Data Structures & Algorithms

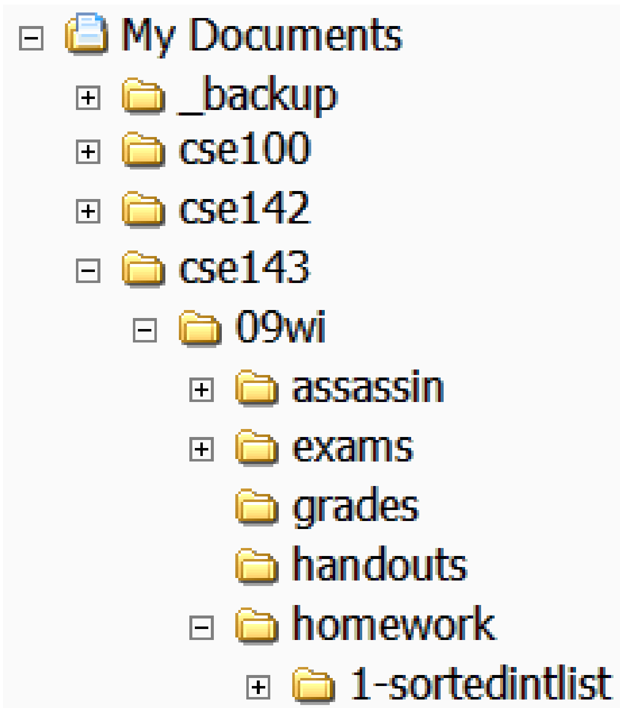
Tree

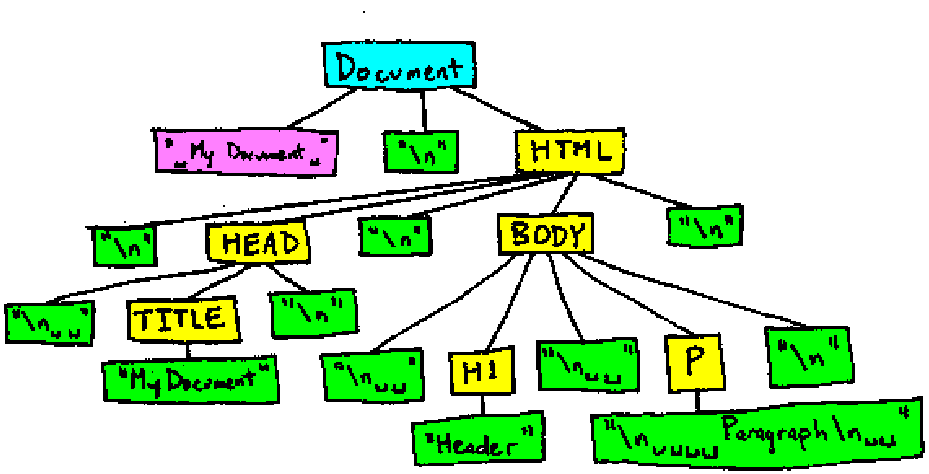
- Basics
- Binary Tree
- implementations
- Tree Traversal
- Binary Search Tree
- AVL Tree

Maastricht University

When do we need a tree?

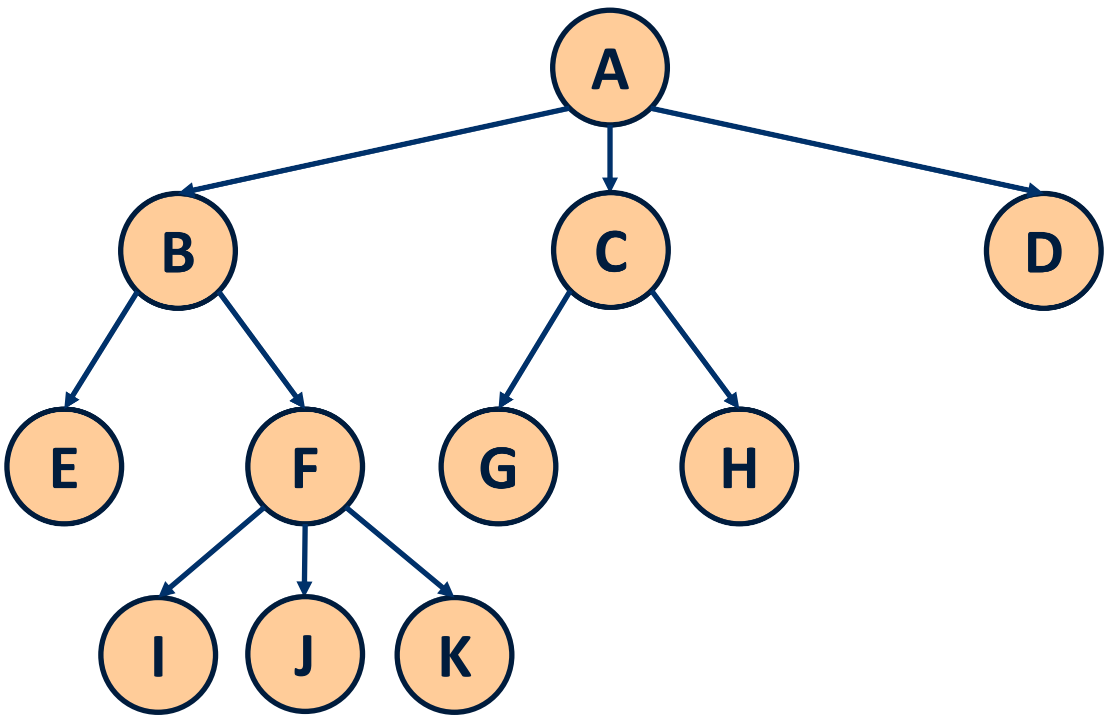
- Folders/files on a computer
- Organizational charts
- ML: decision trees
- HTML Document Structure



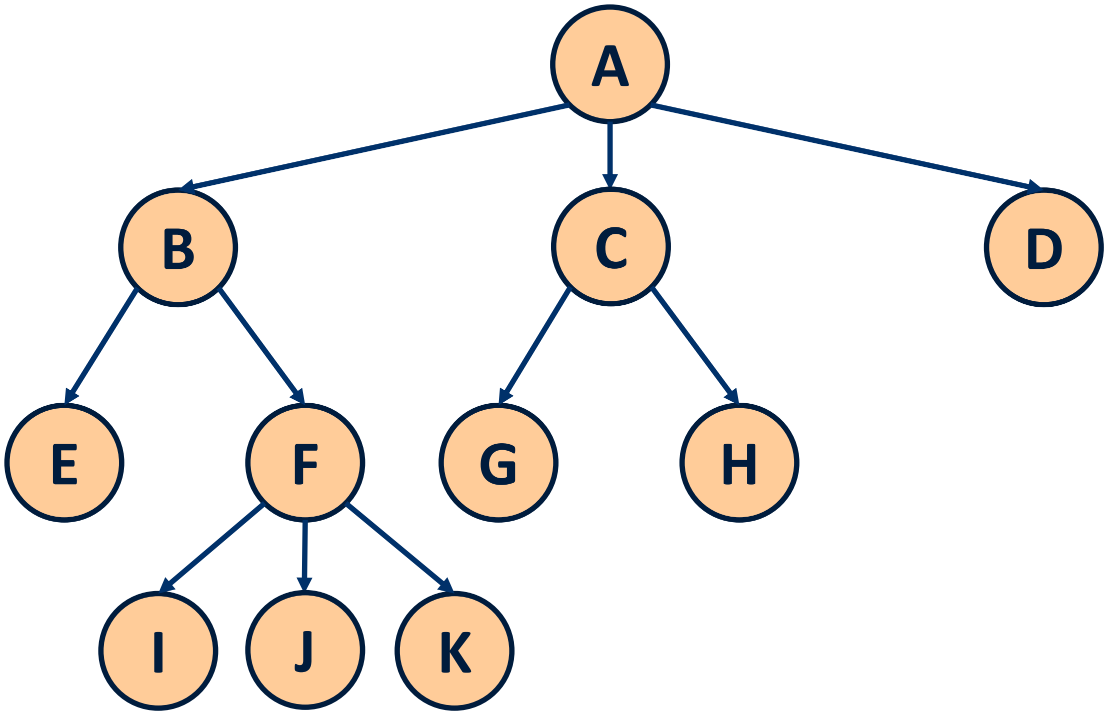


Definition

- **Connected** graph with **no cycles**
- Single path from **root** to **leaf**
- Nodes with **parent-child** relations

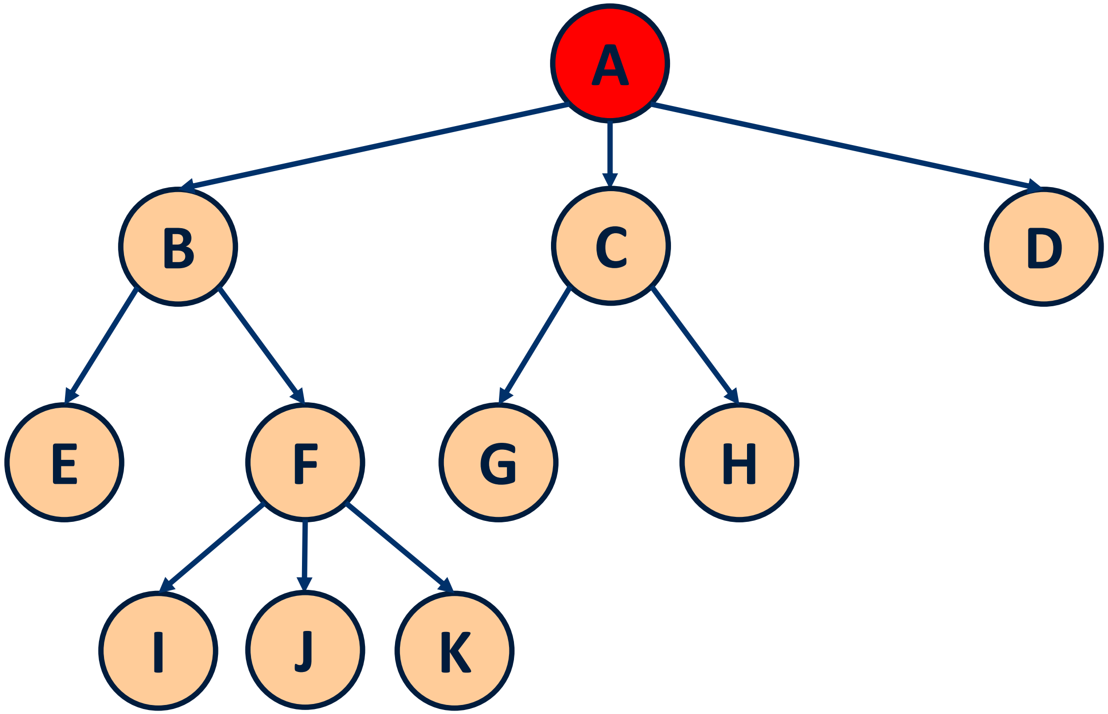


Terminology



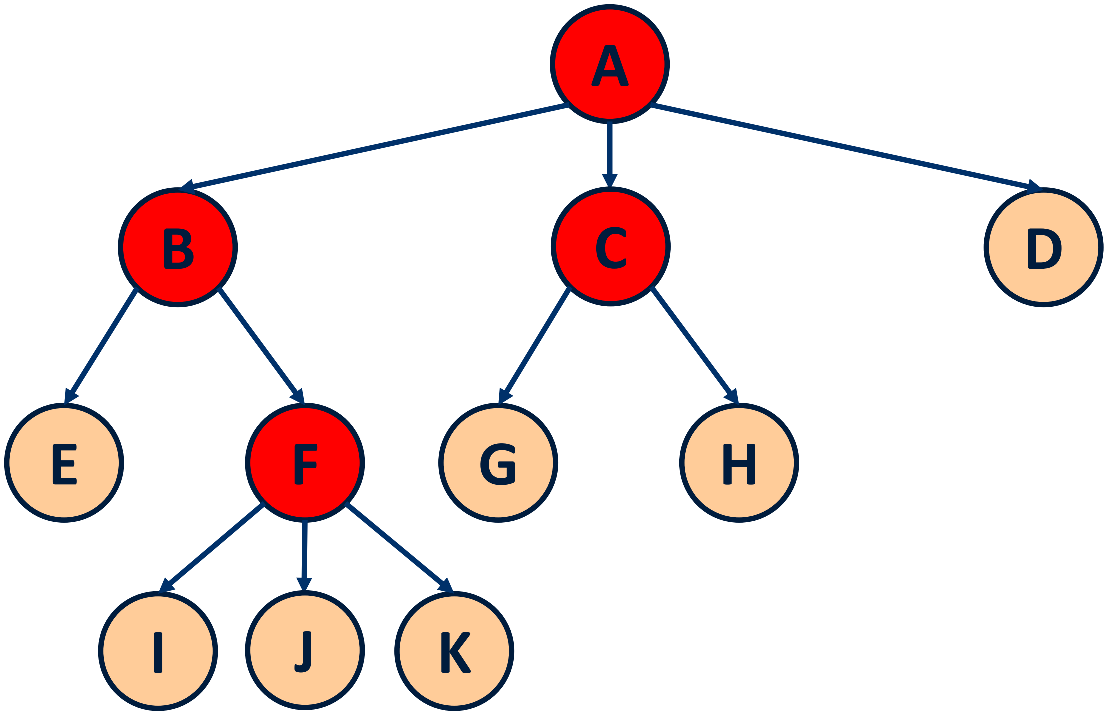
Terminology

- Root: only node with no parents



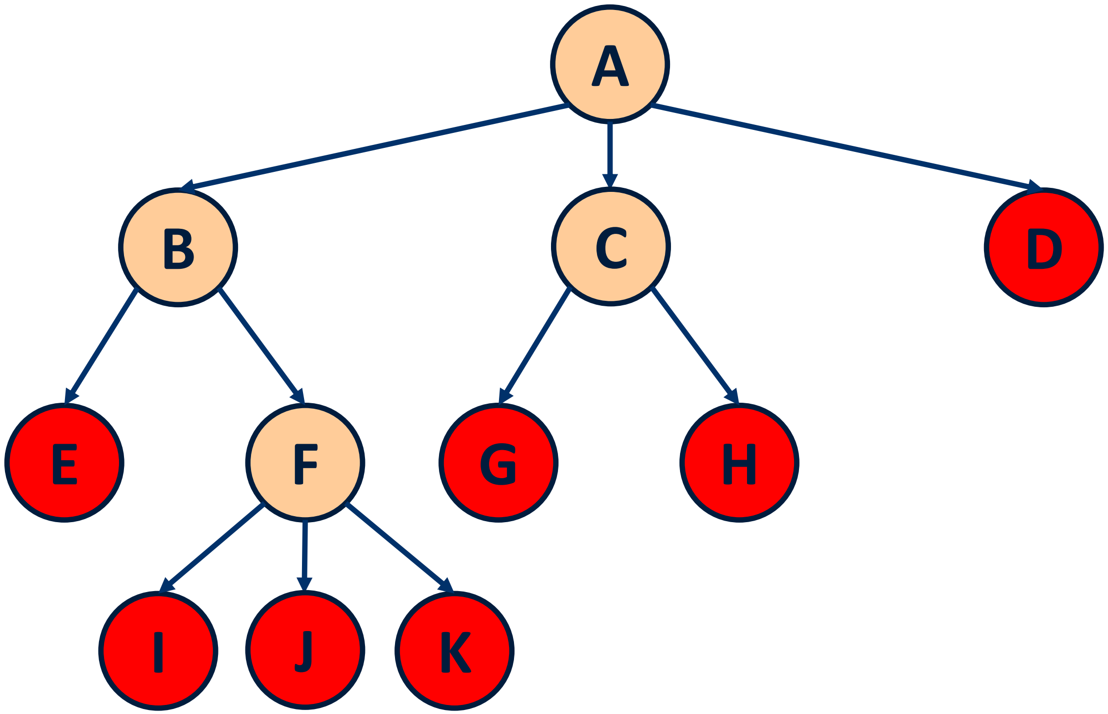
Terminology

- Root: only node with no parents
- Internal Node: any node with at least a child



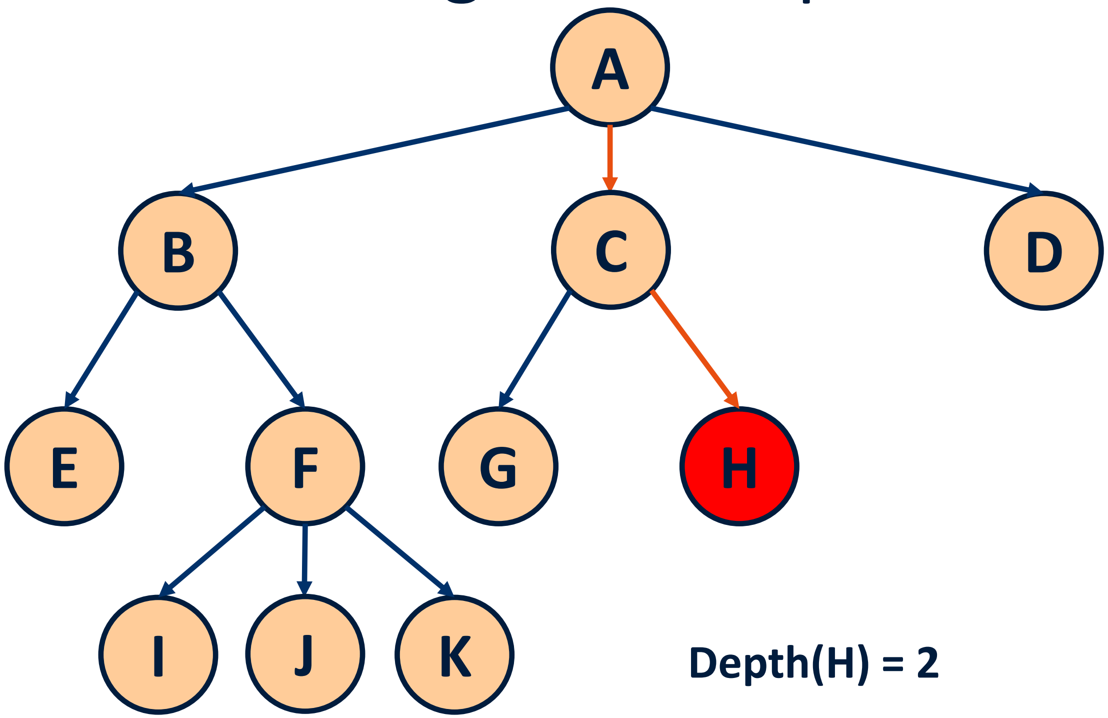
Terminology

- Root: only node with no parents
- Internal Node: any node with at least a child
- External Node (Leaf): any node with no children



Terminology

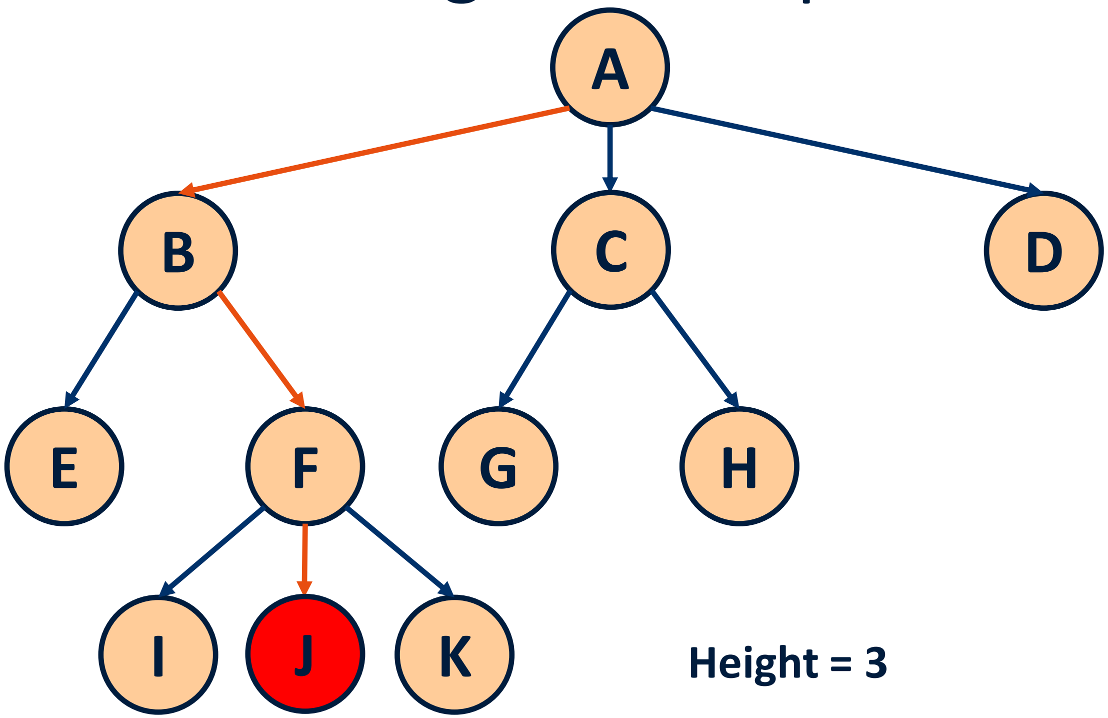
- Root: only node with no parents
- Internal Node: any node with at least a child
- External Node (Leaf): any node with no children
- Depth of a Node: Number of edges in the path from the root



