# Algorithms & Data Structures

Part – 3: Topic 2

Anish Jindal
anish.jindal@durham.ac.uk

# Trees

# Trees

Now for something more data-structurely.

General trees will be studied in Part 4, we'll use only a special type:

(Rooted) binary trees.

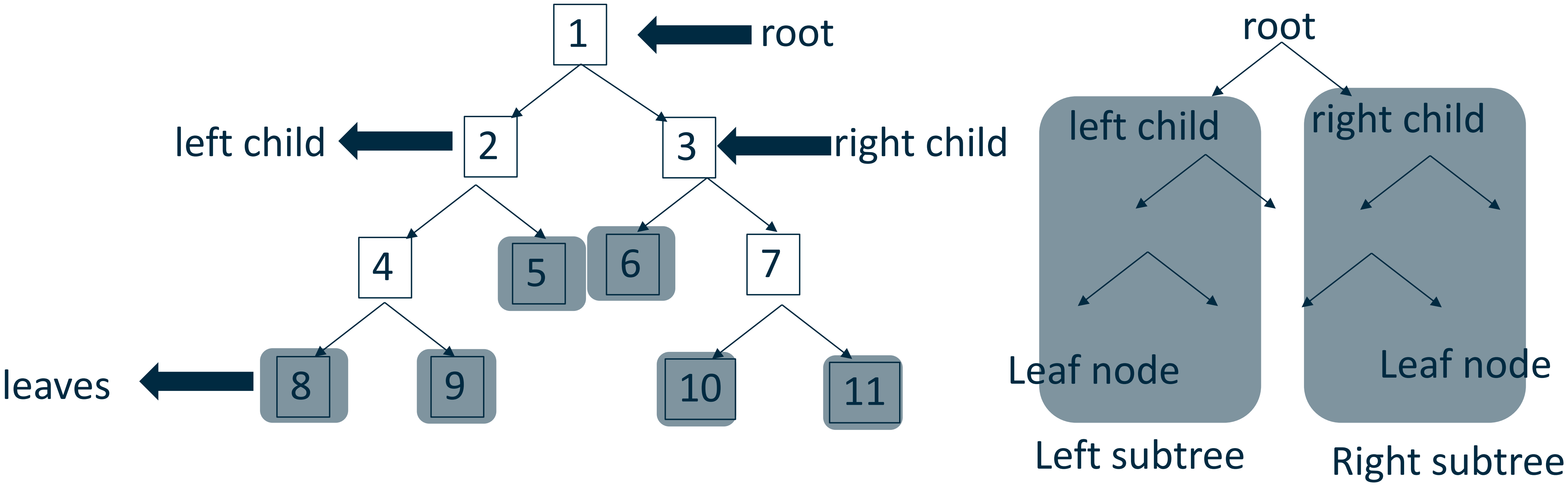
# (Rooted) binary tree

A tree whose elements have at most 2 children is called a binary tree.

Since each element can have only 2 children, we typically name them the left and right child.

Rooted - one node, known as the root, is designated as the topmost node.

- All other nodes in the tree are descendants of the root.


1
root
left child
2
3
right child
4
5
6
7
leaves
8
9
10
11
root
left child
right child
Leaf node
Leaf node
Left subtree
Right subtree

A (rooted) binary tree is a finite set of nodes which are either empty or consists of a root and two disjoint binary trees – left subtree and right subtree.

- one entry point, the ***root***
  - parentless
- Every node has **at most** two children
  - non-root node has a unique “parent” drawn above it or each node’s “children” are drawn right below it
- Each child node is labeled as being either **left child** or **right child**
  - Child-less nodes are called “leaves”

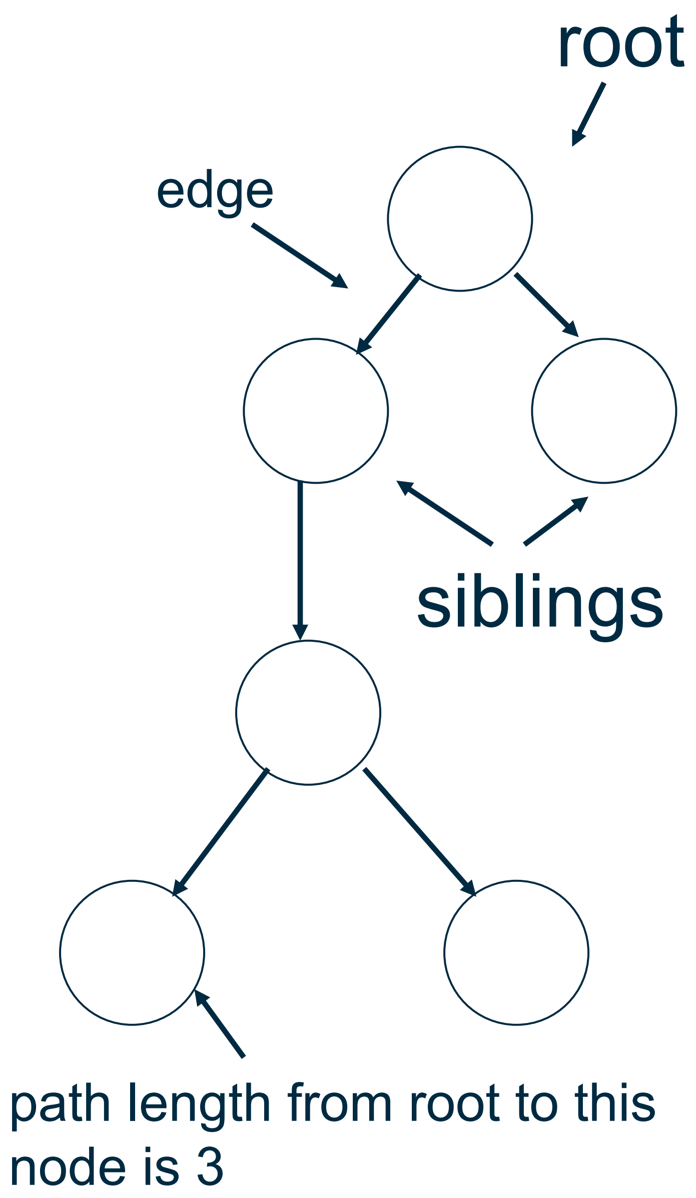
# Properties of rooted binary trees

Every node (excluding root) is connected by a directed edge from exactly one other node.

