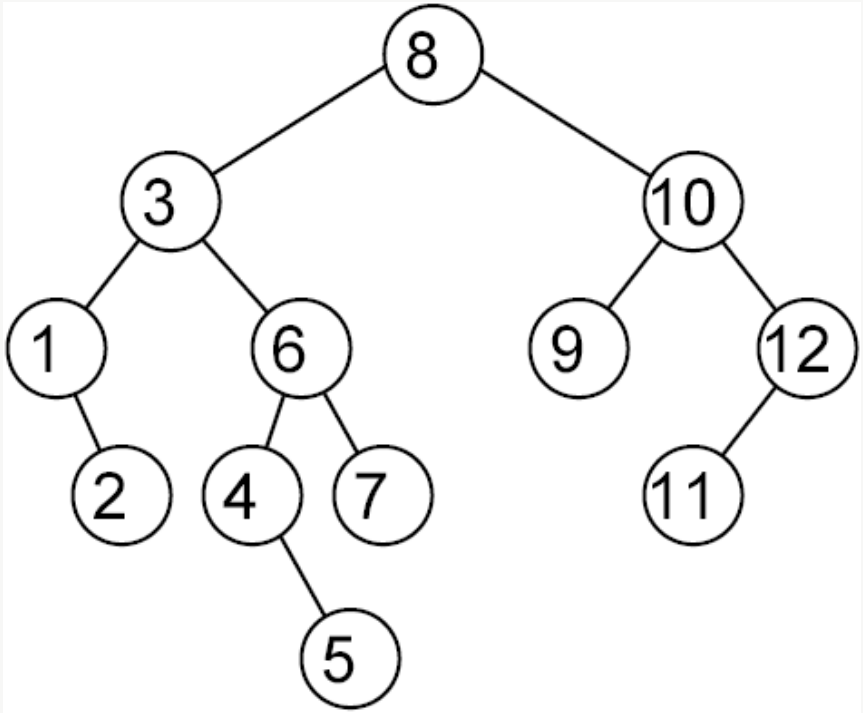
What if trees pre-sort (in some way)?

The search could concentrate in sections of the tree, gaining some time:
On average, O(log n) can be obtained

The worst cases can be kept close to O(log n) as well, so long as the tree is close to balanced



Binary Search Trees



e.g. look for 5:

5<8, go left

5>3, go right

5<6, go left

5>4, go right

5=5, found it!

What do branches represent?

Branches represent **ordering**:

- Any key in the left subtree is lower than the parent's key
- Any key in the right subtree is larger than the parent's key
- ... so no keys can be repeated

Inorder traversal produces an ordered sequence

Most operations visit only one **single path** down from the root.

If the tree is reasonably balanced, this takes O(log n)

Search trees as sets

```
public class Tree implements Set {
    private TreeNode root = null;

    // Is the element in the tree?
    public boolean search(int el) { .... }

    // Add the element to the tree
    public void insert(int el) { ... }

    // Remove the element from the tree
    public void delete(int el) { ... }
}
```

Searching a binary search tree

```
public boolean search(int el) {
    TreeNode p = root;
    while (p != null) {
        if (el == p.key)
            return true;
        if (el < p.key)
            p = p.left;
        else if (el > p.key)
            p = p.right;
    }
    return false;
}
```

Does this remind you of some other algorithm?