Depth(H) = 2

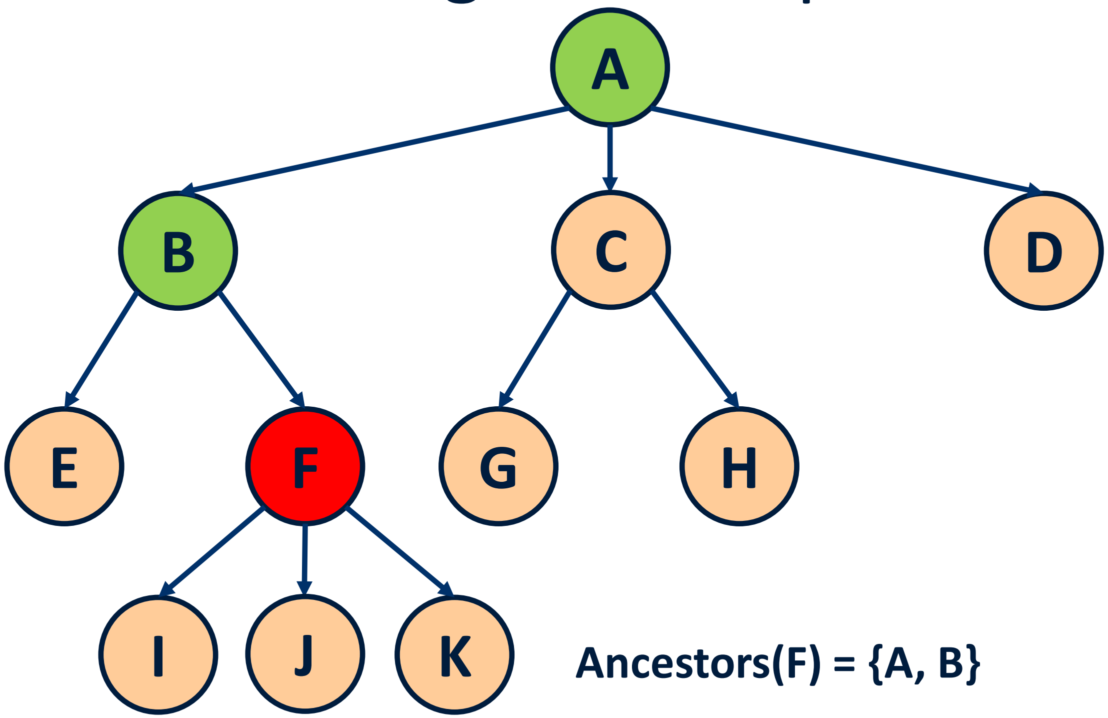
Terminology

- Root: only node with no parents
- Internal Node: any node with at least a child
- External Node (Leaf): any node with no children
- Depth of a Node: Number of edges in the path from the root
- Height: Max depth



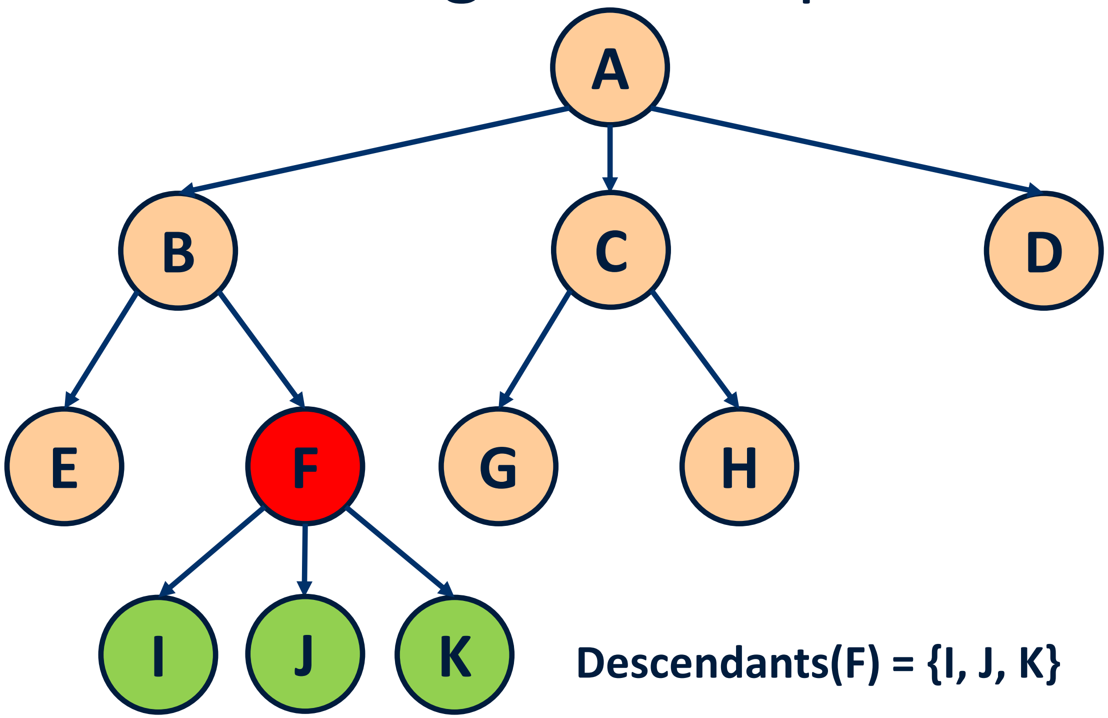
Terminology

- Root: only node with no parents
- Internal Node: any node with at least a child
- External Node (Leaf): any node with no children
- Depth of a Node: Number of edges in the path from the root
- Height: Max depth
- Ancestors



Terminology

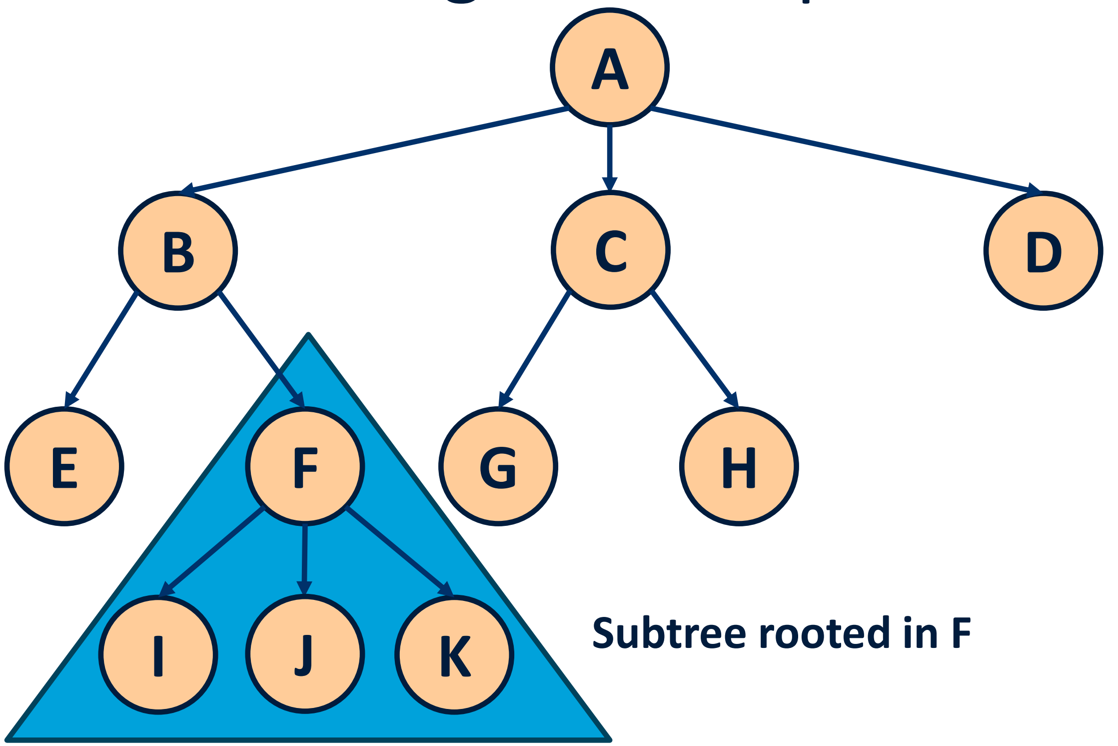
- Root: only node with no parents
- Internal Node: any node with at least a child
- External Node (Leaf): any node with no children
- Depth of a Node: Number of edges in the path from the root
- Height: Max depth
- Ancestors
- Descendants



Descendants(F) = {I, J, K}

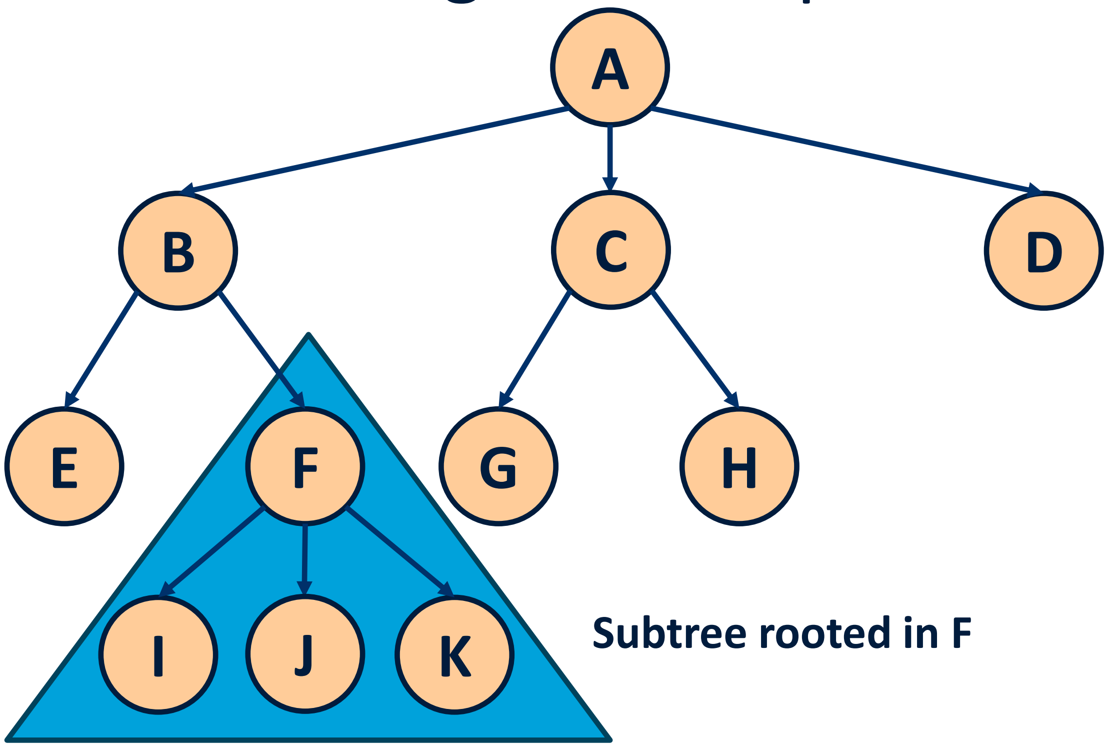
Terminology

- Root: only node with no parents
- Internal Node: any node with at least a child
- External Node (Leaf): any node with no children
- Depth of a Node: Number of edges in the path from the root
- Height: Max depth
- Ancestors
- Descendants
- Subtree



Terminology

- Root: only node with no parents
- Internal Node: any node with at least a child
- External Node (Leaf): any node with no children
- Depth of a Node: Number of edges in the path from the root
- Height: Max depth
- Ancestors
- Descendants
- Subtree
- Element=Key



Tree ADT

- Defines operations to manipulate the root element
  - The TreeNode will provide operations to add/remove children

- Main operations:
  - addRoot(E e): add an element e as root
  - getRoot(): returns the value e in the root
  - hasRoot(): checks if there is a root node
  - Size(): number of elements in the tree

Tree ADT

```
public interface Tree<E> {
    void addRoot(E e);
    TreeNode<E> getRoot();
    boolean hasRoot();
    int size();
}
```

TreeNode ADT

- Provides operations to modify the Node, add/remove children, check properties of the node

- Main operations:
  - get/setElement()
  - isRoot()/isInternal()/isLeaf()
  - getParent()/getChildren()/hasChild()/addChild()
  - Delete()

TreeNode ADT

```
public interface TreeNode<E> {
    E getElement();
    void setElement(E e);
    boolean isRoot();
    boolean isInternal();
    boolean isExternal();
    TreeNode<E> getParent();
    TreeNode<E>[] getChildren();
    void addChild(E e);
    void delete();
    boolean hasChild(E e);
}
```

Binary Tree

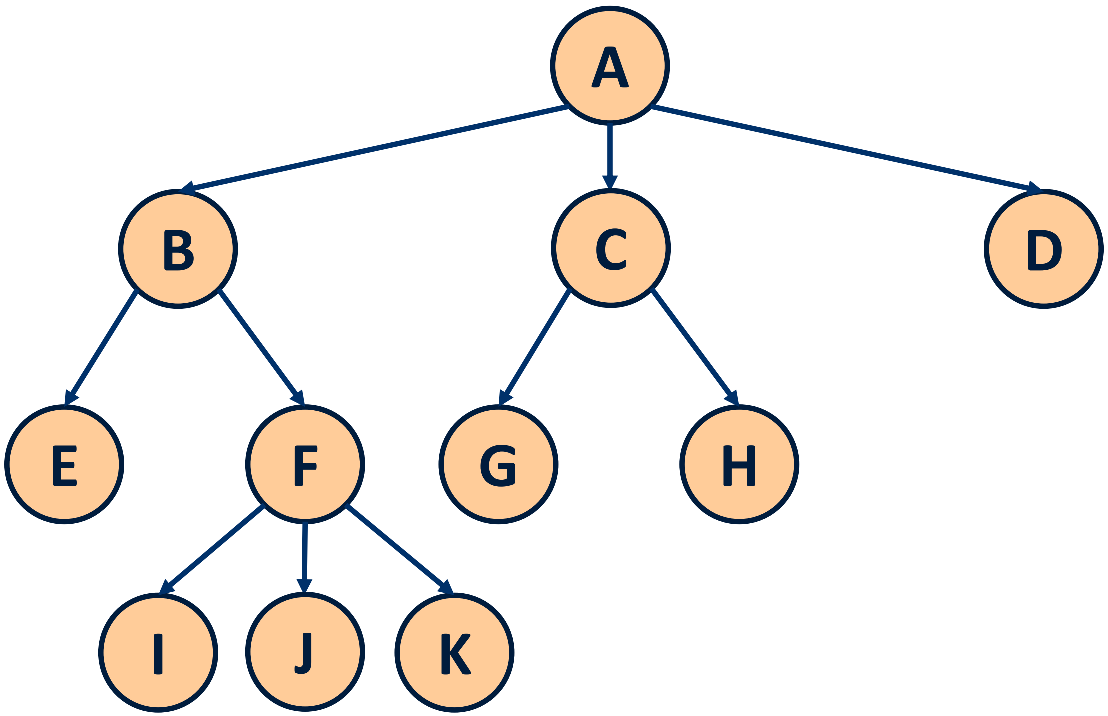
Binary Tree

- A binary tree is a tree with the following properties:
  - Each internal node has **at most** two children **(exactly two for proper binary trees)**
  - The children of a node are an **ordered pair**

- We call the children of an internal node **left child and right child**

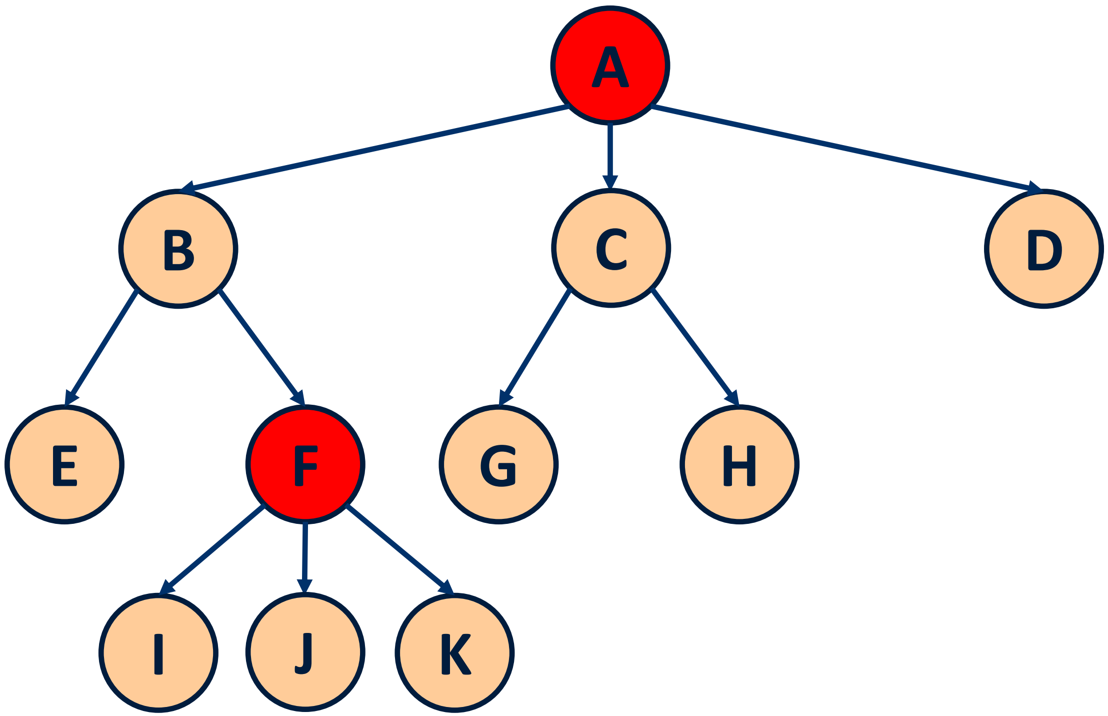
Examples

- Is this a Binary Tree?



Examples

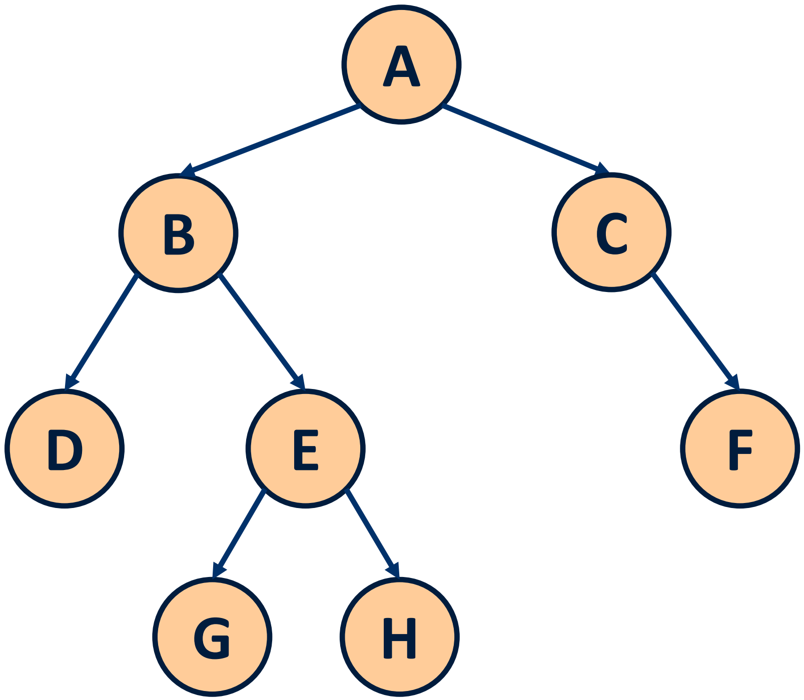
- Is this a Binary Tree?