*edge:* the link from one node to another

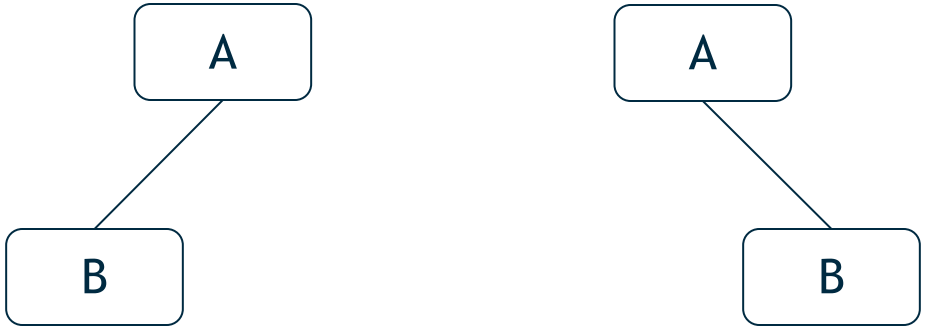
*siblings:* two nodes that have the same parent

*path length:* the number of edges that must be traversed to get from one node to another



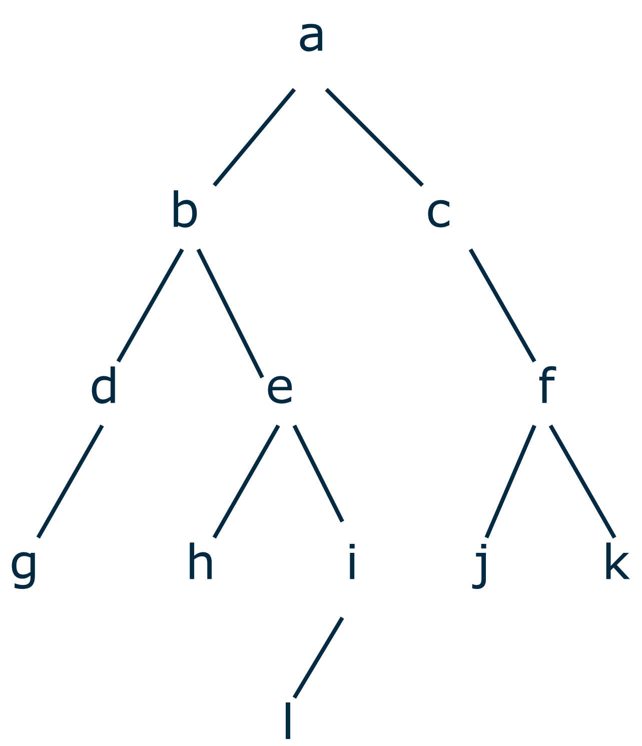
# Left ≠ Right

The following two binary trees are *different:*




In the first binary tree, node A has a left child but no right child;
in the second, node A has a right child but no left child

# Size and depth




The **size** of a binary tree is the number of nodes in it

- This tree has size 12

The **depth** of a node is its distance from the root

- a is at depth zero
- e is at depth 2

The **depth** of a binary tree is the depth of its deepest node

- This tree has depth 4

# Properties of rooted binary trees

**Lemma**

*Let $T$ be a rooted binary tree of height $h$. Then*

- *$T$ has at most $2^{h+1} - 1$ nodes,*
- *$T$ has at most $2^h$ leaves.*

**Proof.**

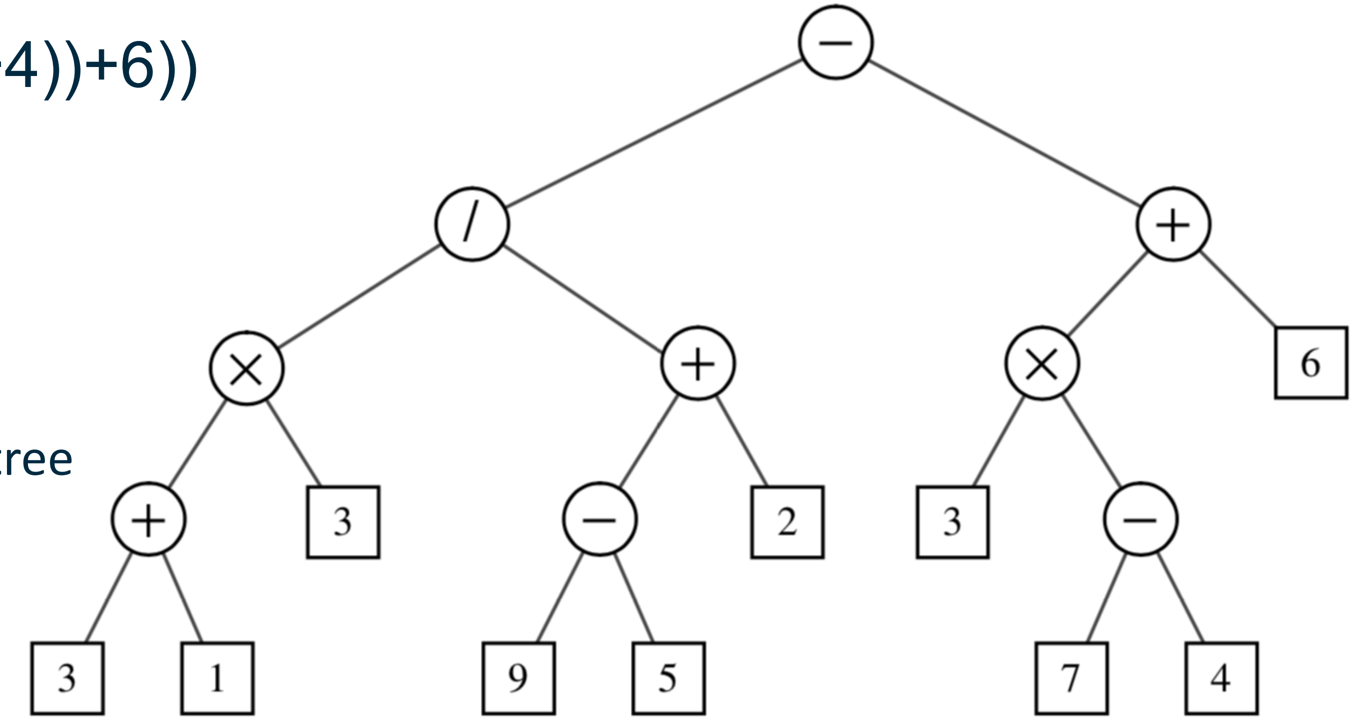
The max number of nodes is in a complete tree of height $h$ (i.e. all levels are complete): $1 + 2 + 2^2 + \ldots + 2^h = 2^{h+1} - 1$.