Inserting a key into a search tree

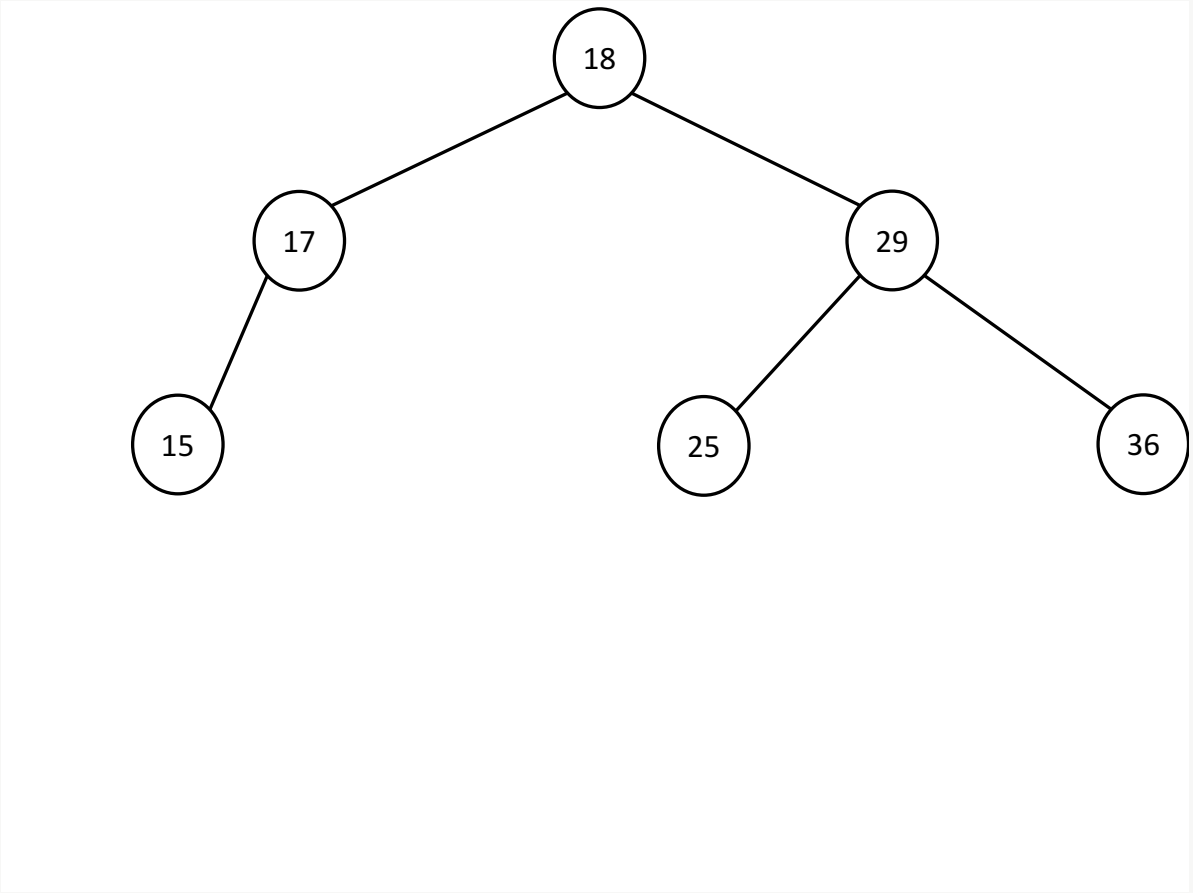
Search for the key

If the key is present, do nothing

Otherwise, replace the null pointer you found at the end of the search with a new node containing the key

(this requires a pointer to the parent node!)

Insertion example



Insert 16, 9, 1, 32

Insertion: Java code

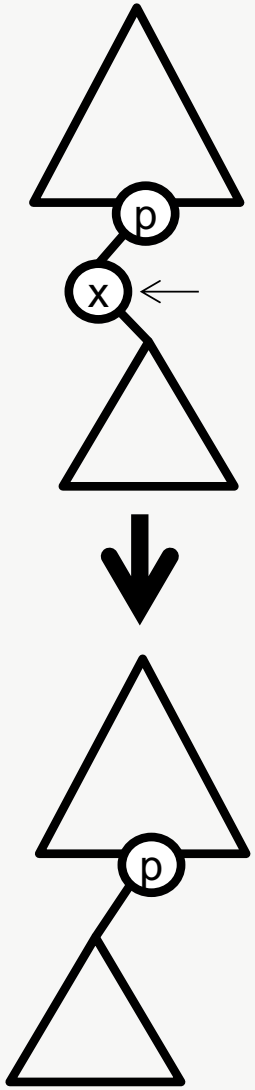
```
public void insert(int el) {
    if (root == null)  // the tree is empty
        root = new TreeNode(el);
    else {
        TreeNode prev, p = root;
        do {
            if (el == p.key) return;  // node present
            prev = p;    // remember parent
            if (el < p.key) p = p.left;
            else p = p.right;
        } while (p != null);  // until leaf found
        if (el < prev.key)  // determine left/right
            prev.left = new TreeNode(el);
        else
            prev.right = new TreeNode(el);
    }
}
```