- **No! There are internal nodes with 3 children**

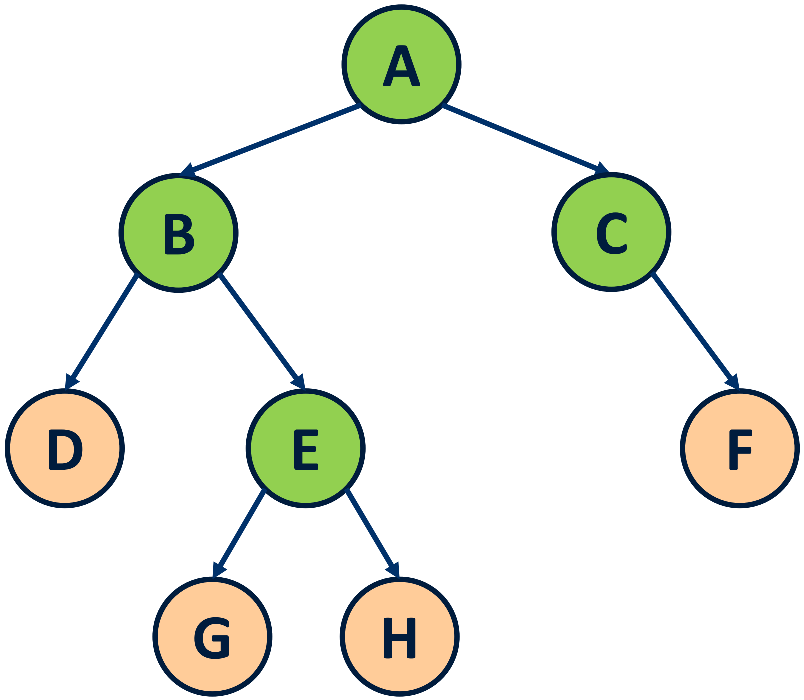
Examples

- Is this a Binary Tree?



Examples

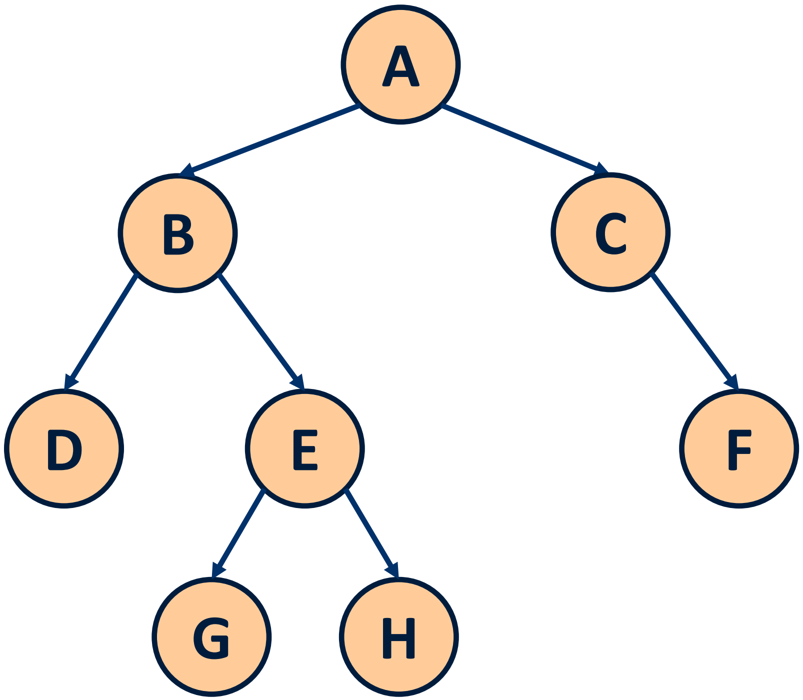
- Is this a Binary Tree?



- **Yes! Each internal node has at most 2 children**

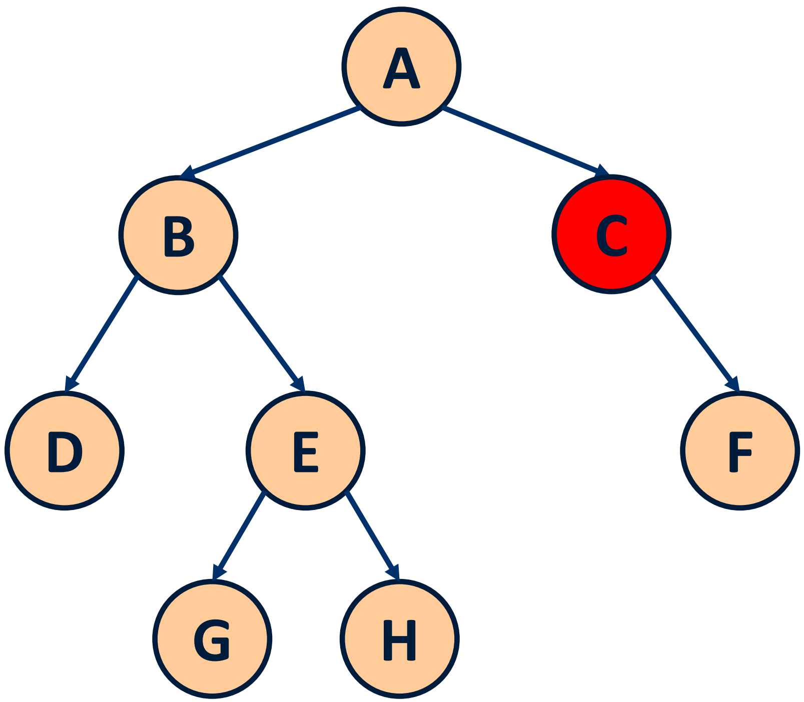
Examples

- Is this a **Proper** Binary Tree?



Examples

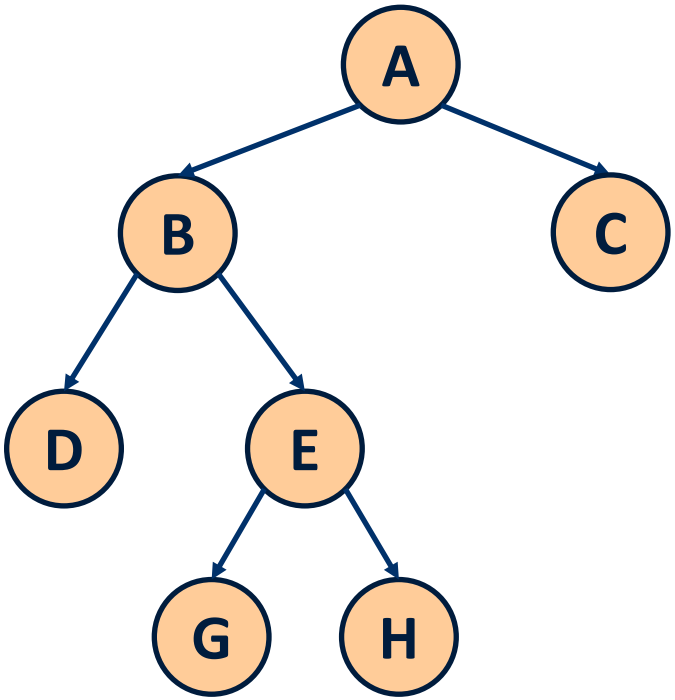
- Is this a **Proper** Binary Tree?



- **No! C only has 1 child**

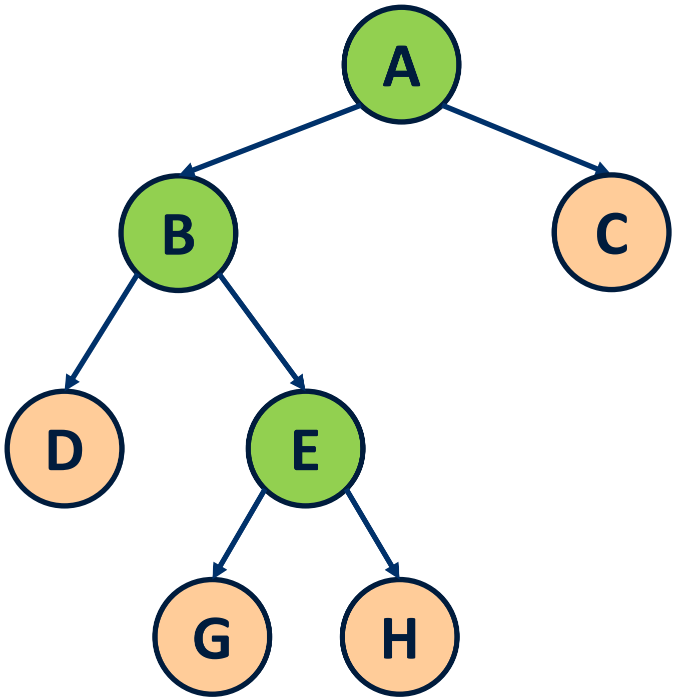
Examples

- Is this a **Proper** Binary Tree?



Examples

- Is this a **Proper** Binary Tree?



- **Yes! Each internal node has exactly 2 children**

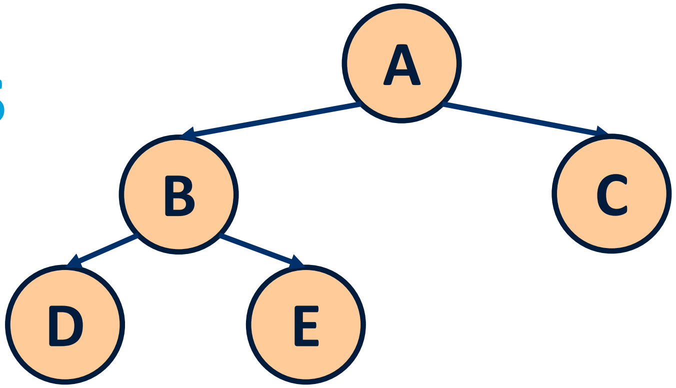
BinaryTree and BinaryTreeNode Operations

- In a binary tree each node provides operation to modify and read left and right children

- Operations:
  - leftChild()/rightChild()
  - addLeftChild()/addRightChild()

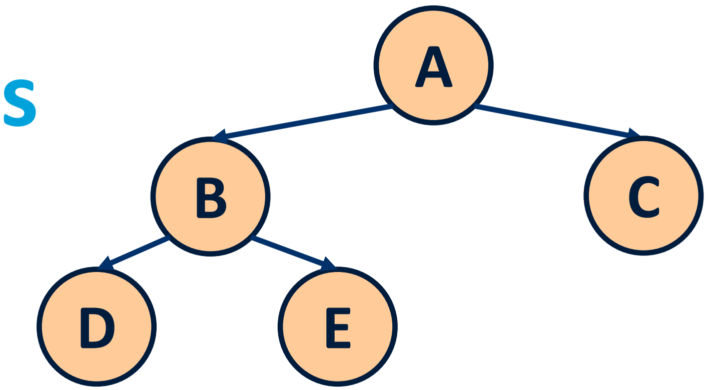
Binary Tree Implementations

Binary Tree implementations



- Two main strategies

Binary Tree implementations



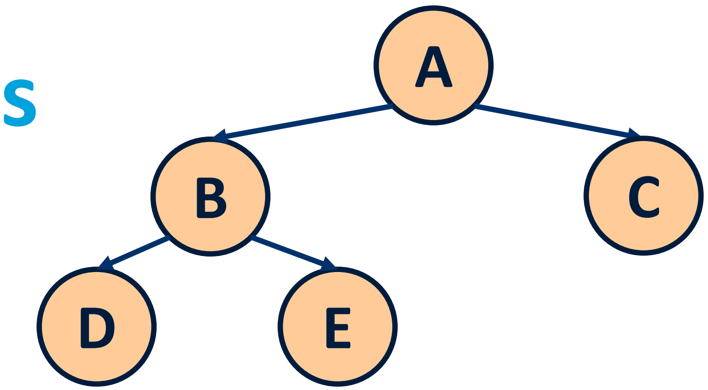
- Two main strategies

Array-based

A	B	C	D	E	∅	∅	∅
0	1	2	3	4	5	6	7

- Elements are stored in an array
- Direct access
- We need a way to obtain the loction of the children of a node
- The whole array must be allocated in memory

Binary Tree implementations



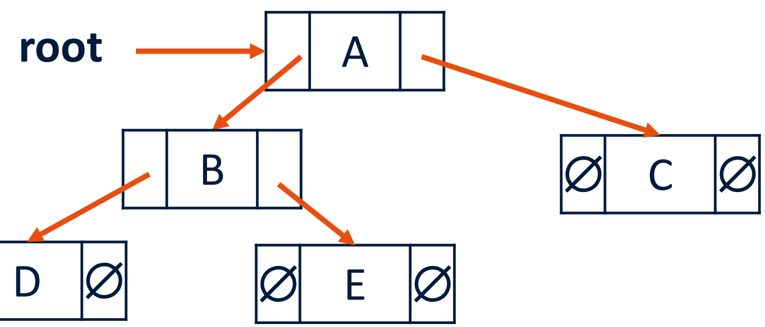
- Two main strategies

Array-based

A	B	C	D	E	∅	∅	∅
0	1	2	3	4	5	6	7

- Elements are stored in an array
- Direct access
- We need a way to obtain the loction of the children of a node
- The whole array must be allocated in memory

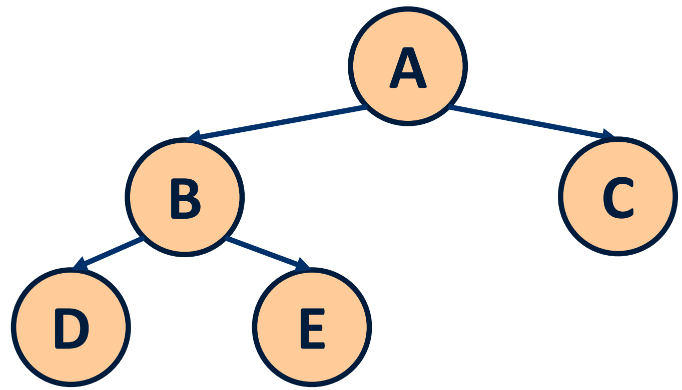
Linked-based



- Elements are stored in independent structures: Nodes
- The location of the root element is stored

Array-Based Binary Tree

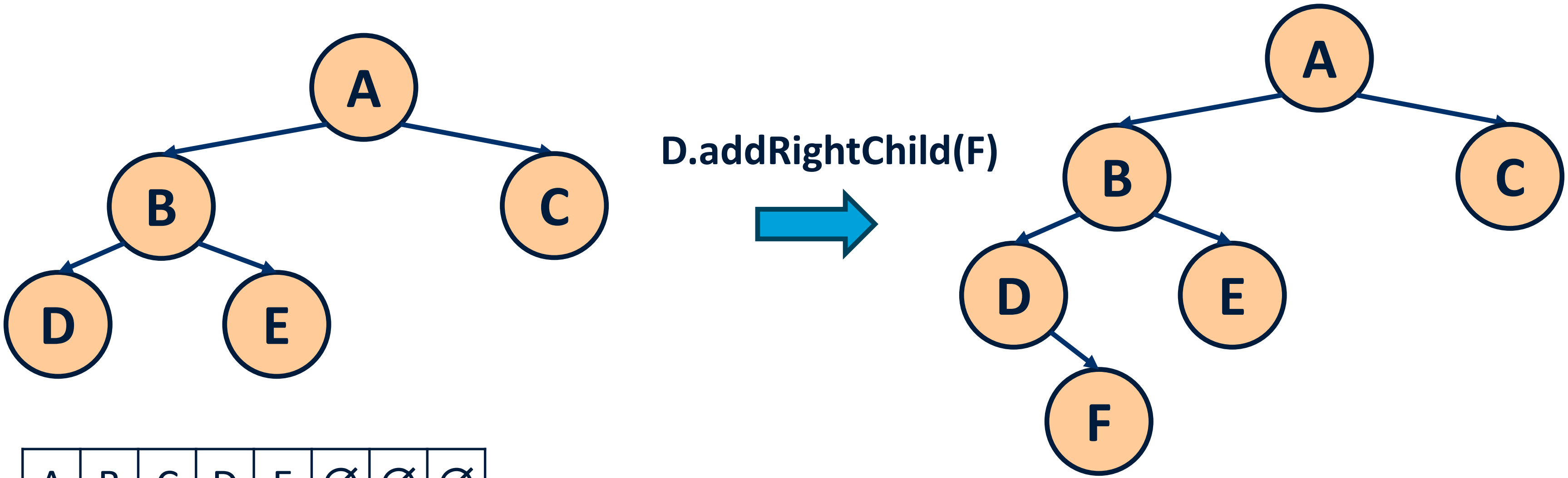
A	B	C	D	E	∅	∅	∅
0	1	2	3	4	5	6	7



- Index(root) = 0

- Given a generic node ***v*** stored at index ***i***
  - Index(left(v)) = 2* i + 1
  - Index(right(v)) = 2 * i + 2
  - Index(parent(v)) = $\lfloor (i - 1) / 2 \rfloor$ **// floor operator**

Array-Based Representation: Issues?



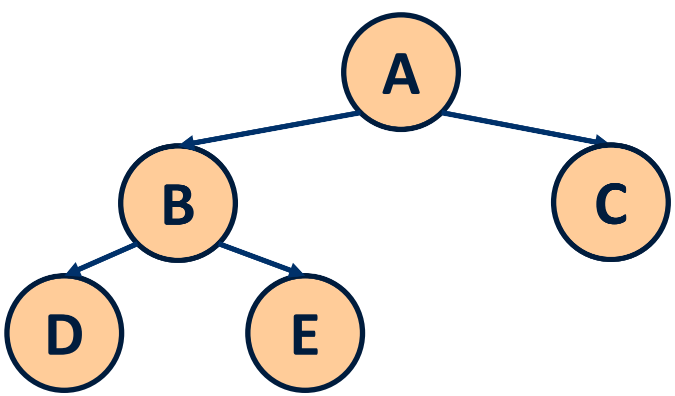
A	B	C	D	E	∅	∅	∅
0	1	2	3	4	5	6	7

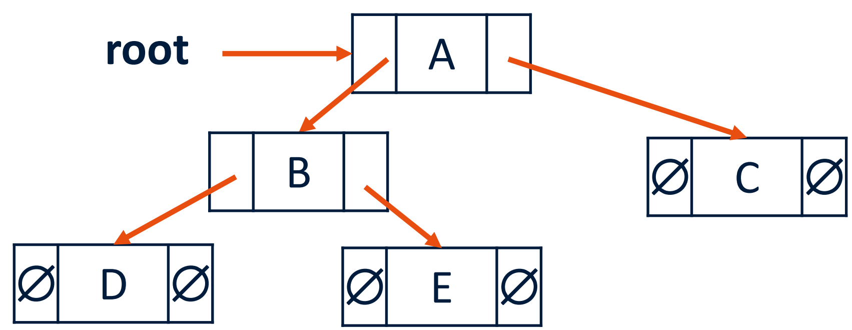
A	B	C	D	E	∅	∅	∅	**F**	∅	∅	∅	∅	∅	∅	∅
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

- When we resize we need to make space for a whole level
- Possibly many empty locations
  - What is the worst case?