The second statement follows by induction on $h$. Case $h = 0$ is obvious. Consider the left and right subtrees of the root of $T$. Then each of them has height $\leq h - 1$ and (by induction hypothesis) $\leq 2^{h-1}$ leaves. Then $T$ has at most $2^{h-1} + 2^{h-1} = 2^h$ leaves. □

# Arithmetic expression represented by a binary tree

- An arithmetic expression can be represented by a binary tree whose leaves are associated with variables or constants, and whose internal nodes are associated with one of the operators +, −, ×, and /.
- The tree below represents the expression:

$$((((3+1)\times 3)/((9-5)+2))-((3\times(7-4))+6))$$




An arithmetic expression tree is a “proper” binary tree

# We'll be using binary trees to store data

How? Perhaps more importantly, why?

Well, they aren't really all that different from lists

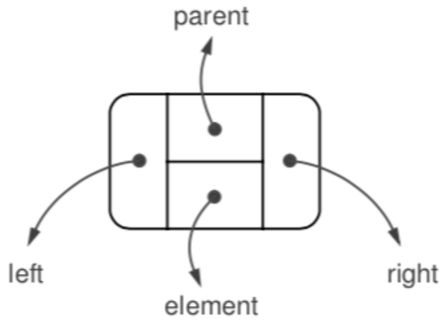
Recall: in a (doubly-linked) list, each node had

- pointer to predecessor, pointer to successor, item

In a binary tree, it's very similar: each node has

- pointer to parent (or NULL, or possibly to itself, if root)
- pointer to left child (or NULL, or to itself, if there isn't one)
- pointer to right child (or NULL, or to itself, if there isn't one)
- item (or element)

Can then navigate tree much like a list



# Why indeed?

Because they're terribly useful to store data in, giving fast **insert**, **lookup**, and **delete** operations (dictionary operations)!

(. . . if done properly)

The idea is simple: suppose you've got a sequence of such "dictionary operations", e.g.

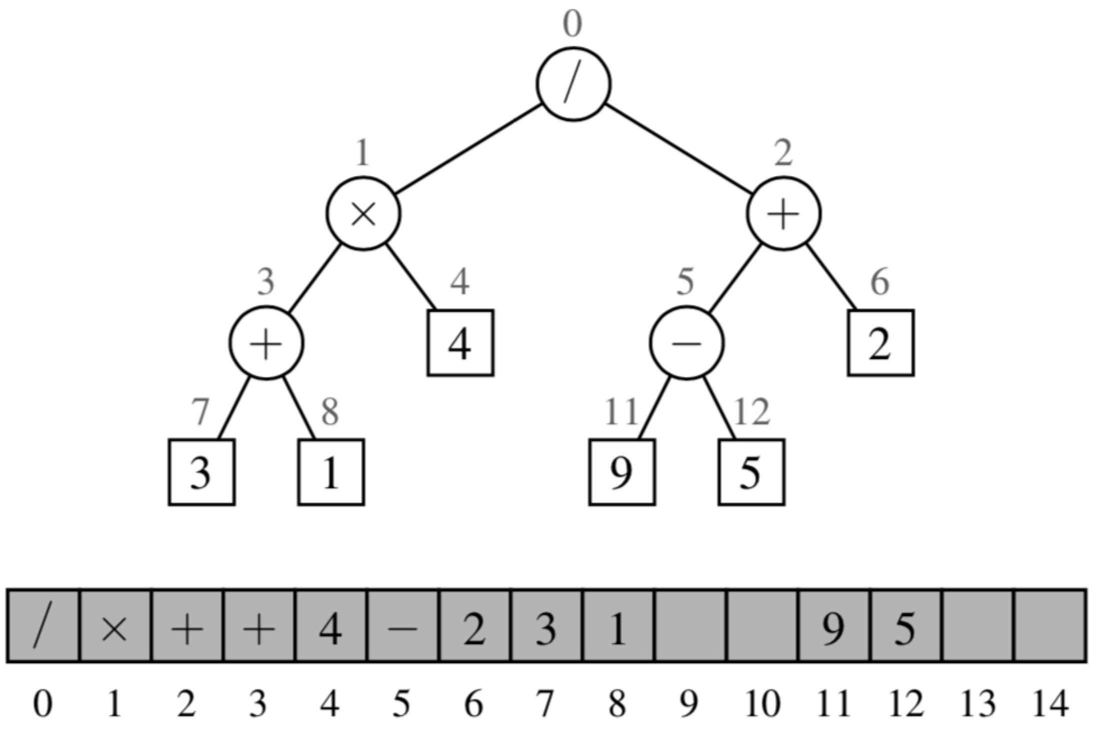
insert(12), insert(27), delete(12), insert(12), lookup(27), . . .

Start with an empty tree

- For each **insert** operation, traverse existing tree according to fixed rules, and insert new element in appropriate place
- For each **lookup** operation, traverse tree according to fixed rules
- For each **delete** operation, first do a lookup, and, if found, delete (and fix tree structure if necessary)

# Array- representation of binary tree

One advantage of an array-based representation of a binary tree is that a position $p$ can be represented by the single integer.