Deletion

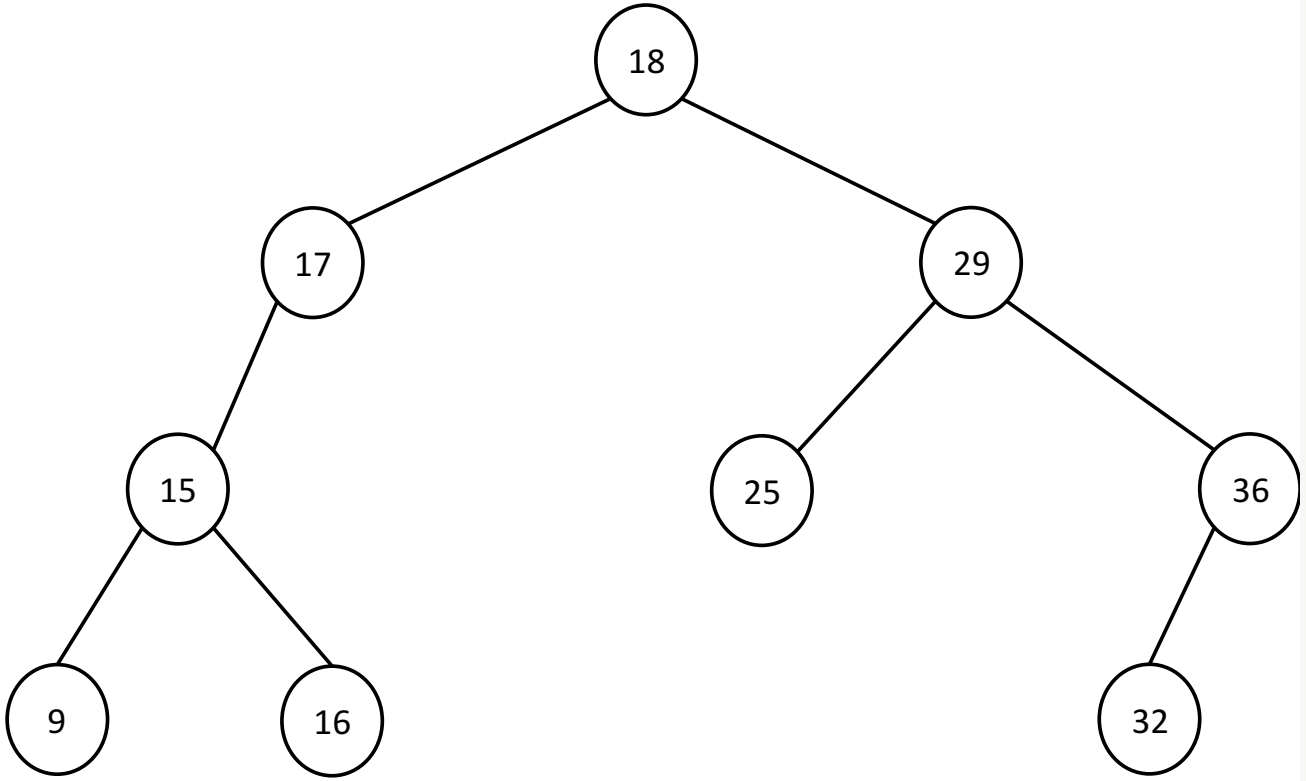
First find the node containing the key to be deleted.

If both its children are null, we just delete it from its parent.

If one child is null, we can simply delete the node and replace it with its non-null child.