Linked-Based (Binary) Tree

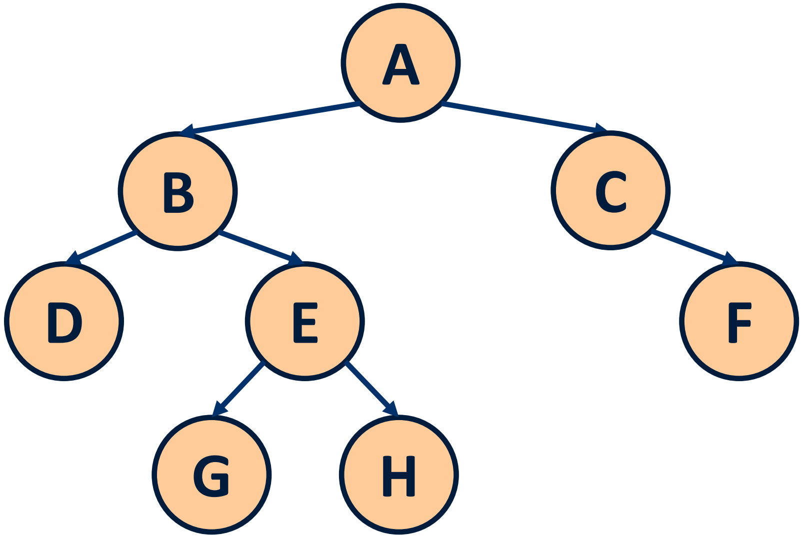
- Nodes are represented by objects storing:
  - Element
  - (optional) Parent node
  - Children
    - Left and right for binary trees
    - A list of elements for generic trees





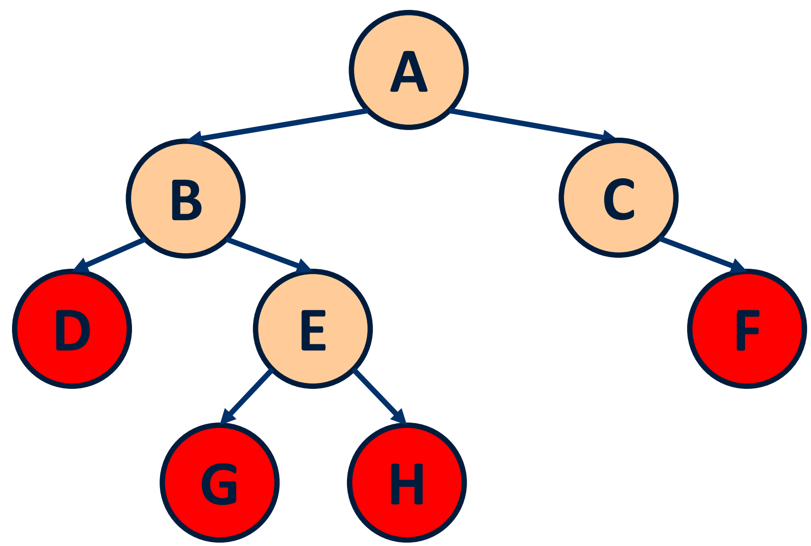
Removing Nodes from Binary Tree

- Depending on the node we want to delete:



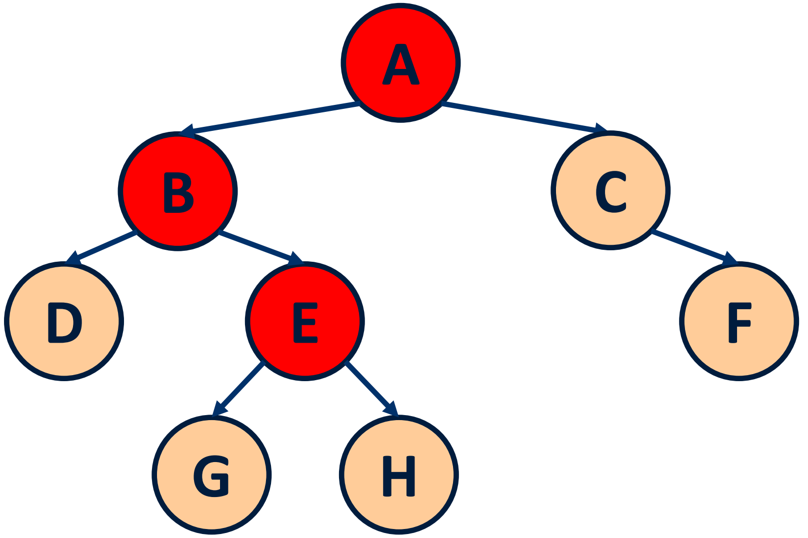
Removing Nodes from Binary Tree

- Depending on the node we want to delete:
  - Leafs: we can remove them



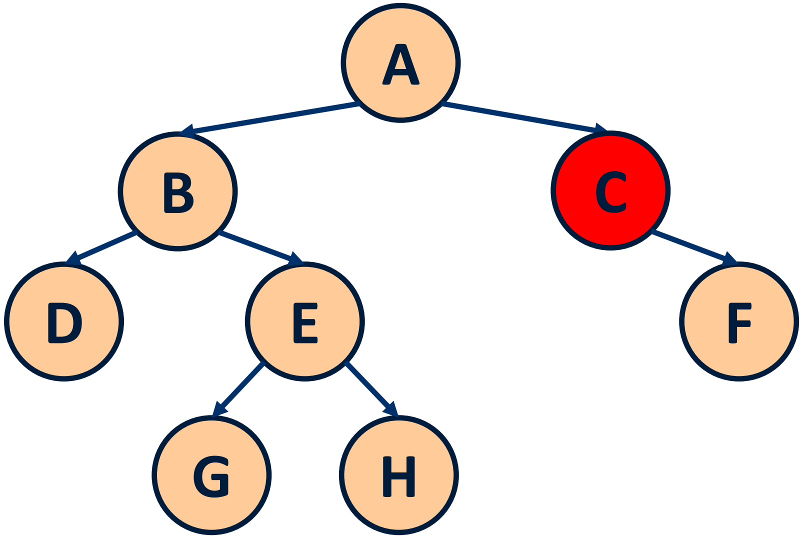
Removing Nodes from Binary Tree

- Depending on the node we want to delete:
  - Leafs: we can remove them
  - Internal node with two children:
    we cannot remove them



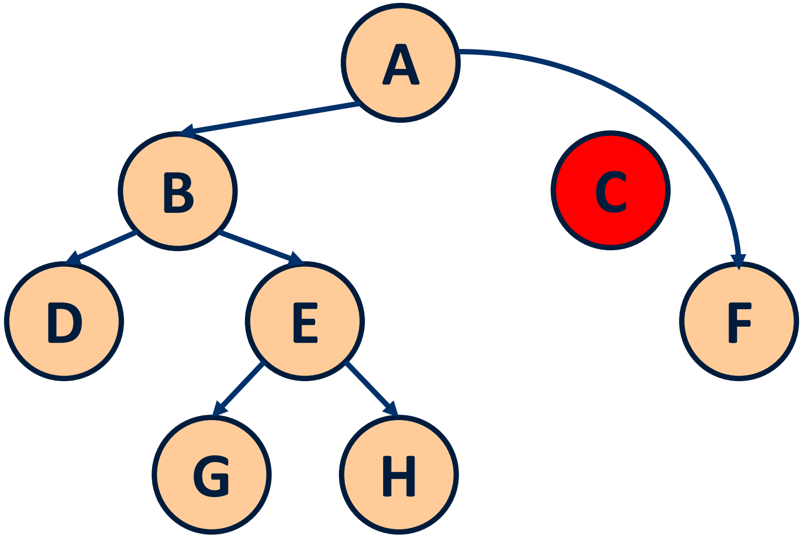
Removing Nodes from Binary Tree

- Depending on the node we want to delete:
  - Leafs: we can remove them
  - Internal node with two children:
    we cannot remove them
  - Internal node with one child:
    we remove it and assign the child to the parent



Removing Nodes from Binary Tree

- Depending on the node we want to delete:
  - Leafs: we can remove them
  - Internal node with two children: we cannot remove them
  - Internal node with one child: we remove it and assign the child to the parent



Tree Traversal

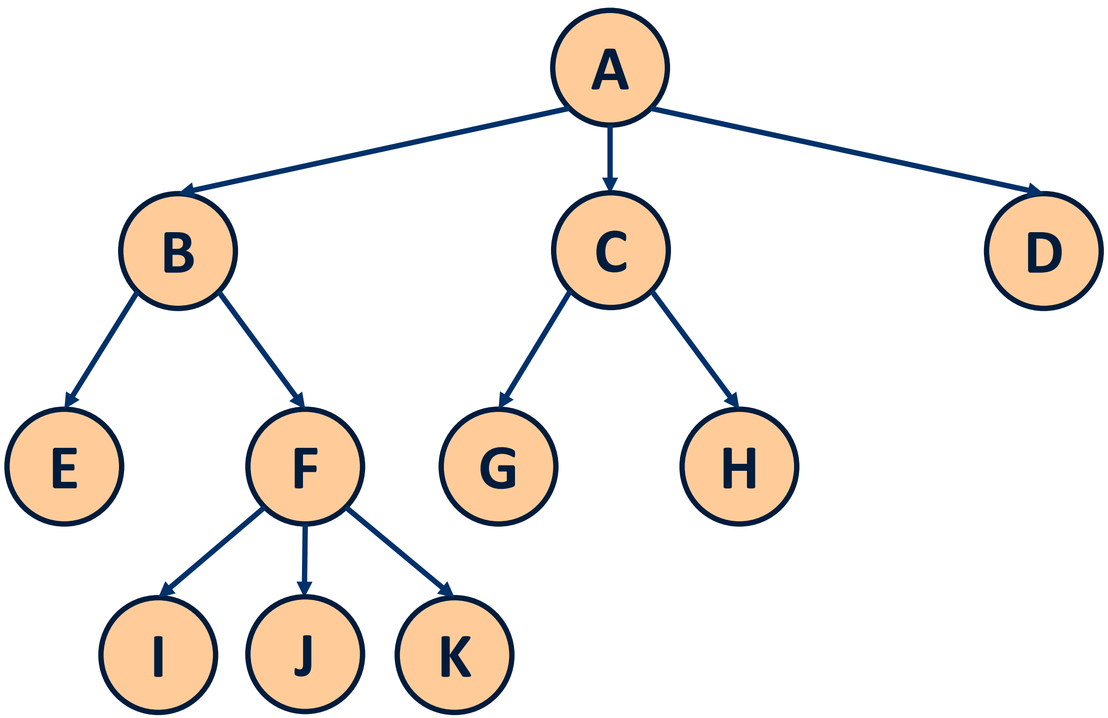
Tree Traversal

- Different ways to traverse a tree

- Goal: start at root, visit all nodes in the tree
  - Pre-order
  - Post-order
  - In-order (**only** for binary trees)

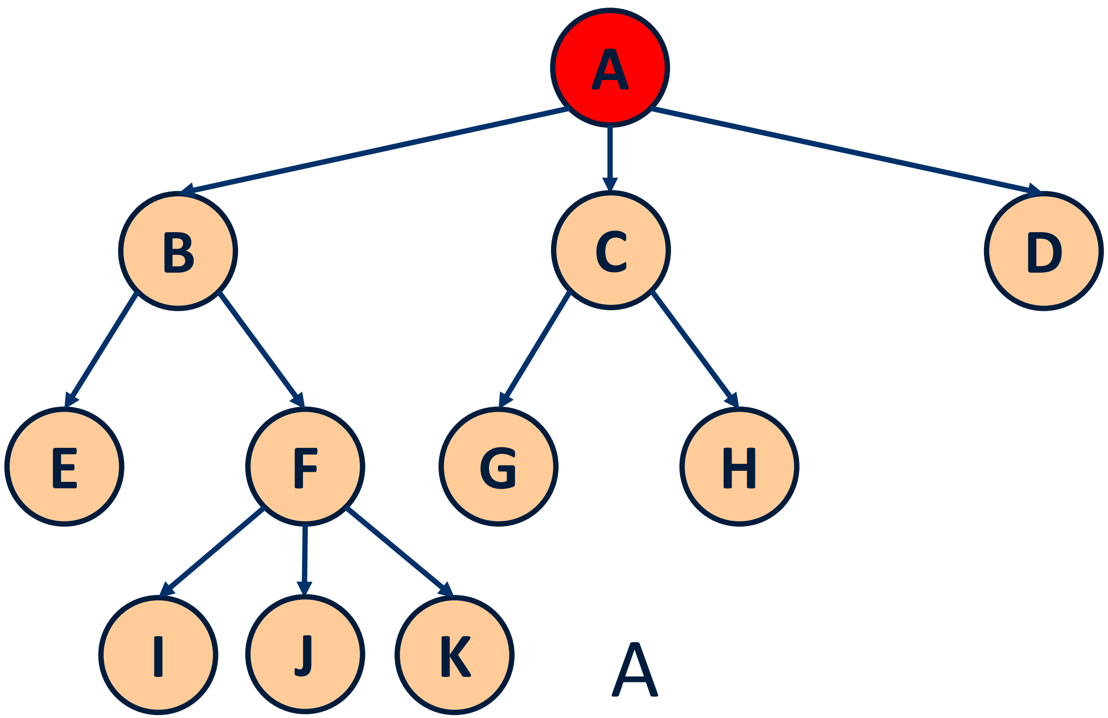
Pre-order Traversal

- In a preorder traversal, a node is visited before its descendants



Pre-order Traversal

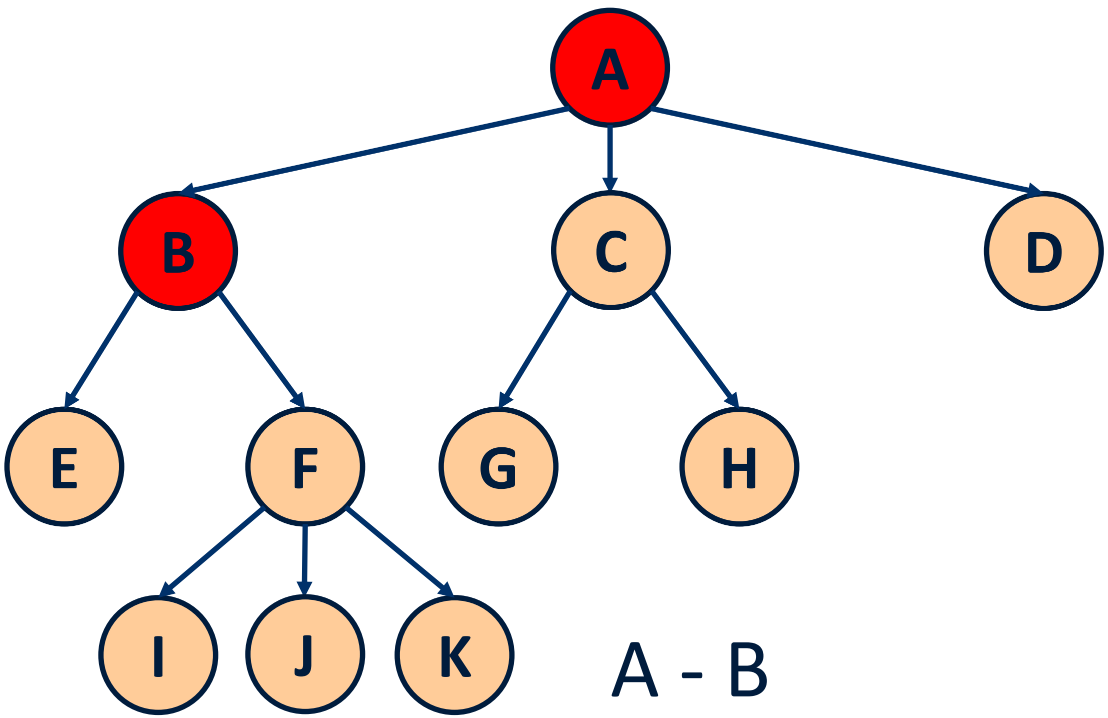
- In a preorder traversal, a node is visited before its descendants



A

Pre-order Traversal

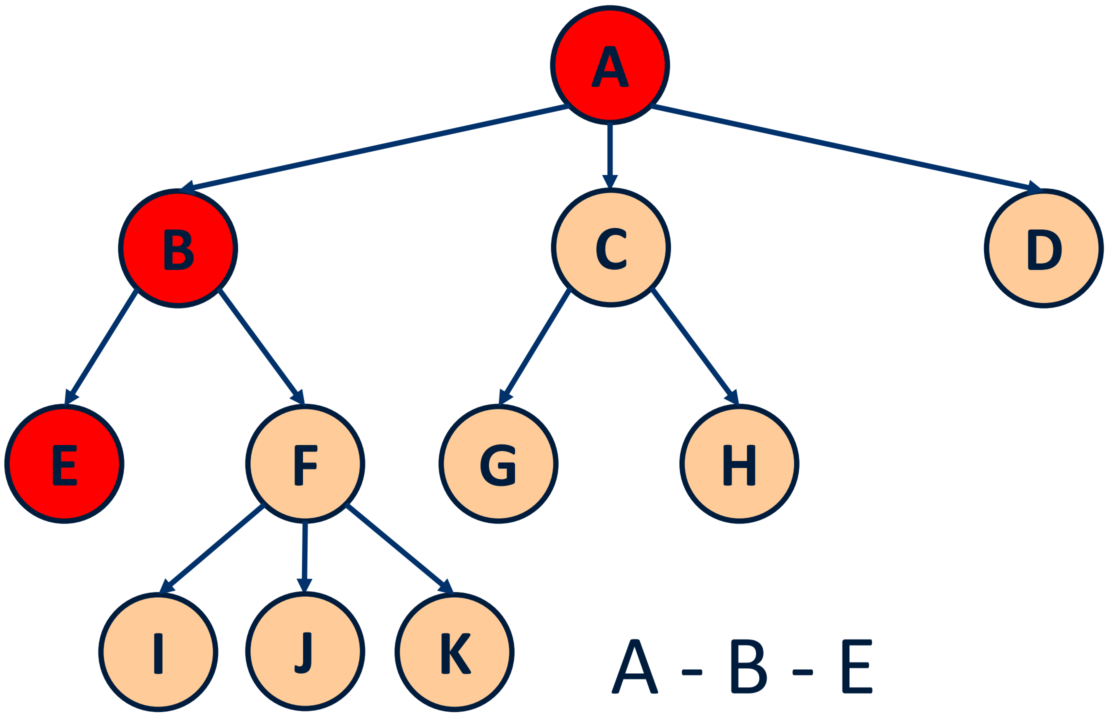
- In a preorder traversal, a node is visited before its descendants



A - B

Pre-order Traversal

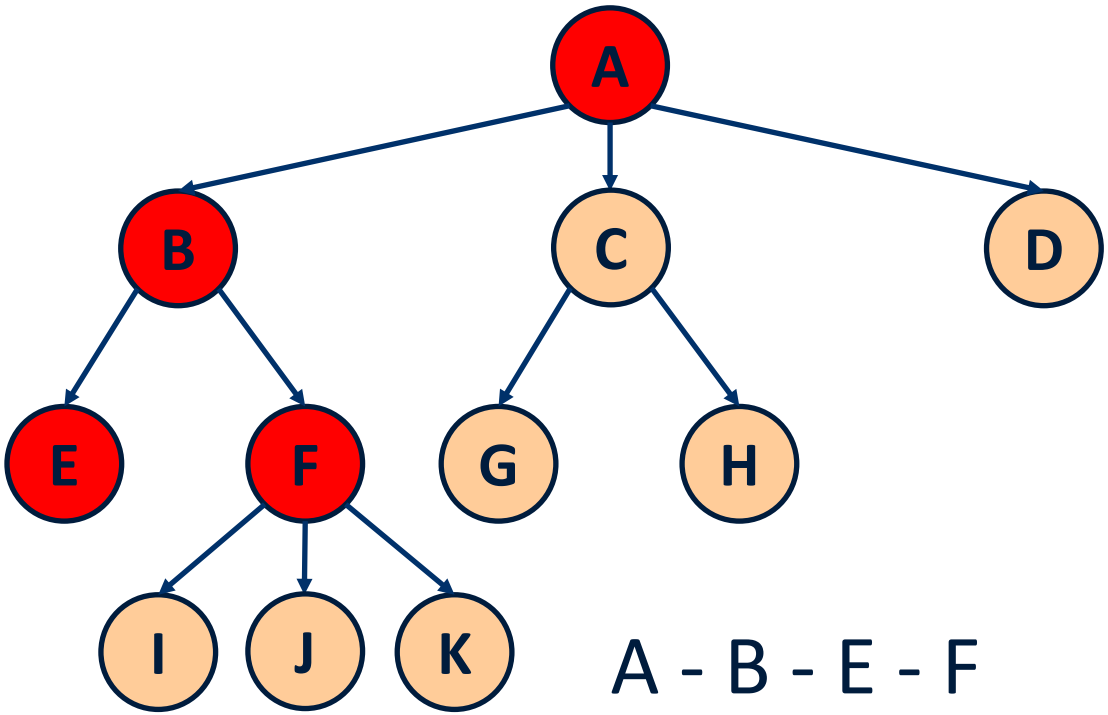
- In a preorder traversal, a node is visited before its descendants



A - B - E

Pre-order Traversal

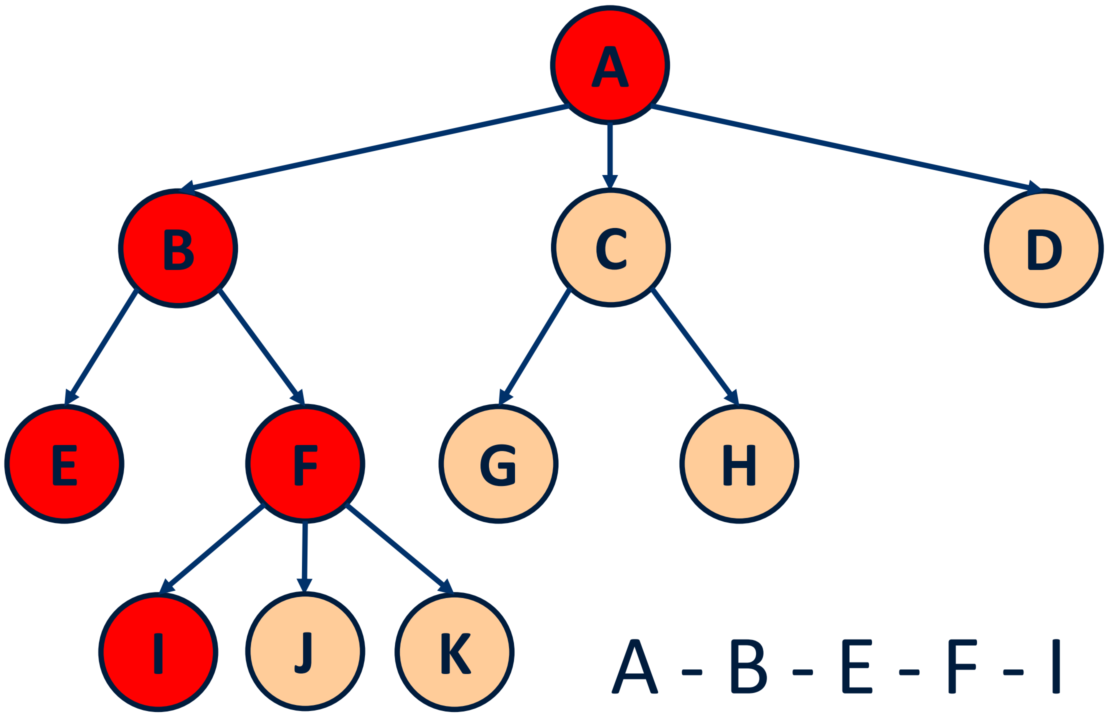
- In a preorder traversal, a node is visited before its descendants