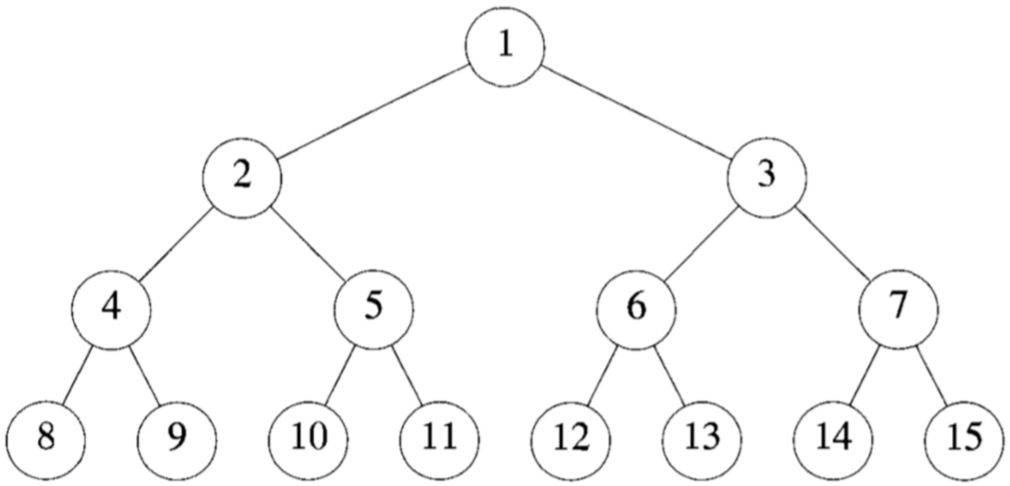



However, space usage of an array-based representation depends greatly on the shape of the tree

Some update operations for trees cannot be efficiently supported.

For example, deleting a node and promoting its child takes $O(n)$ time because it is not just the child that moves locations within the array, but all descendants of that child.

# How to search in a binary tree?




(1) Start at the root

(2) Search the tree level by level, until you find the element you are searching for or you reach a leaf.

Is this better than searching a linked list?

No → $O(n)$

# Binary Search Tree (BST)

# Binary search tree (BST)

The simple binary search tree may, for some input sequences, not be particularly good

Anyway, basic principle of dictionary data structures could just as well be implemented with list:

- insert(x): append to list
- lookup(x): traverse list and return "TRUE" (plus perhaps pointer to location of element) if found
- delete(x): first lookup, then splice out

BST (binary search tree) not all that different, really.

A **binary search tree** (BST) is a tree in which no node has more than two children (not necessarily exactly two).

The one additional crucial property of BSTs (this is what we call an invariant):

**BST property**

You must build and maintain the tree such that it's true for **every node** $v$ of the tree that:

- **all elements in its left sub-tree are "smaller" than** $v$
- **all elements in its right sub-tree are "bigger" than** $v$

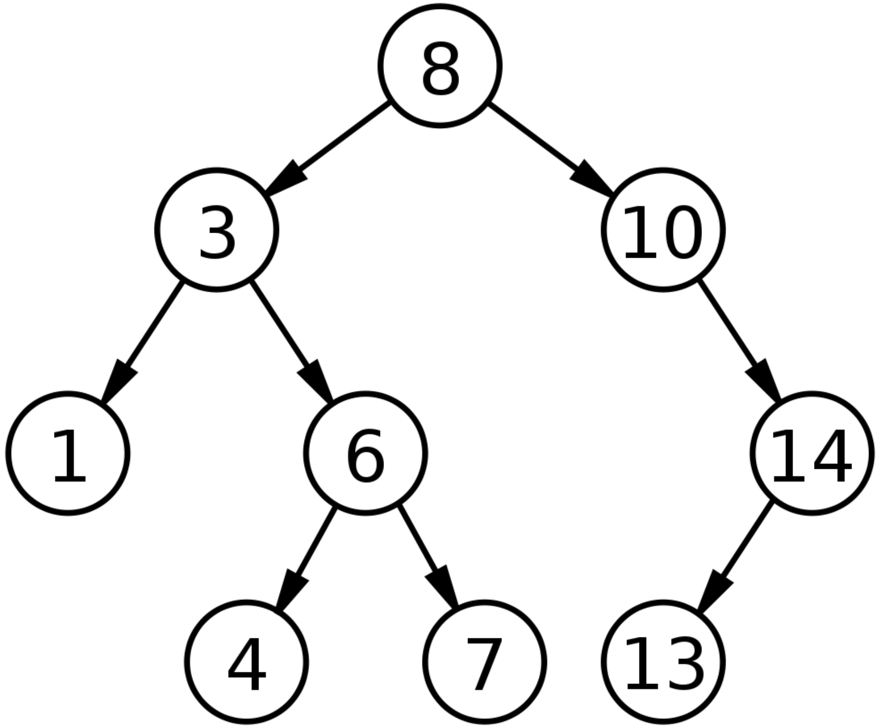
Smaller and bigger refer to the value. The left/right sub-tree refers to the tree rooted in a node's left/right child.

Just saying "left child smaller and right child bigger" not sufficient!

(Why?)

**Your operations that modify the tree must take care not to destroy this property!**

# Example



Before we go to BST operations:

May know **tree traversals** from elsewhere.

To **traverse** (or **walk**) the tree is to visit each node in the tree exactly once

Tree traversals are naturally recursive

Since a BST has three “parts,” there are six possible ways to traverse the binary tree:

- root, left, right
- left, root, right
- left, right, root
- root, right, left
- right, root, left
- right, left, root

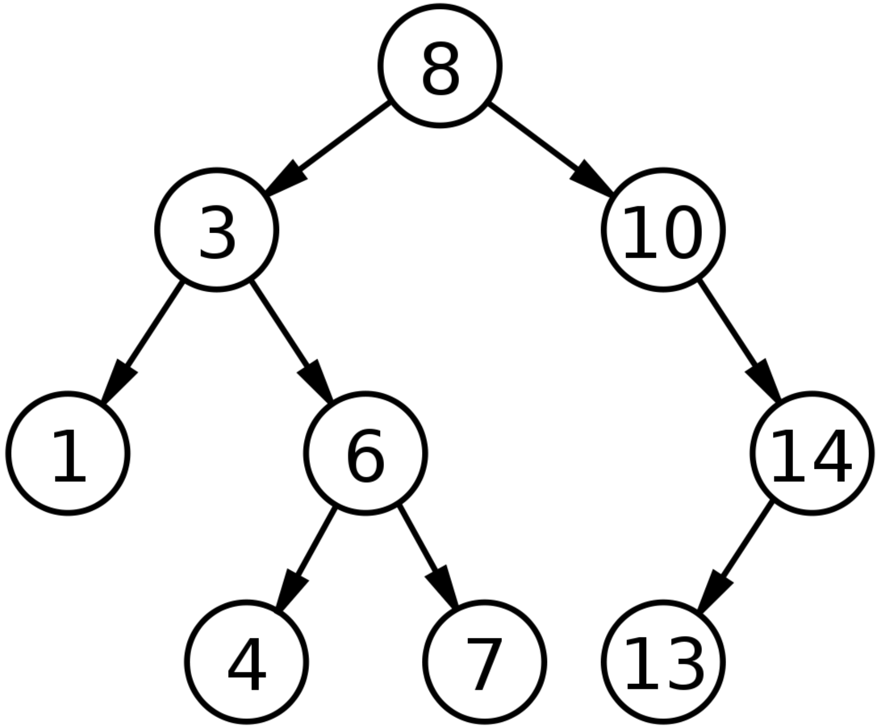
# Preorder traversal

In preorder, the root is visited *first*

Here's a preorder traversal to print out all the elements in the tree:

```
PREORDER-TREE-WALK (x)

if x ≠ NIL
        print x.key
        PREORDER-TREE-WALK (x.left)
        PREORDER-TREE-WALK (x.right)
```