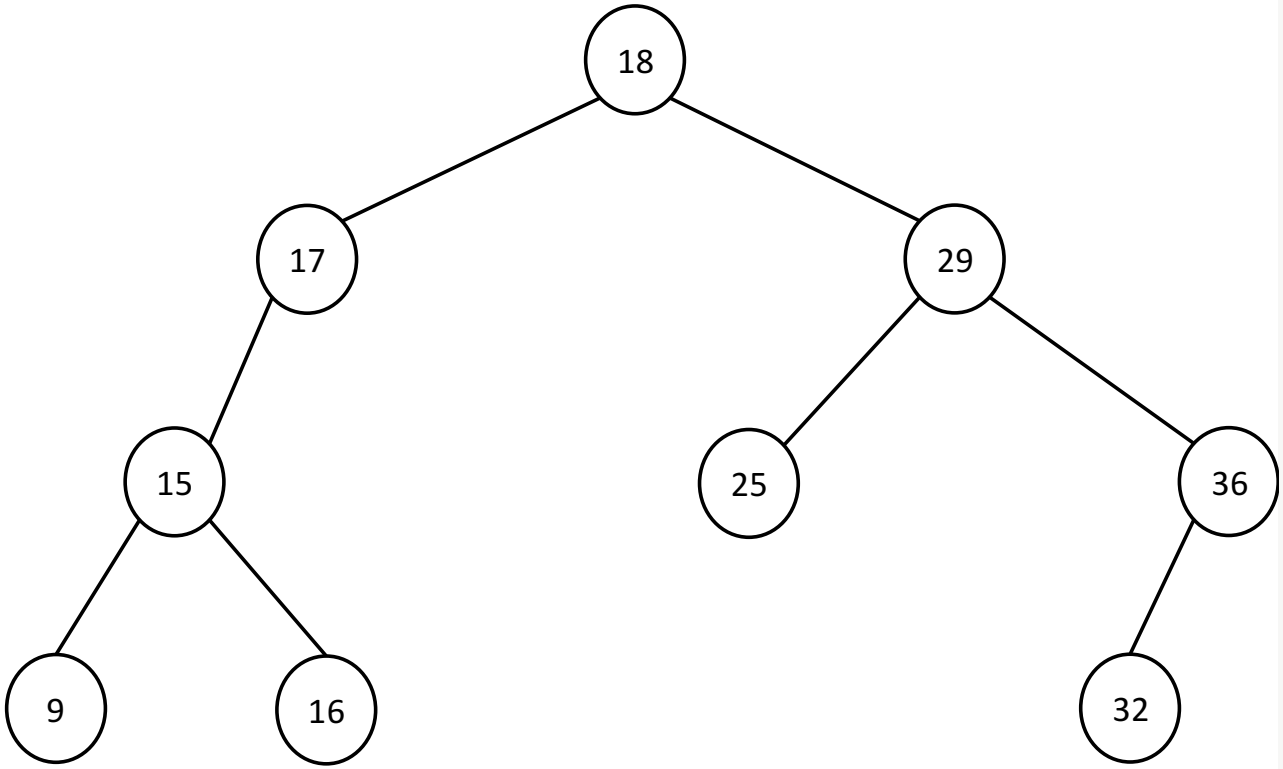


Deletion example



Delete 9 and 32

Deletion example

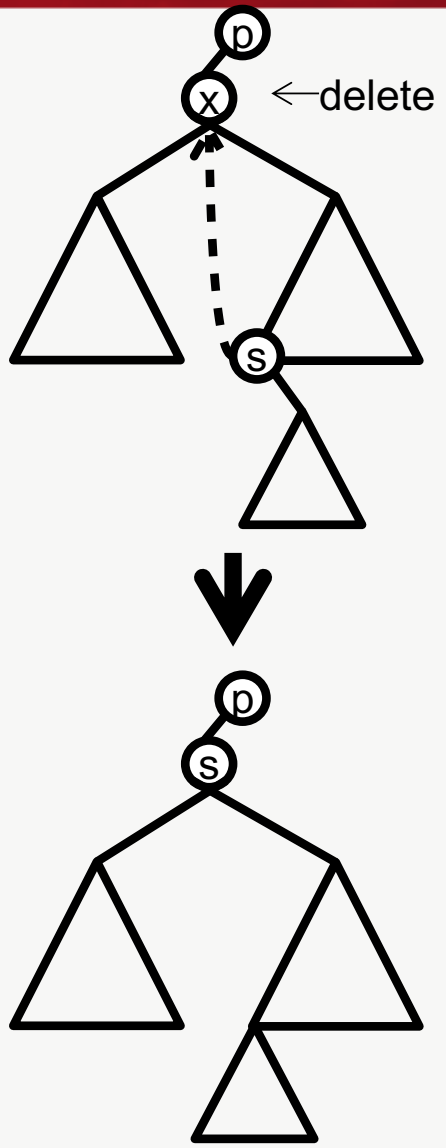


Delete 36, 16 and 15

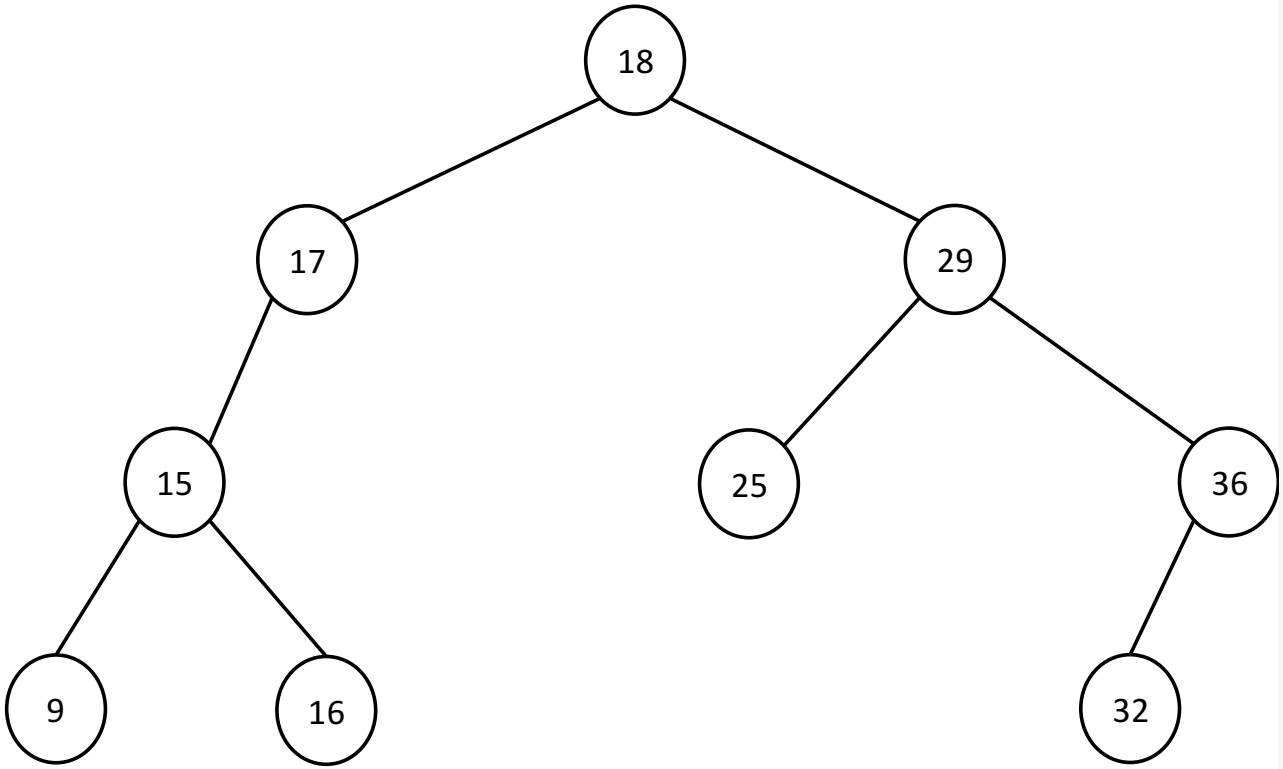
Deletion (cntd.)

If neither child is null, replace node (x) with its direct successor (s), which is the left-most node in the right subtree. This can be easily removed as it has at most one child (see previous slide) .

Alternatively, its direct predecessor can also be used.
In either case it may make the tree unbalanced.