A - B - E - F

Pre-order Traversal

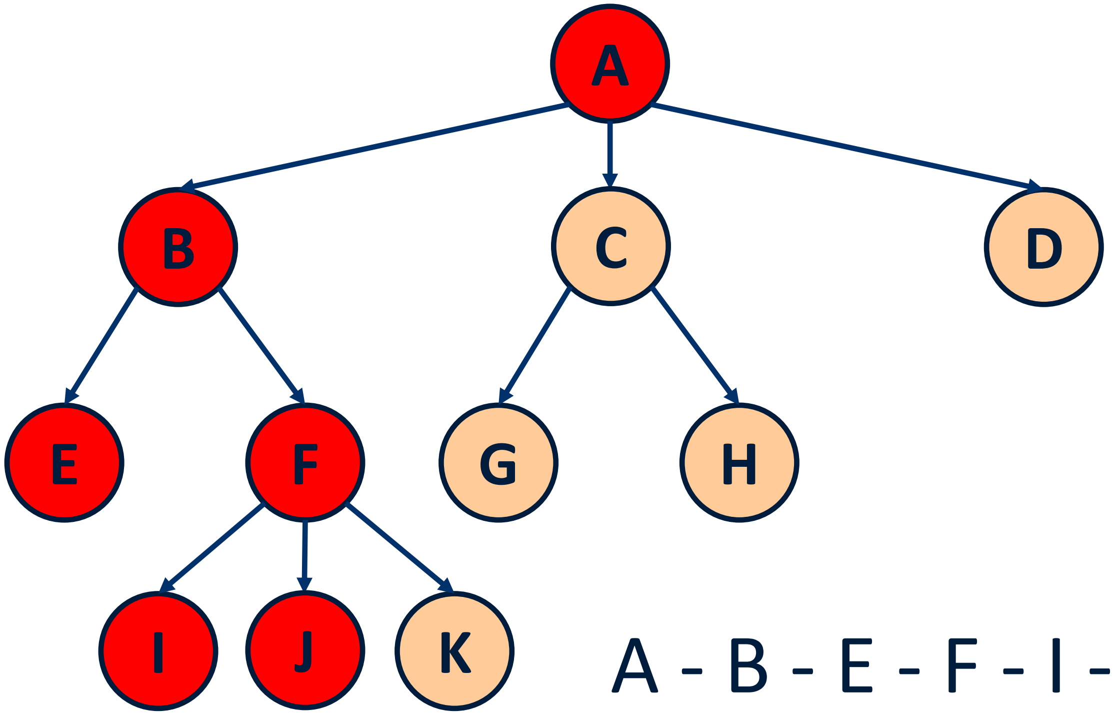
- In a preorder traversal, a node is visited before its descendants



A - B - E - F - I

Pre-order Traversal

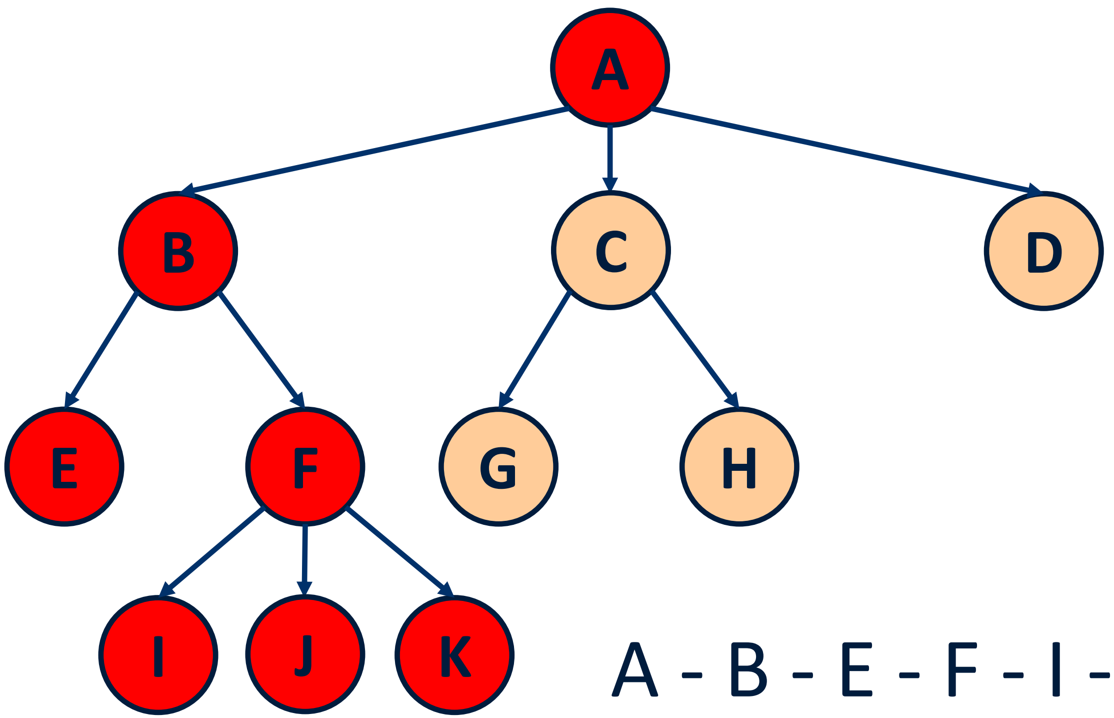
- In a preorder traversal, a node is visited before its descendants



A - B - E - F - I - J

Pre-order Traversal

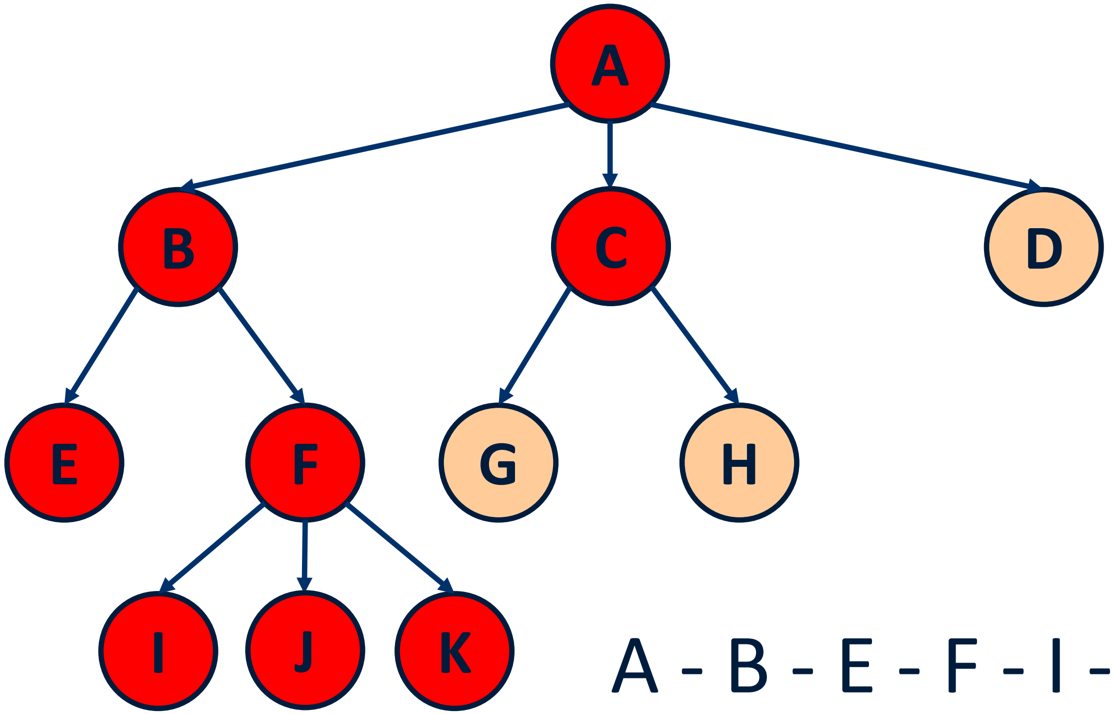
- In a preorder traversal, a node is visited before its descendants



A - B - E - F - I - J - K

Pre-order Traversal

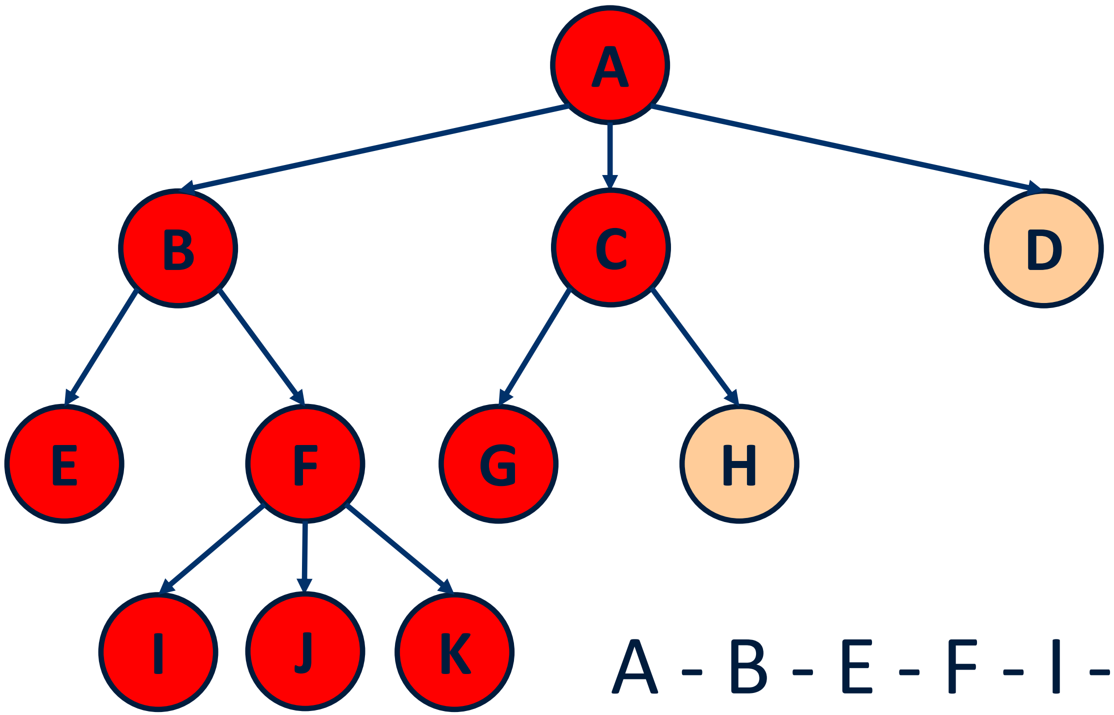
- In a preorder traversal, a node is visited before its descendants



A - B - E - F - I - J - K - C

Pre-order Traversal

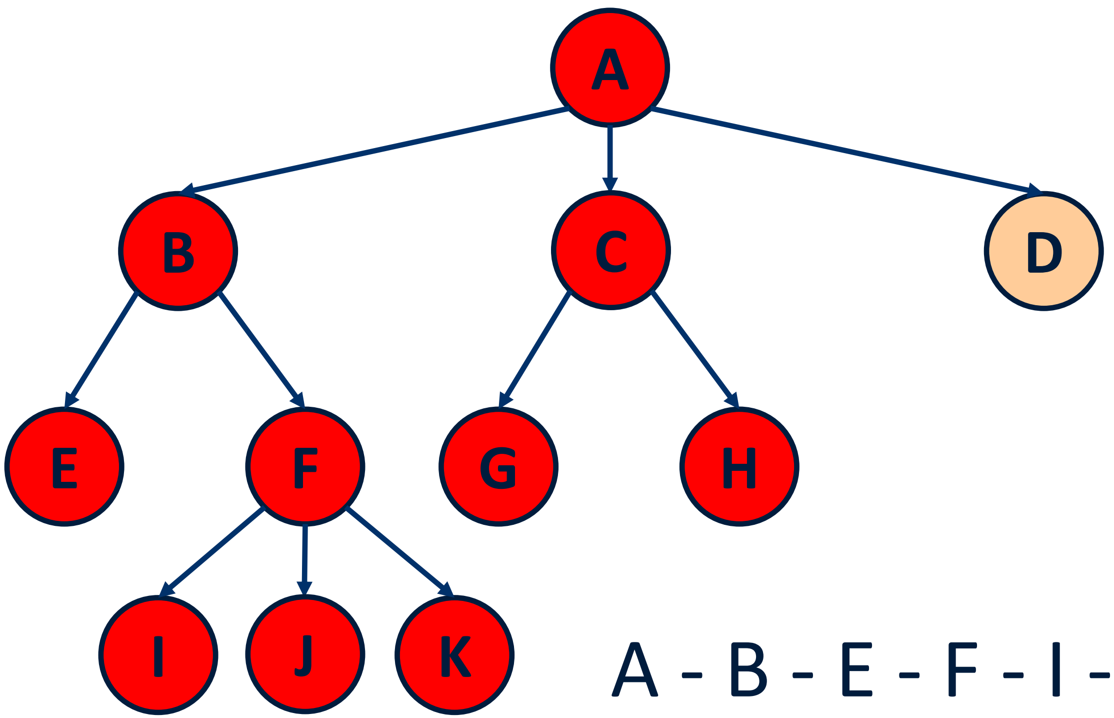
- In a preorder traversal, a node is visited before its descendants



A - B - E - F - I - J - K - C - G

Pre-order Traversal

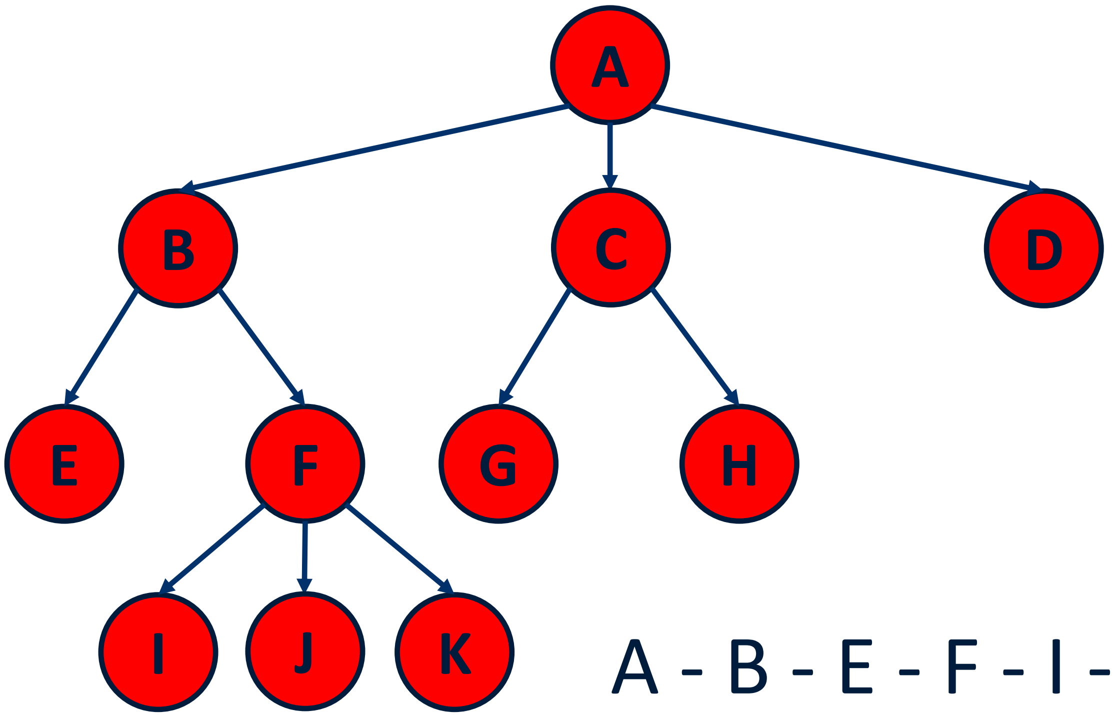
- In a preorder traversal, a node is visited before its descendants



A - B - E - F - I - J - K - C - G - H

Pre-order Traversal

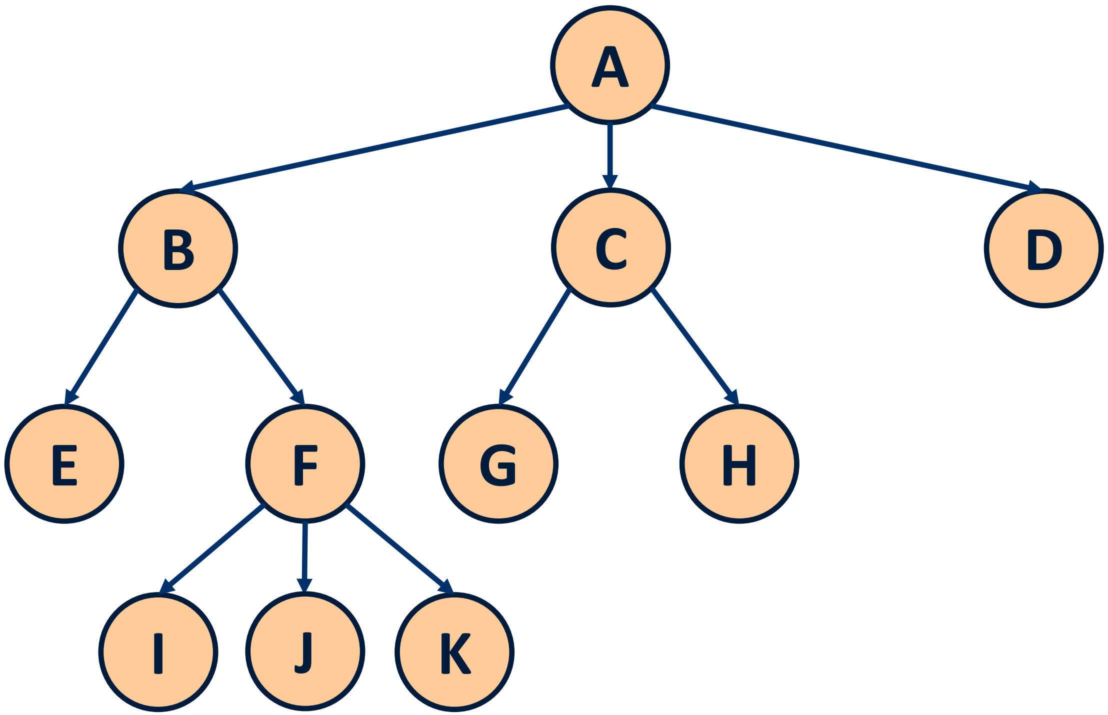
- In a preorder traversal, a node is visited before its descendants