# Example



# Inorder traversal

In inorder, the root is visited *in the middle*

Here’s an inorder traversal to print out all the elements in the tree:

```
INORDER-TREE-WALK (x)

if x ≠ NIL
        INORDER-TREE-WALK (x.left)
        print x.key
        INORDER-TREE-WALK (x.right)
```

# Postorder traversal

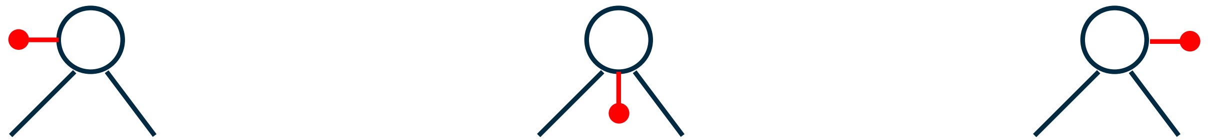
In postorder, the root is visited *last*

Here's a postorder traversal to print out all the elements in the tree:

```
POSTORDER-TREE-WALK (x)

if x ≠ NIL
        POSTORDER-TREE-WALK (x.left)
        POSTORDER-TREE-WALK (x.right)
        print x.key
```

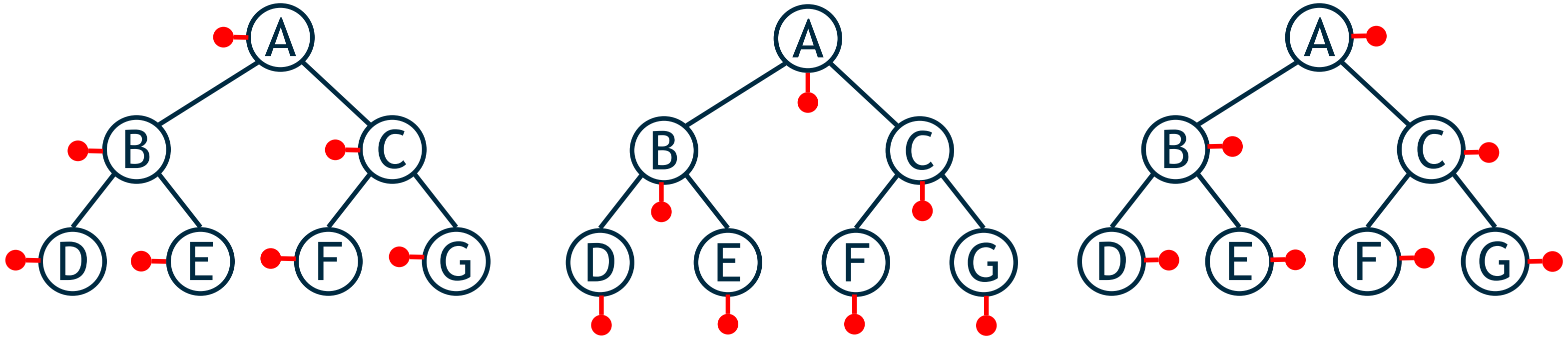
# Tree traversals using "flags"

The order in which the nodes are visited during a tree traversal can be easily determined by imagining there is a "flag" attached to each node, as follows:



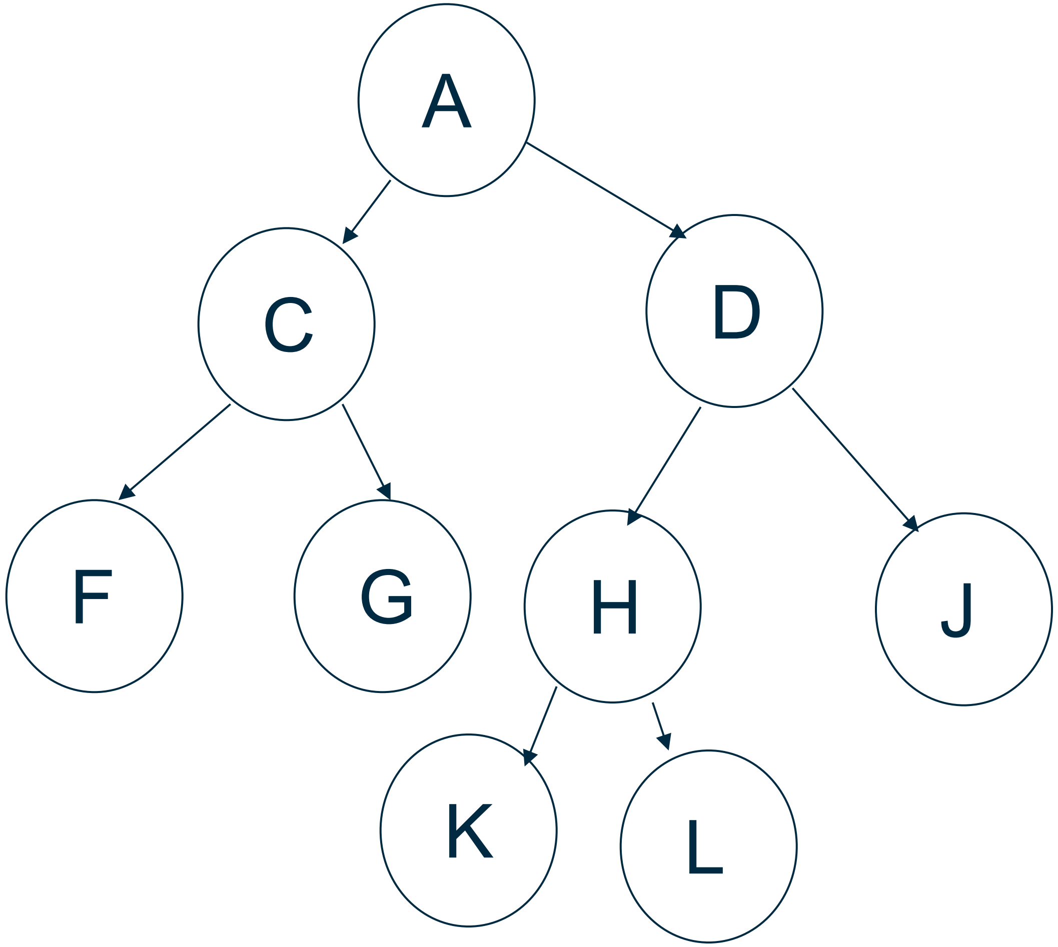
preorder | inorder | postorder

To traverse the tree, collect the flags:




A B D E C F G | D B E A F C G | D E B F G C A

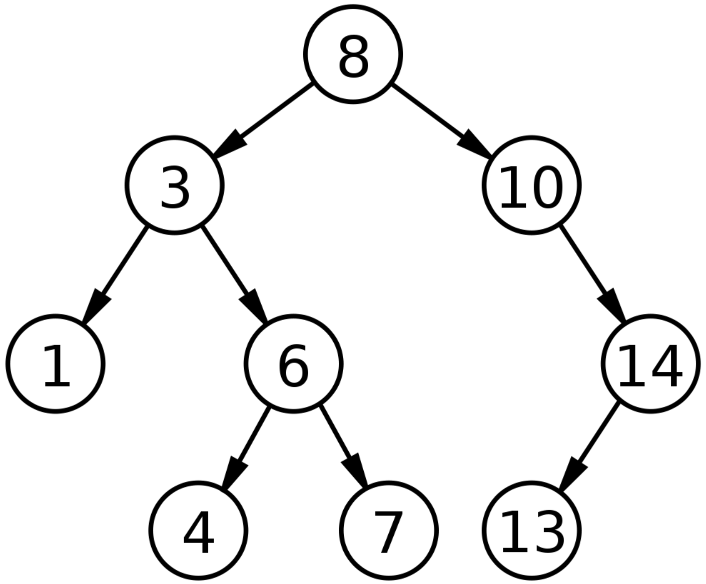
# Test your knowledge!




What is the result of a post order traversal of the tree to the left?

A. F C G A K H L D J
B. F G C K L H J D A
C. A C F G D H K L J
D. A C D F G H J K L
E. L K J H G F D C A

# Examples




- in-order: 1,3,4,6,7,8,10,13,14
- pre-order: 8,3,1,6,4,7,10,14,13
- post-order: 1,4,7,6,3,13,14,10,8

For BSTs, the in-order traversal gives the elements in sorted order!

No coincidence, either!

Anybody notice anything interesting?

# In-order traversal of BSTs sorts

By def, in BST

- everything on the left of a node is smaller, and
- everything on the right is bigger.

In-order traversal

- first recurses into left, then
- prints node, then
- recurses into right.

Means:

- first the entire left sub-tree is being printed (all the smaller guys),
- then the node itself,
- then the right sub-tree with the bigger guys.

Formal proof – in practicals!

# A class for BSTs

Assume we've got a data structure Node for nodes:

```
class Node:
    """
    Tree node: left and right child + data
               which can be any object
    """
    def __init__(self, data):
        """
        Node constructor
        @param data node data object
        """
        self.left = None
        self.right = None
        self.data = data
```

# Inserting into a BST

```
root = Node(8)
```

creates single node with data (value, payload, key) 8.

# Inserting into a BST

To be called on root of tree.

- If match, return (don't insert again).
- If new key is smaller than that of current node, insert on left.
- Otherwise, insert on right.

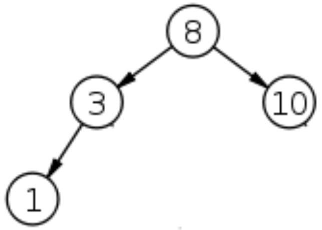
```python
class Node:
    ...
    def insert(self, data):
        """
        Insert new node with data
        @param data node data object to insert
        """
        if data < self.data:
            if self.left is None:
                self.left = Node(data)
            else:
                self.left.insert(data)
        else:
            if self.right is None:
                self.right = Node(data)
            else:
                self.right.insert(data)
```

# Inserting into a BST

Consider following sequence of operations after the root = Node(8)

```
root.insert(3)
root.insert(10)
root.insert(1)
```

That'll give us



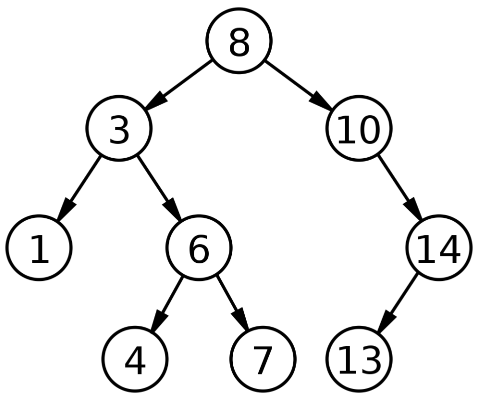

Durham University

# Inserting into a BST

Add this sequence:

```
root.insert(6)
root.insert(4)
root.insert(7)
root.insert(14)
root.insert(13)
```

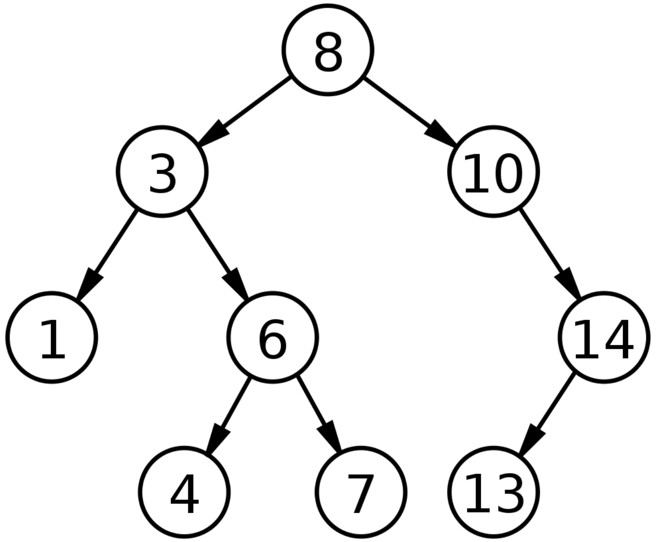
Output:



# Searching in a BST

To be called on the root of the tree.