Deletion example



Delete 15 and 18

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

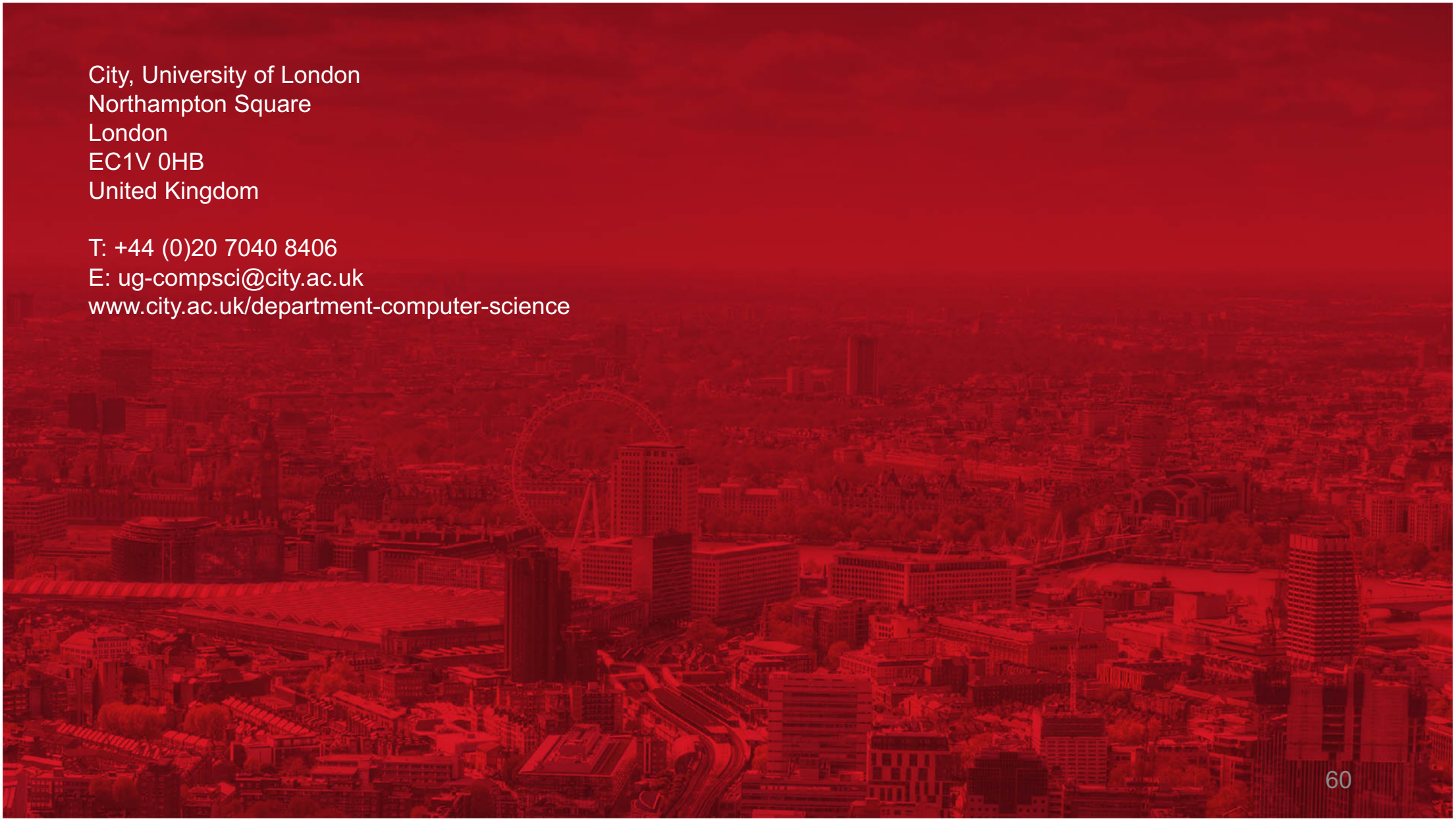
Reading

- Weiss: Chapters 17 and 18.1-18.3

- Drozdek: Sections 6.1 to 6.6

Next session: Advanced Trees

Drozdek: Sections 6.7, 6.8, and 7.1.1 OR Weiss: Sections 18.4, 18.8, 21.1-21.3, 21.5-21.7



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