A - B - E - F - I - J - K - C - G - H - D

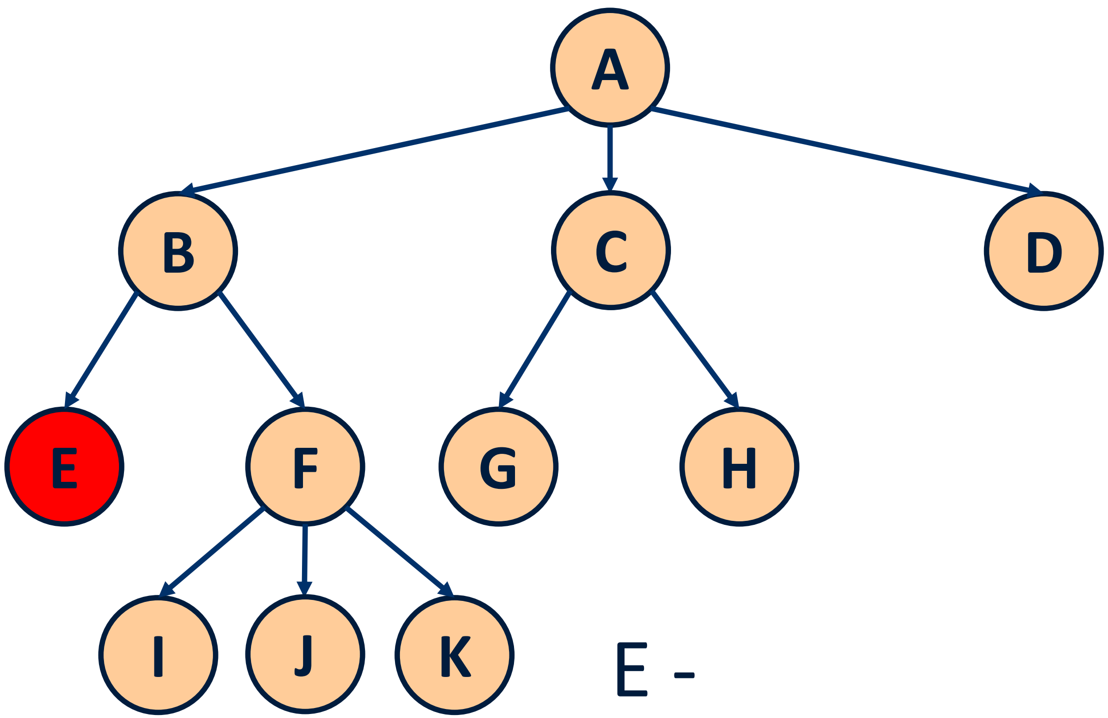
Post-order Traversal

- In a postorder traversal, a node is visited after its descendants



Post-order Traversal

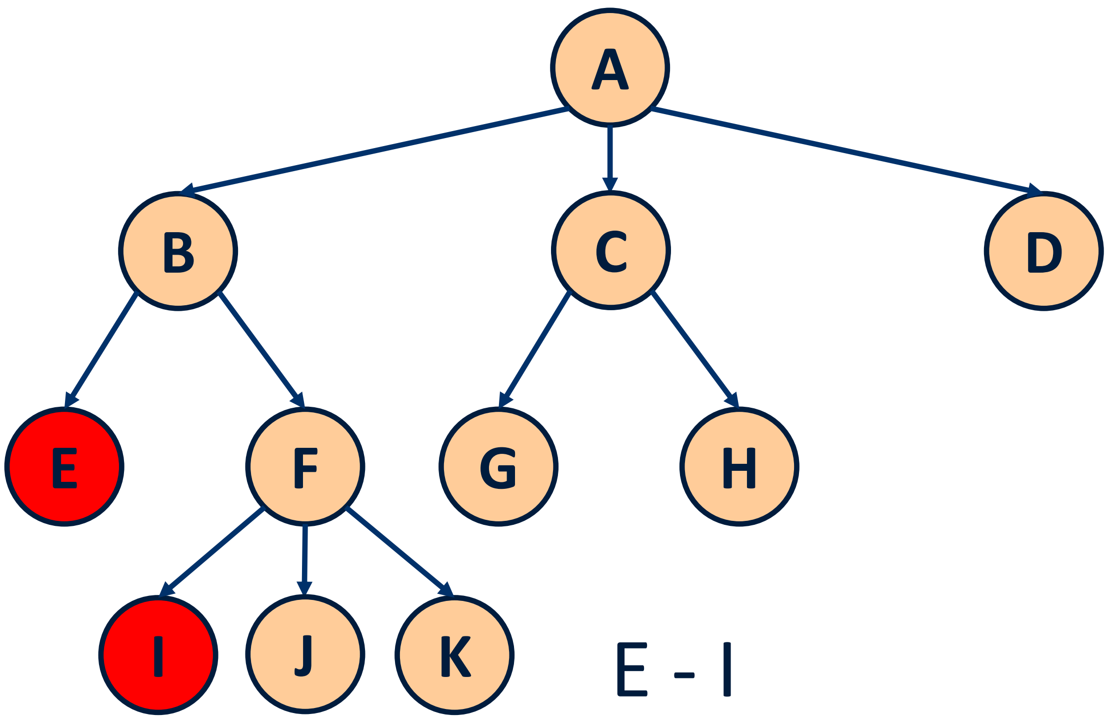
- In a postorder traversal, a node is visited after its descendants



E -

Post-order Traversal

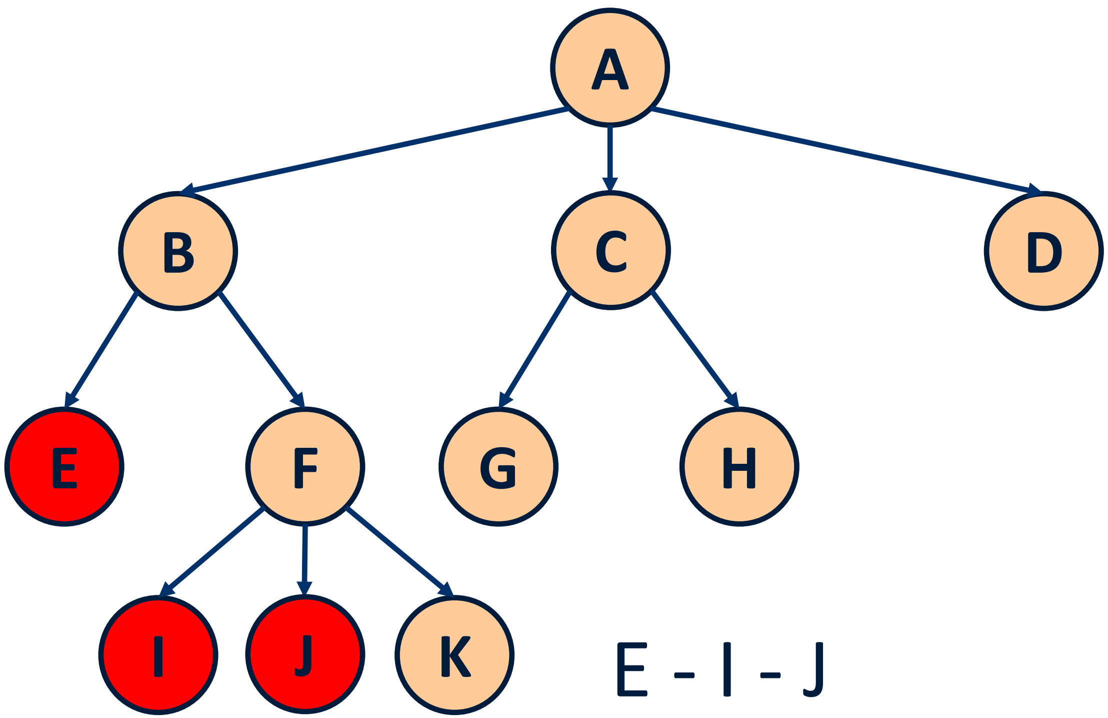
- In a postorder traversal, a node is visited after its descendants



E - I

Post-order Traversal

- In a postorder traversal, a node is visited after its descendants

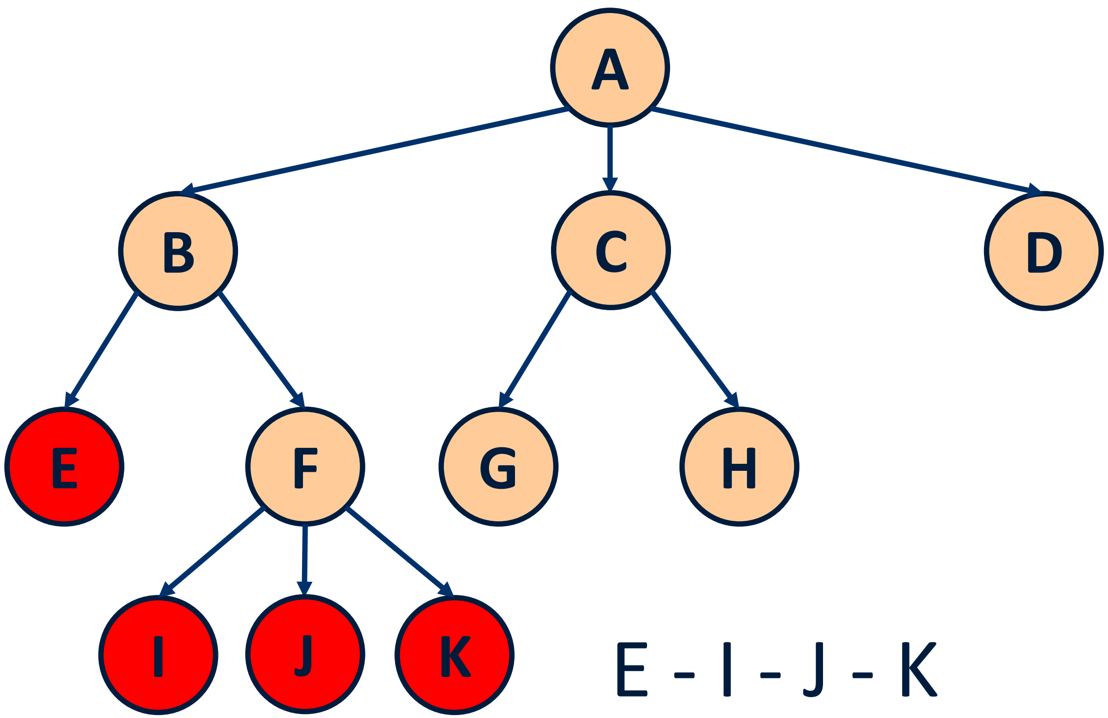


E - I - J

Maastricht University

Post-order Traversal

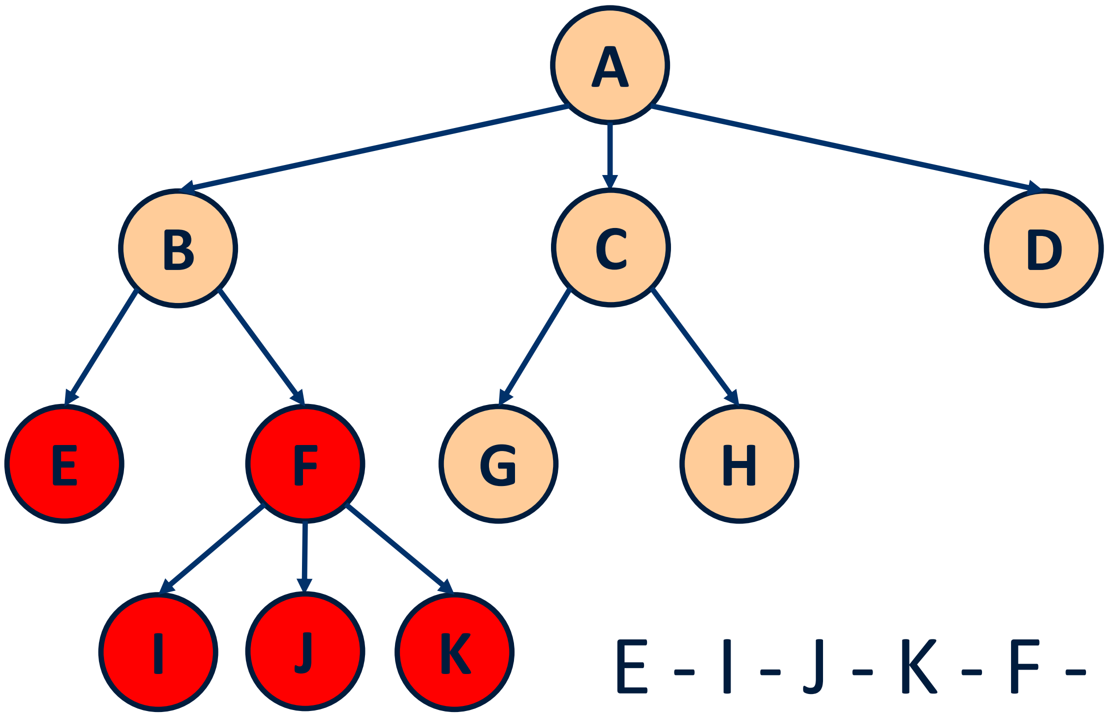
- In a postorder traversal, a node is visited after its descendants



E - I - J - K

Post-order Traversal

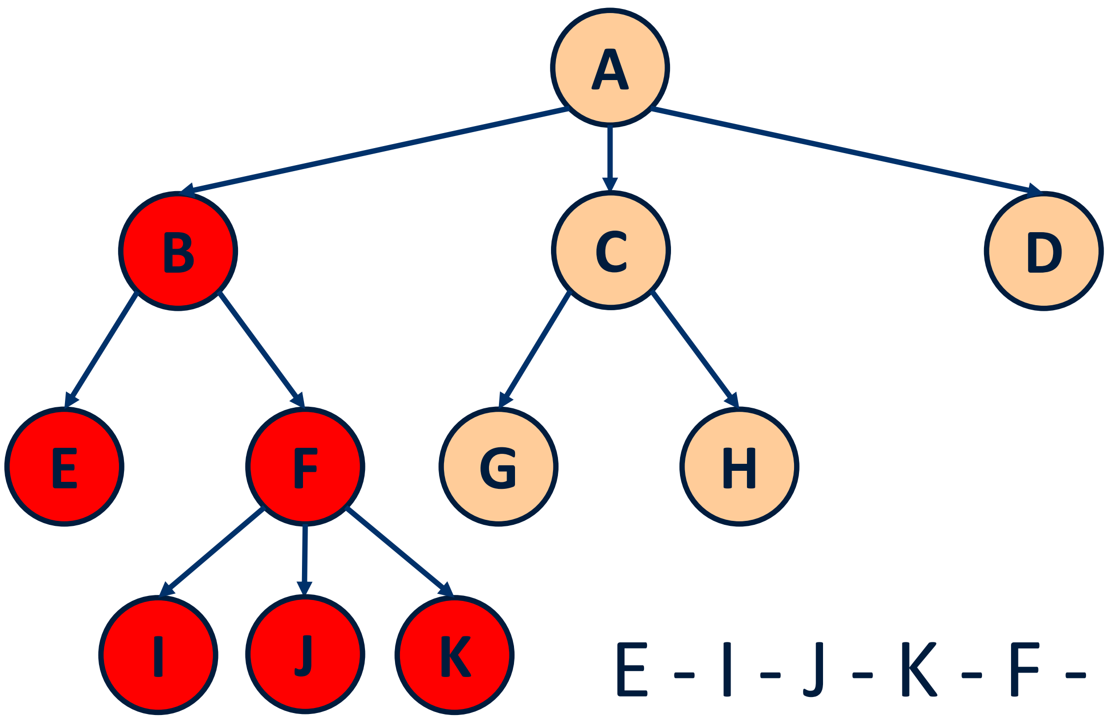
- In a postorder traversal, a node is visited after its descendants



E - I - J - K - F -

Post-order Traversal

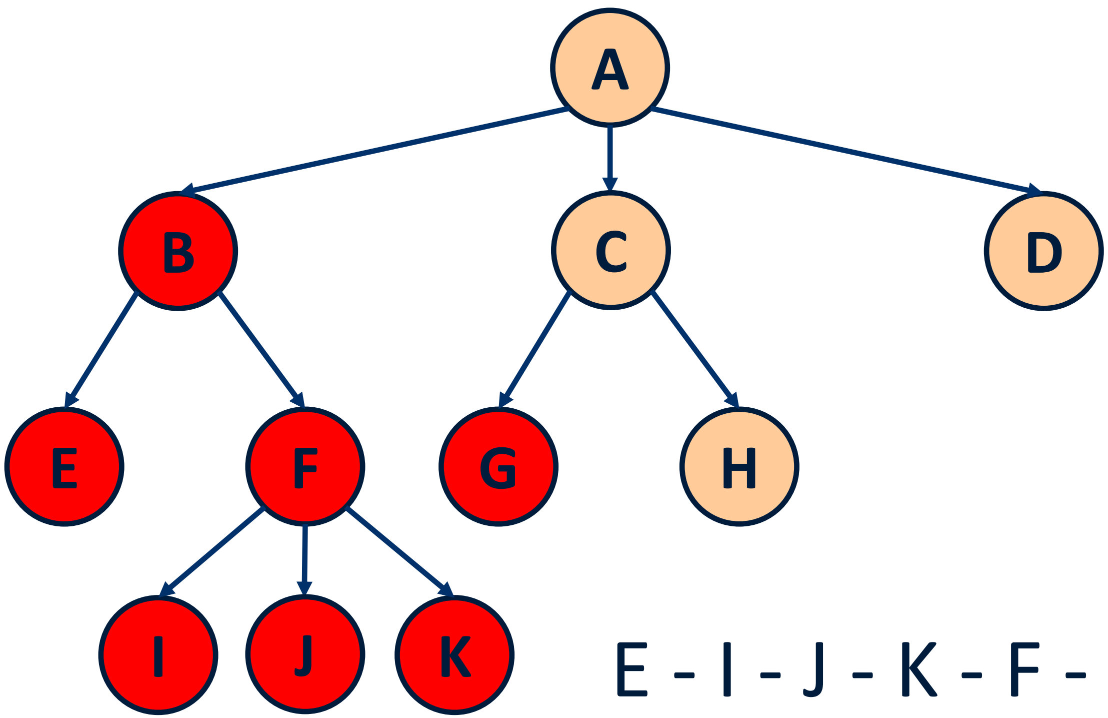
- In a postorder traversal, a node is visited after its descendants



E - I - J - K - F - B -

Post-order Traversal

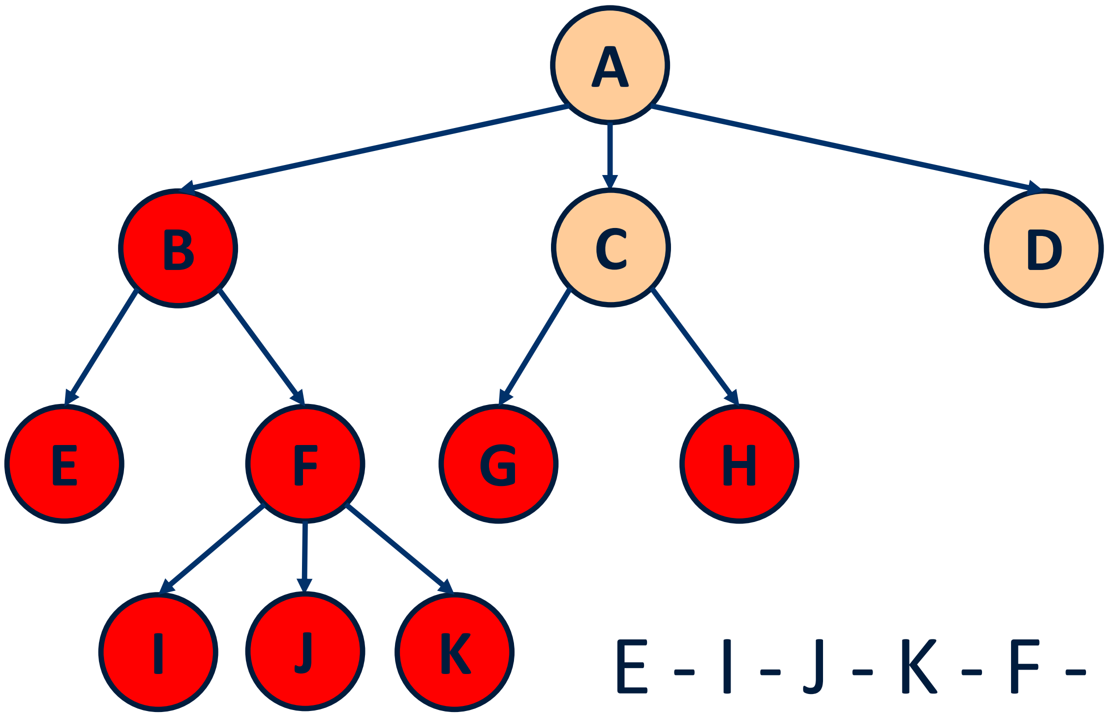
- In a postorder traversal, a node is visited after its descendants