- If match, return.
- If what we’re looking for is smaller, go left (can’t be anywhere else).
- Otherwise, go right.



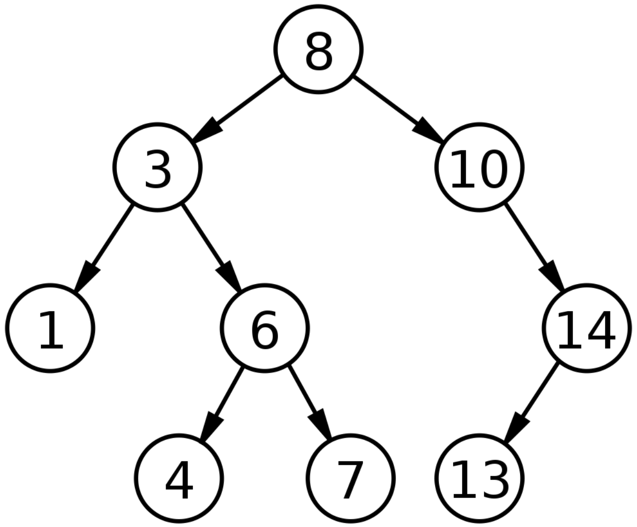
# Searching in a BST

```
class Node:
    ...
    def lookup(self, data, parent=None):
        """
        Lookup node containing data

        @param data node data object to look up
        @param parent node's parent
        @returns node and node's parent if found or None, None
        """
        if data < self.data:
            if self.left is None:
                return None, None
            return self.left.lookup(data, self)
        elif data > self.data:
            if self.right is None:
                return None, None
            return self.right.lookup(data, self)
        else:
            return self, parent
```

This method also returns parent (for later use)

# Searching in a BST

```
node, parent = root.lookup(7)
node, parent = root.lookup(15)
```



# What about deleting?

Most difficult dictionary operation on BSTs.

We first do a lookup on the element that we wish to remove. If that returns "not found" then we're done (element not in tree).

Otherwise, three cases to be considered:

- If node is a leaf (no child) then simply remove it
- If node has one child (left or right) then remove and replace it with that one child – lift (the one) sub-tree up.
- If node has two children then. . . what?

# Deleting from a BST: counting children

```
class Node:
    ...
    def children_count(self):
        """
        Returns the number of children

        @returns number of children: 0, 1, 2
        """
        if node is None:
            return None
        cnt = 0
        if self.left:
            cnt += 1
        if self.right:
            cnt += 1
        return cnt
```

# Deleting from a BST: setup

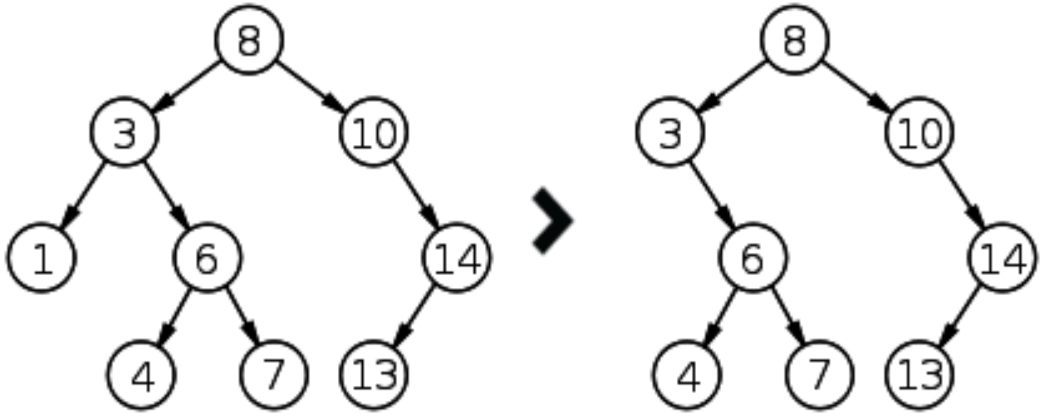
```
class Node:
    ...
    def delete(self, data):
        """
        Delete node containing data

        @param data node's content to delete
        """
        # get node containing data
        node, parent = self.lookup(data)
        if node is not None:
            children_count = node.children_count()
        ...
```

# Deleting from a BST: no child

```
def delete(self, data):
    ...
    if children_count == 0:
        # if node has no children, just remove it
        if parent.left is node:
            parent.left = None
        else:
            parent.right = None
        del node
    ...
```

# Deleting from a BST: no child

```
root.delete(1)
```




Durham University

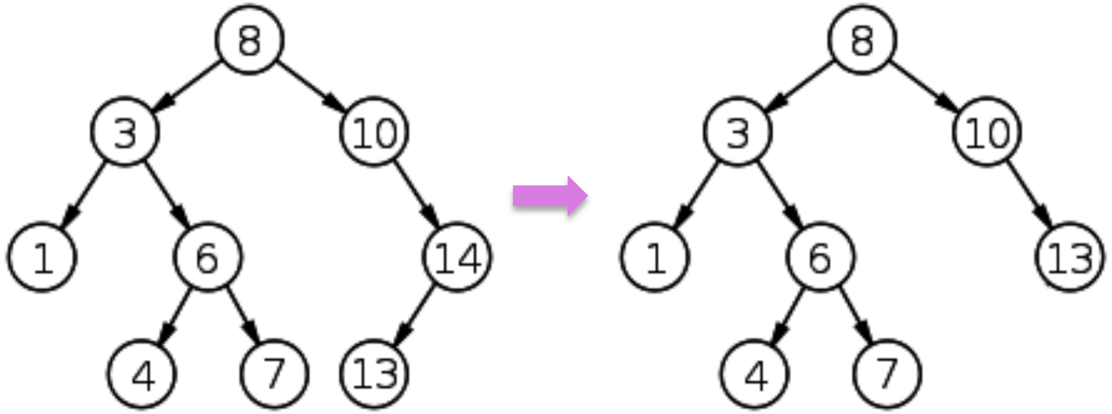
# Deleting from a BST: one child

```
def delete(self, data):
    ...
    elif children_count == 1:
        # if node has 1 child
        # replace node by its child
        if node.left:
                n = node.left
        else:
                n = node.right
        if parent:
            if parent.left is node:
                    parent.left = n
            else:
                    parent.right = n
        del node
    ...
```