E - I - J - K - F - B - G

Post-order Traversal

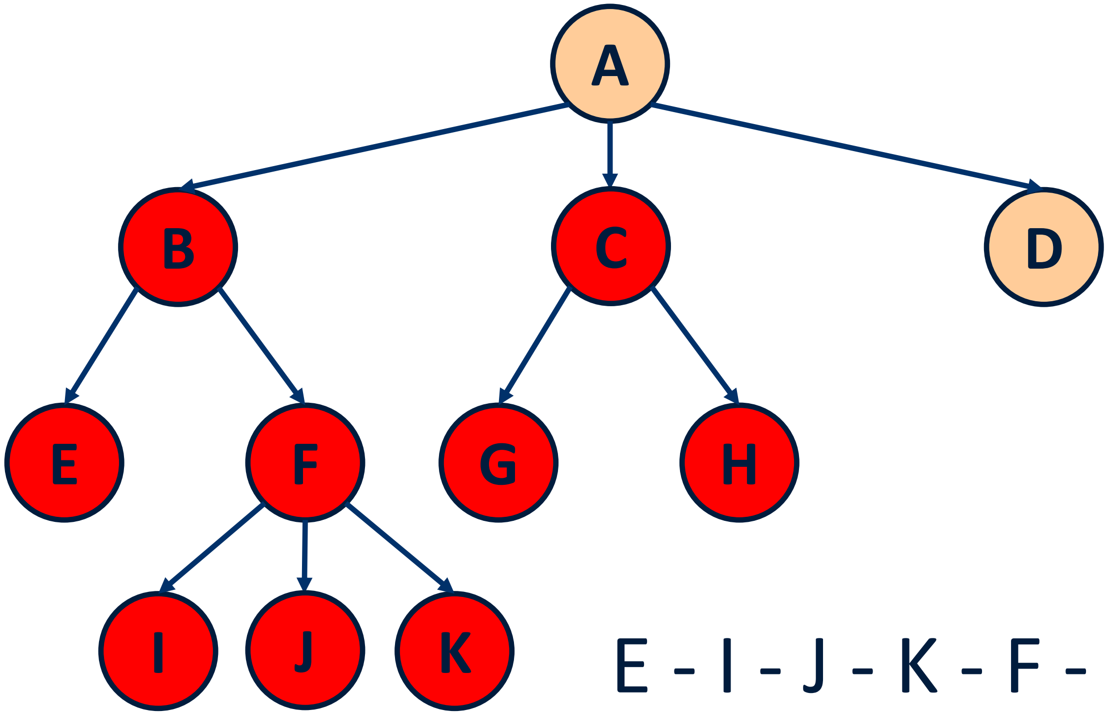
- In a postorder traversal, a node is visited after its descendants



E - I - J - K - F - B - G - H

Post-order Traversal

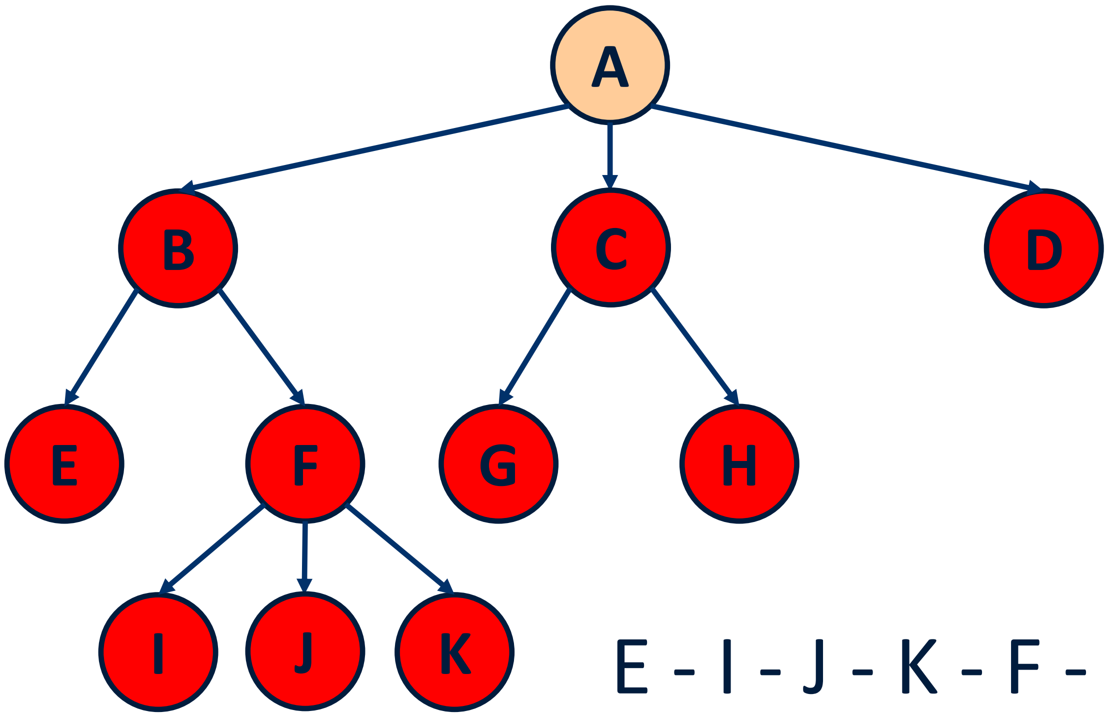
- In a postorder traversal, a node is visited after its descendants



E - I - J - K - F - B - G - H - C

Post-order Traversal

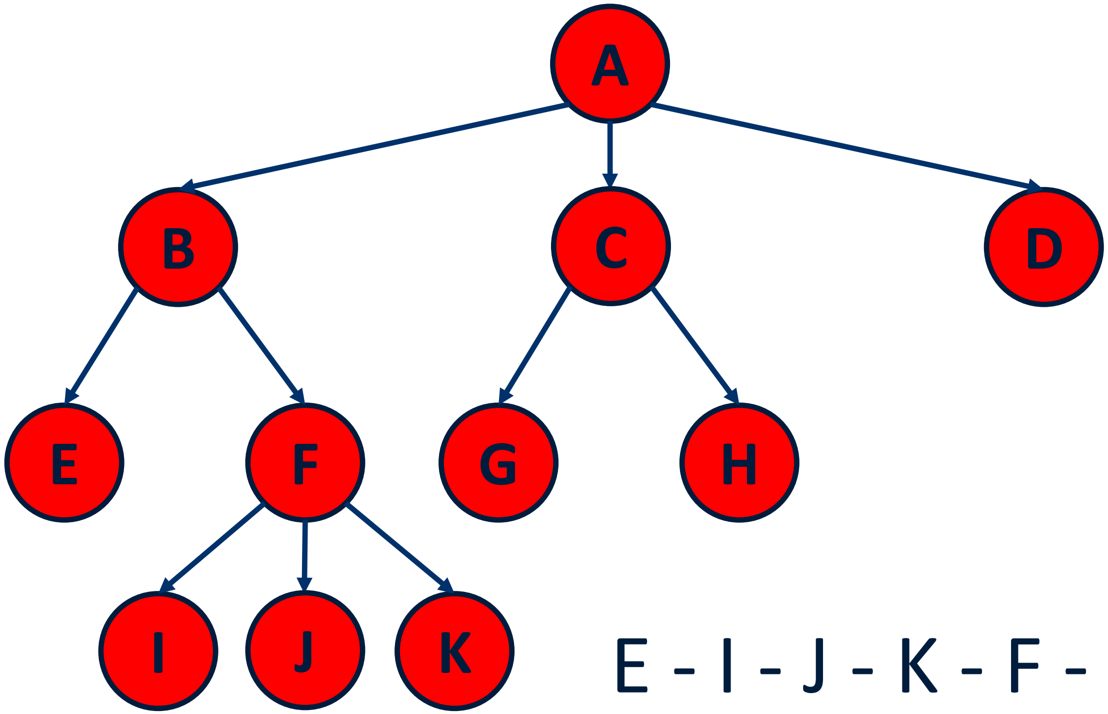
- In a postorder traversal, a node is visited after its descendants



E - I - J - K - F - B - G - H - C - D

Post-order Traversal

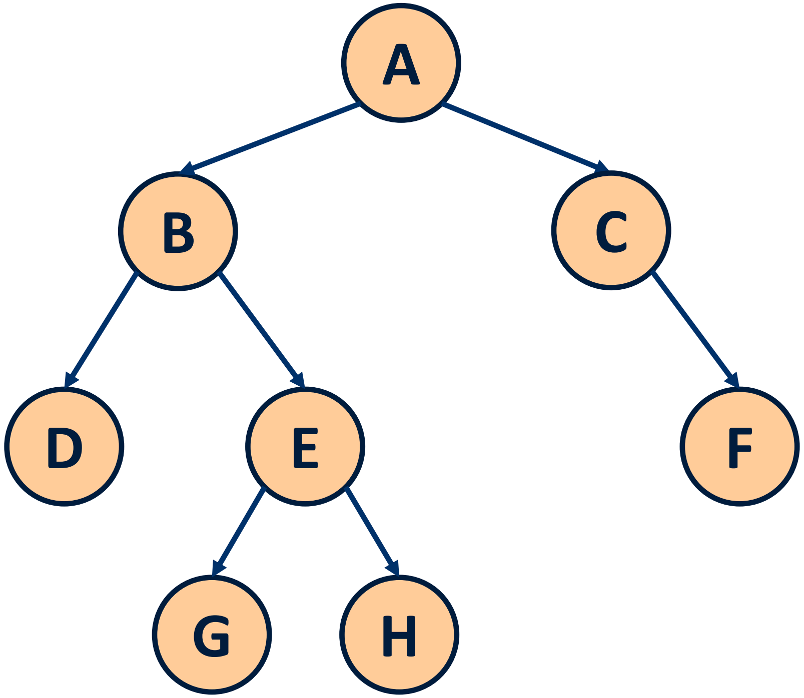
- In a postorder traversal, a node is visited after its descendants



E - I - J - K - F - B - G - H - C - D - A

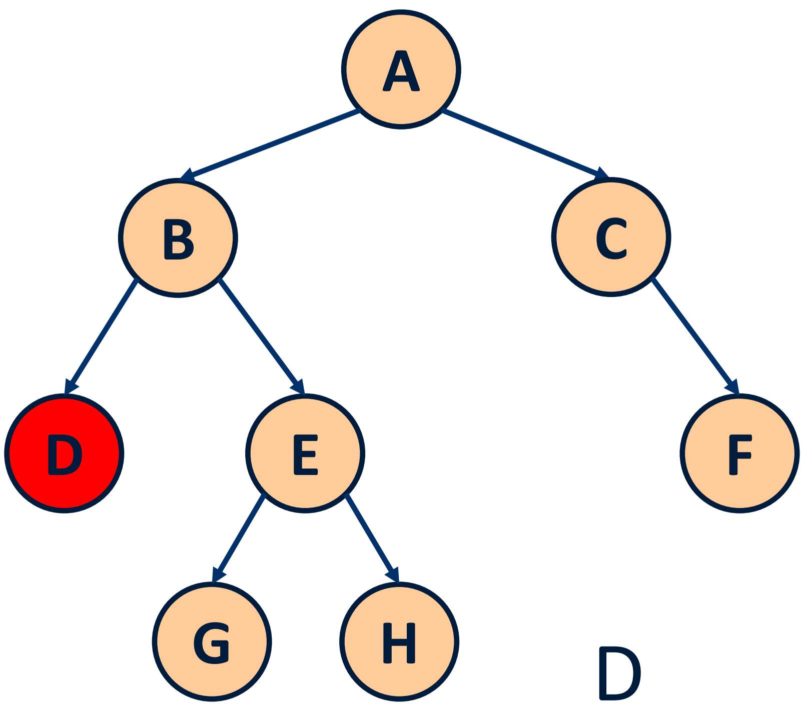
In-order Traversal (ONLY for binary trees)

- In an inorder traversal a node is visited after its left subtree and before its right subtree



In-order Traversal (ONLY for binary trees)

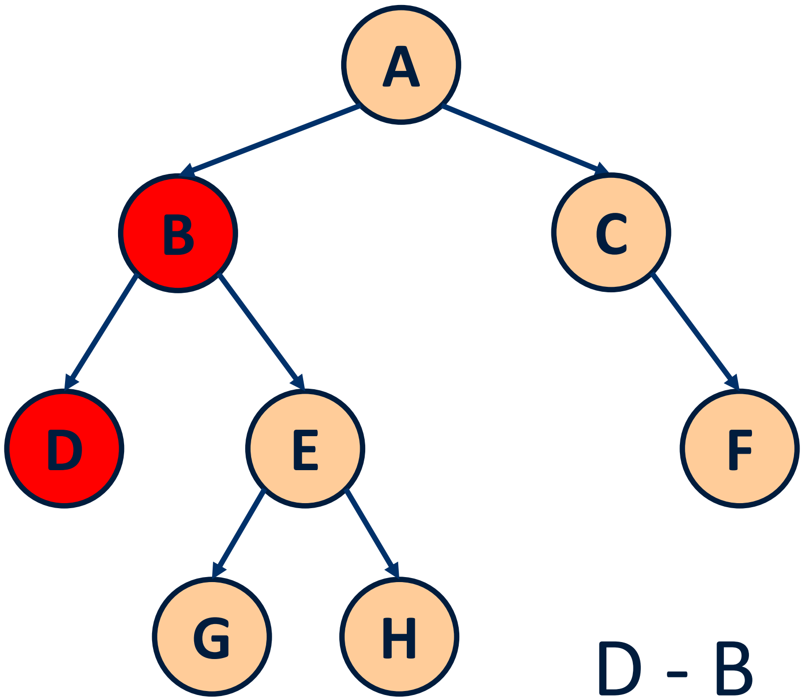
- In an inorder traversal a node is visited after its left subtree and before its right subtree



D

In-order Traversal (ONLY for binary trees)

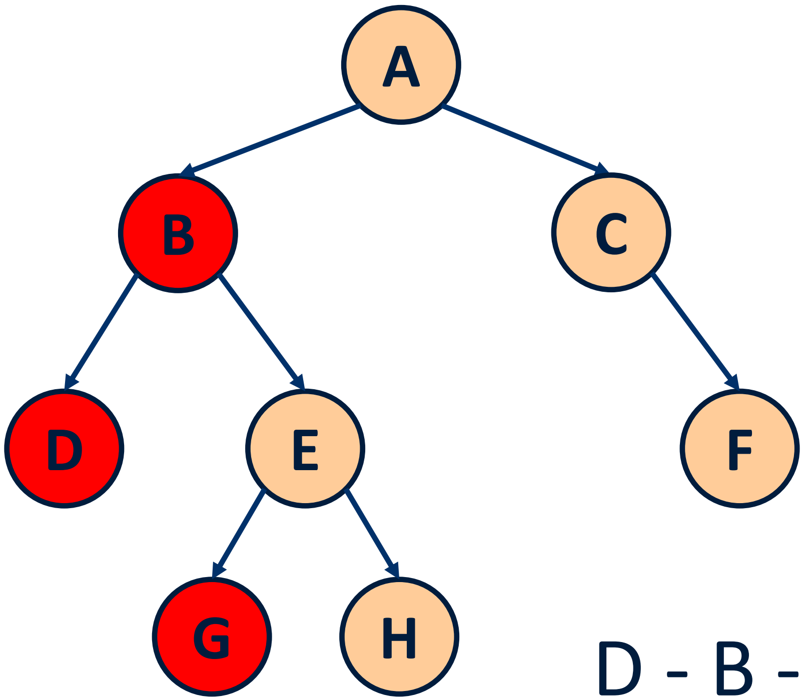
- In an inorder traversal a node is visited after its left subtree and before its right subtree



D - B

In-order Traversal (ONLY for binary trees)

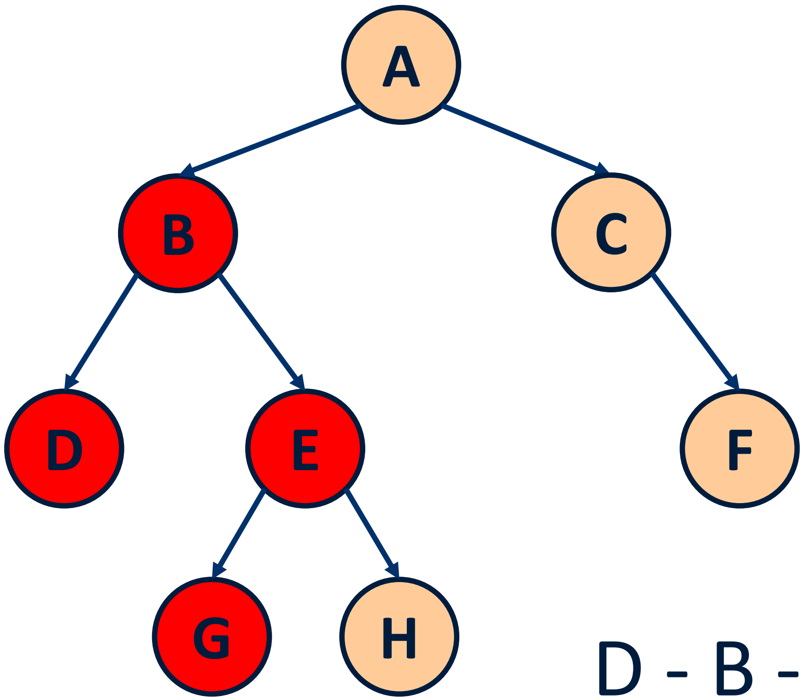
- In an inorder traversal a node is visited after its left subtree and before its right subtree



D - B - G

In-order Traversal (ONLY for binary trees)

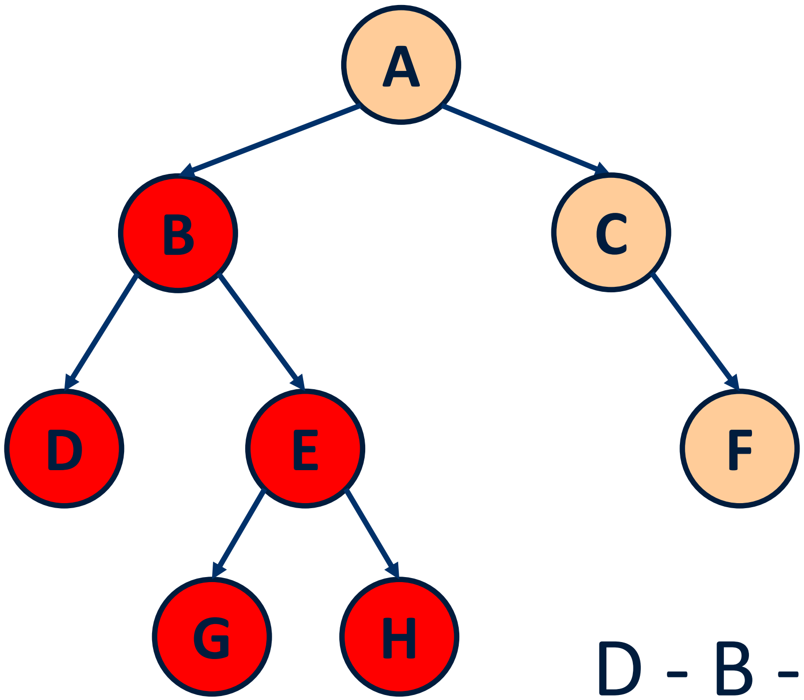
- In an inorder traversal a node is visited after its left subtree and before its right subtree



D - B - G - E

In-order Traversal (ONLY for binary trees)

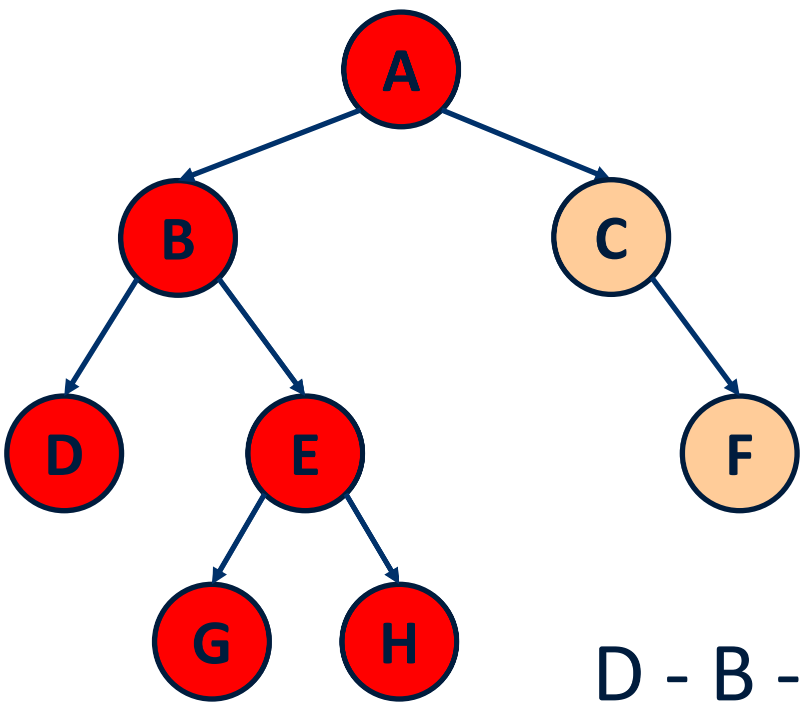
- In an inorder traversal a node is visited after its left subtree and before its right subtree



D - B - G - E - H

In-order Traversal (ONLY for binary trees)