Durham University

# Deleting from a BST: one child

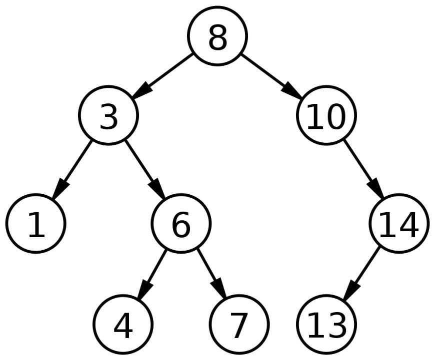
```
root.delete(14)
```




Durham University

# Deleting from a BST: two children

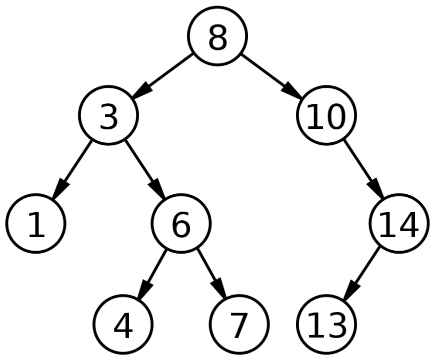
Before we go there: how do you find the smallest element in a given tree?




Answer: starting from root, always go left

# Deleting from a BST: two children

Before we go there: how do you find the smallest element bigger than that in a given node?




Answer: starting from that node, take one step to the right, then always go left

# Deleting from a BST: two children

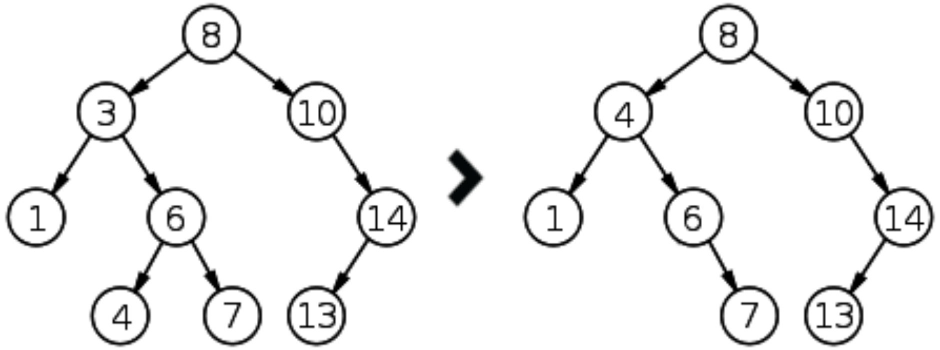
Why is that important, or useful?

Because if we wanted to remove a given node u, we could

- find the smallest node v that's bigger
- copy v's data into u
- delete v

Example:

*root.delete(3)*




BST property maintained, everything all right! (Could also have identified largest in left sub-tree and copied that node's data across!)

Durham University

# Deleting from a BST: two children

```
def delete(self, data):
    ...
    else:
        # if node has 2 children
        # find its successor
        parent = node
        successor = node.right
        while successor.left:
            parent = successor
            successor = successor.left
        # replace node data by its successor data
        node.data = successor.data
        # fix successor's parent's child
        if parent.left == successor:
            parent.left = successor.right
        else:
            parent.right = successor.right
```

"Successor" here means w.r.t. sorted order (left-most in right subtree)

# Traversing a BST

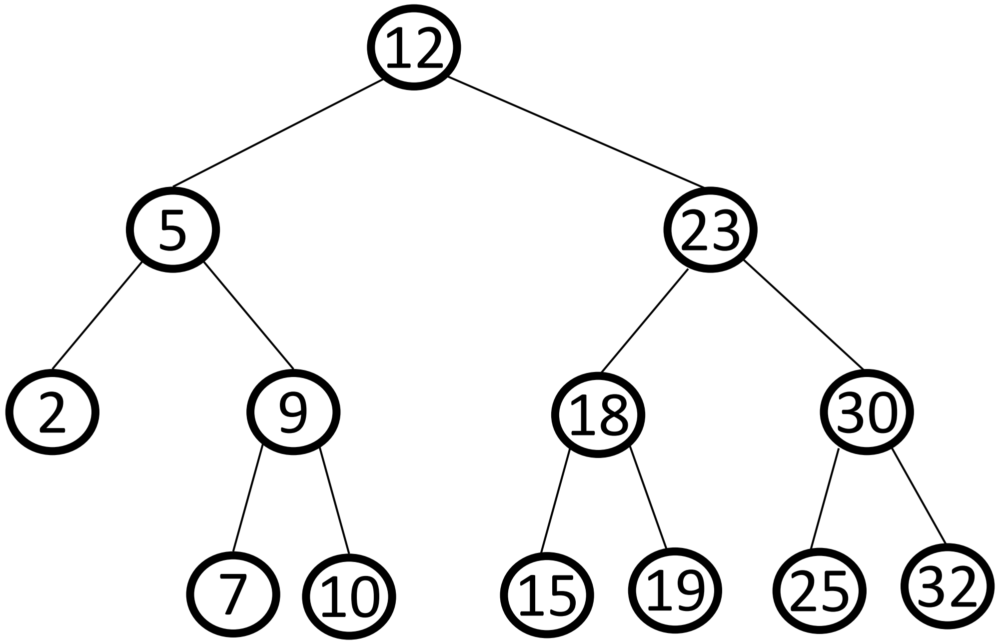
```
class Node:
    ...
    def print_tree(self):
        """
        Print tree content inorder
        """
        if self.left:
            self.left.print_tree()
        print self.data,
        if self.right:
            self.right.print_tree()
```

# BST: Complexity

- O(h)– searching, insertions, deletions.
  - O(log n) In the average case.
- In the worse case, these degenerates to O(N)  - how?
  - but this can be avoided by using balanced trees (AVL, Red-Black)

# Test your knowledge!

delete(23)



# Thank you