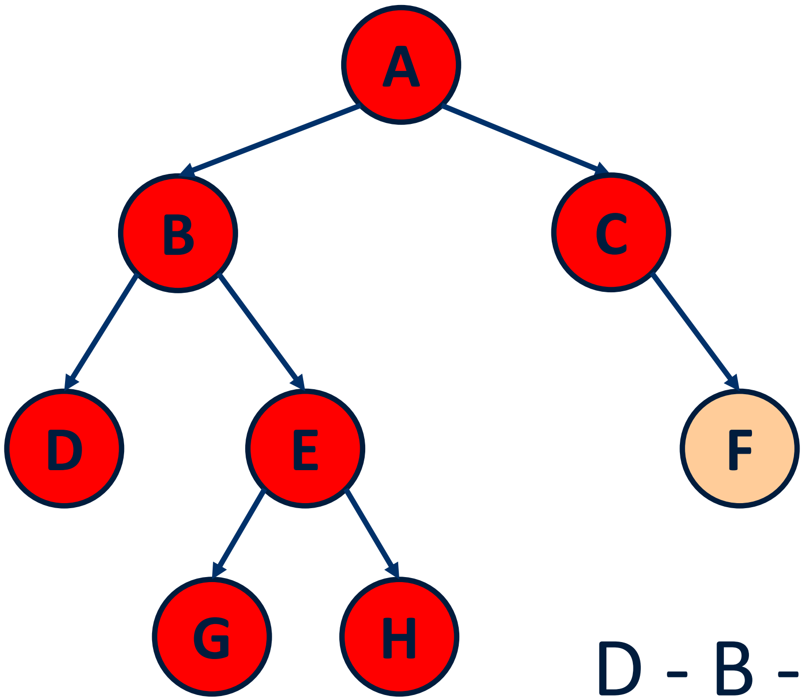
- In an inorder traversal a node is visited after its left subtree and before its right subtree



D - B - G - E - H - A

In-order Traversal (ONLY for binary trees)

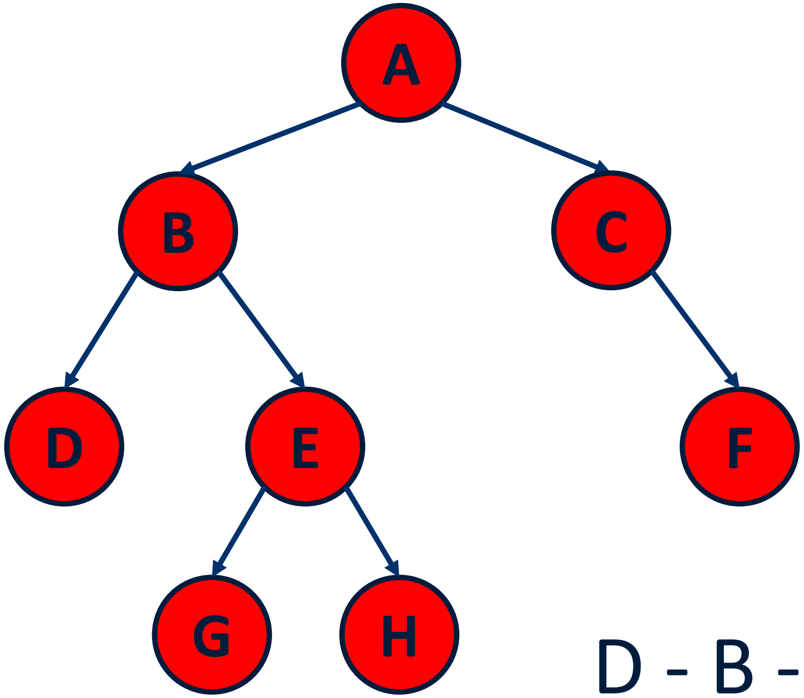
- In an inorder traversal a node is visited after its left subtree and before its right subtree



D - B - G - E - H - A - C

In-order Traversal (ONLY for binary trees)

- In an inorder traversal a node is visited after its left subtree and before its right subtree



D - B - G - E - H - A - C - F

Designing Algorithms for Trees

Maastricht University

Programming with Trees

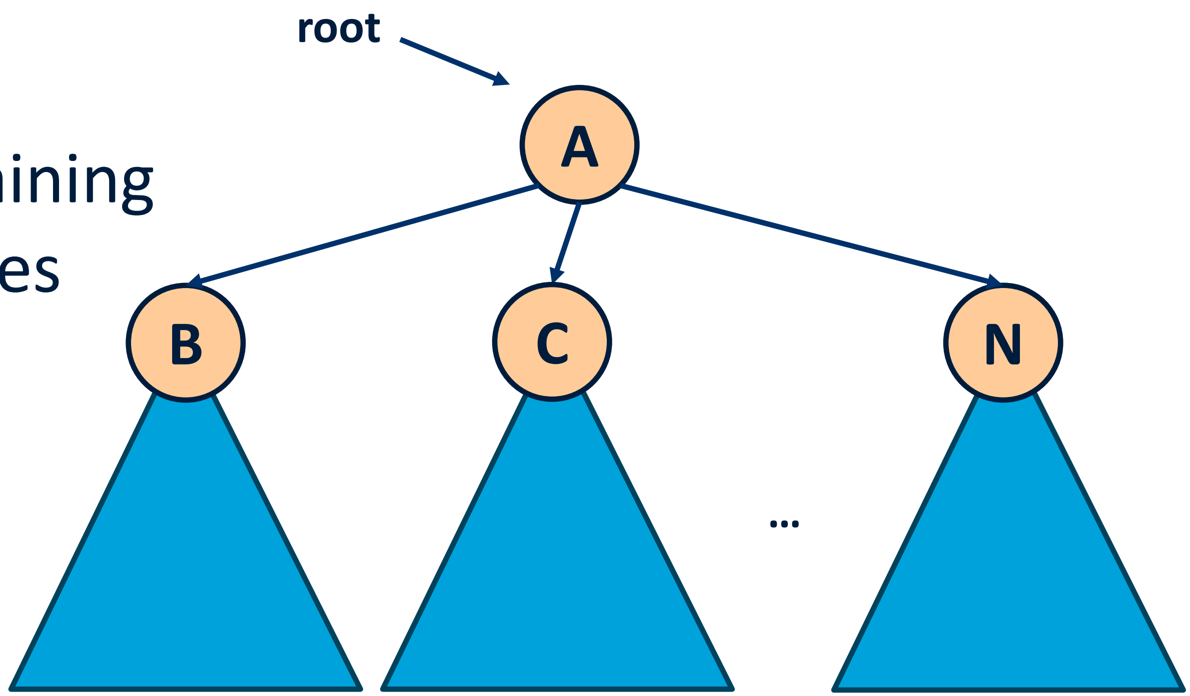
- Many tree algorithms are recursive
  - Use the recursive definition of a tree

- A tree is
  - An empty tree

Programming with Trees

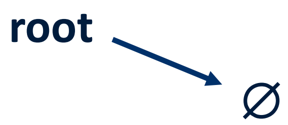
- Many tree algorithms are recursive
  - Use the recursive definition of a tree

- A tree is
  - An empty tree
  - A node maintaining a list of subtrees



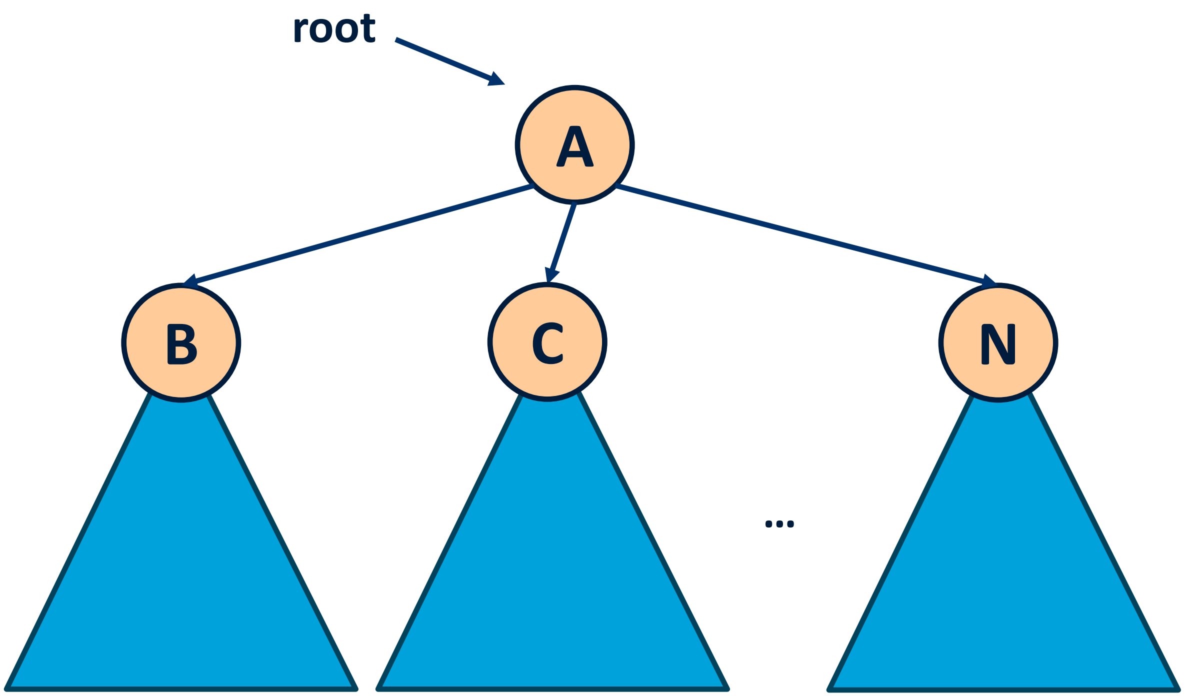
Programming with Trees

- **Pre-order** visit
  - If the node is null -> nothing to do



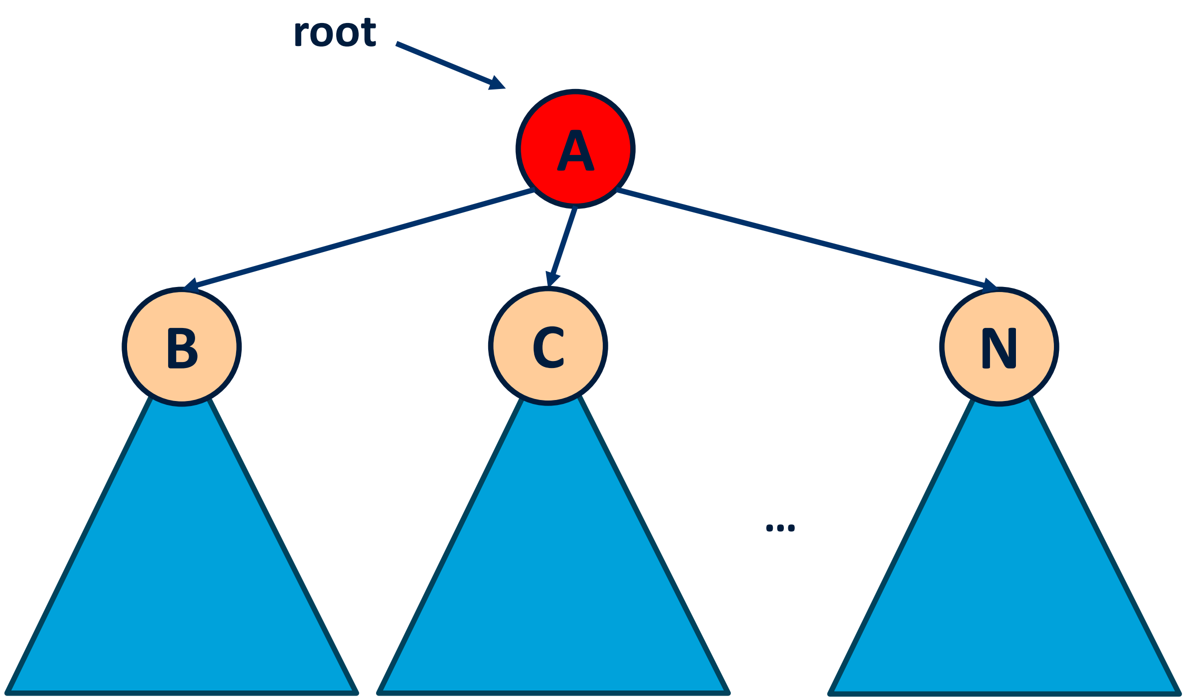
Programming with Trees

- **Pre-order** visit
  - If the node is null -> nothing to do
  - If the node is not null



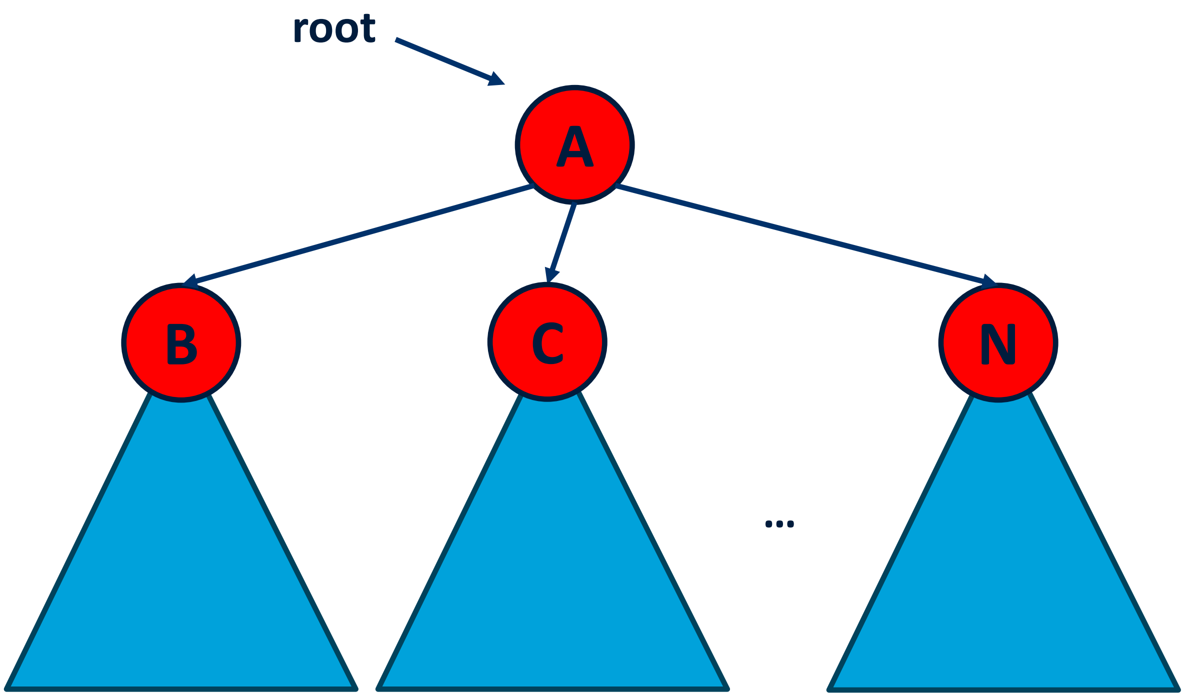
Programming with Trees

- **Pre-order** visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit the node



Programming with Trees

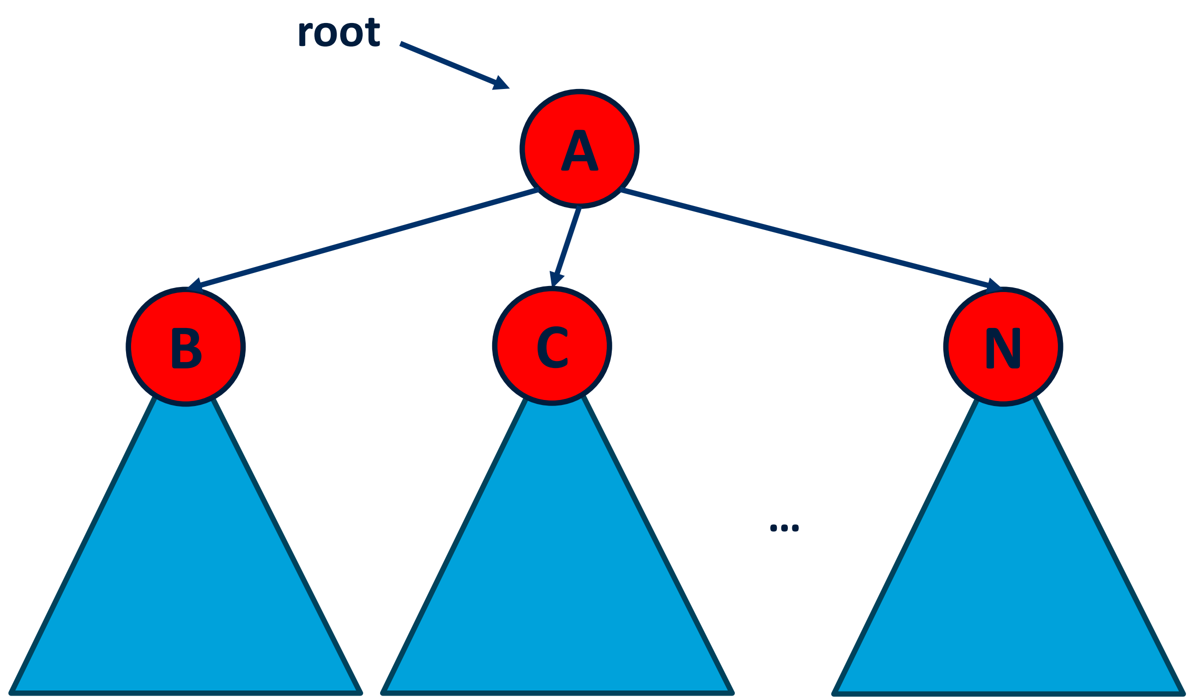
- **Pre-order** visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit the node
    - Visit recursively all the children one by one



Programming with Trees

- **Pre-order** visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit the node
    - Visit recursively all the children one by one

```
preOrder(v)
    if v ≠ ∅
        visit(v)
        for each child w of v
            preOrder(w)
```

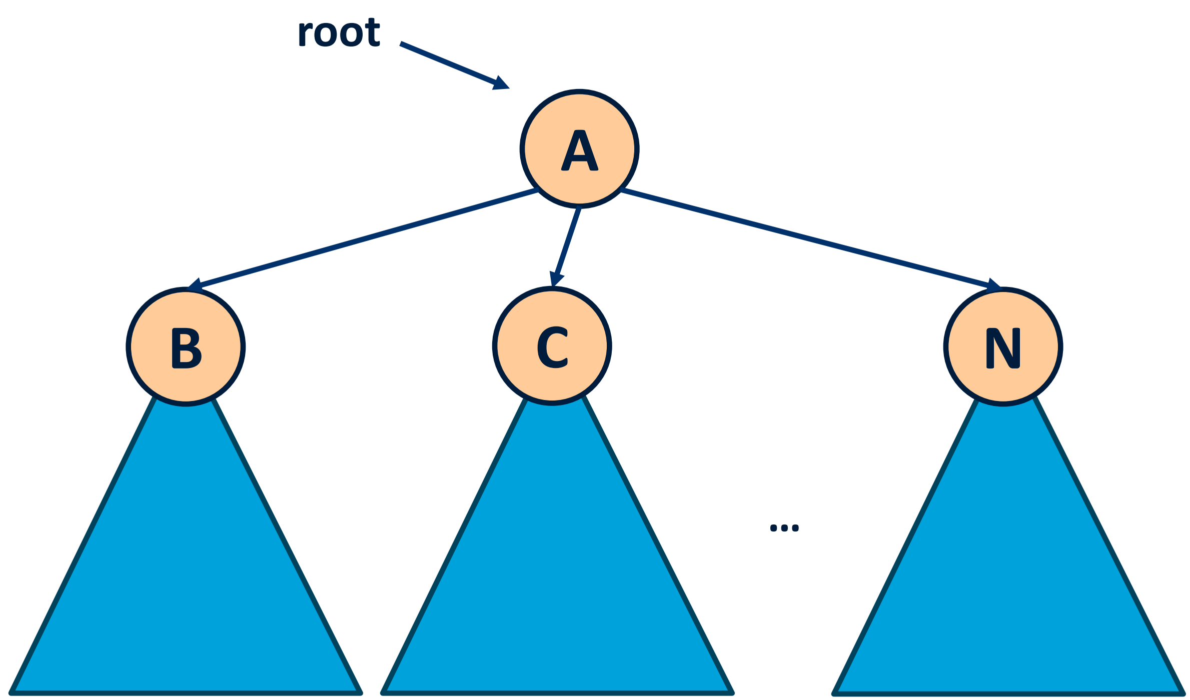


Programming with Trees

- **Post-order** visit
  - If the node is null -> nothing to do

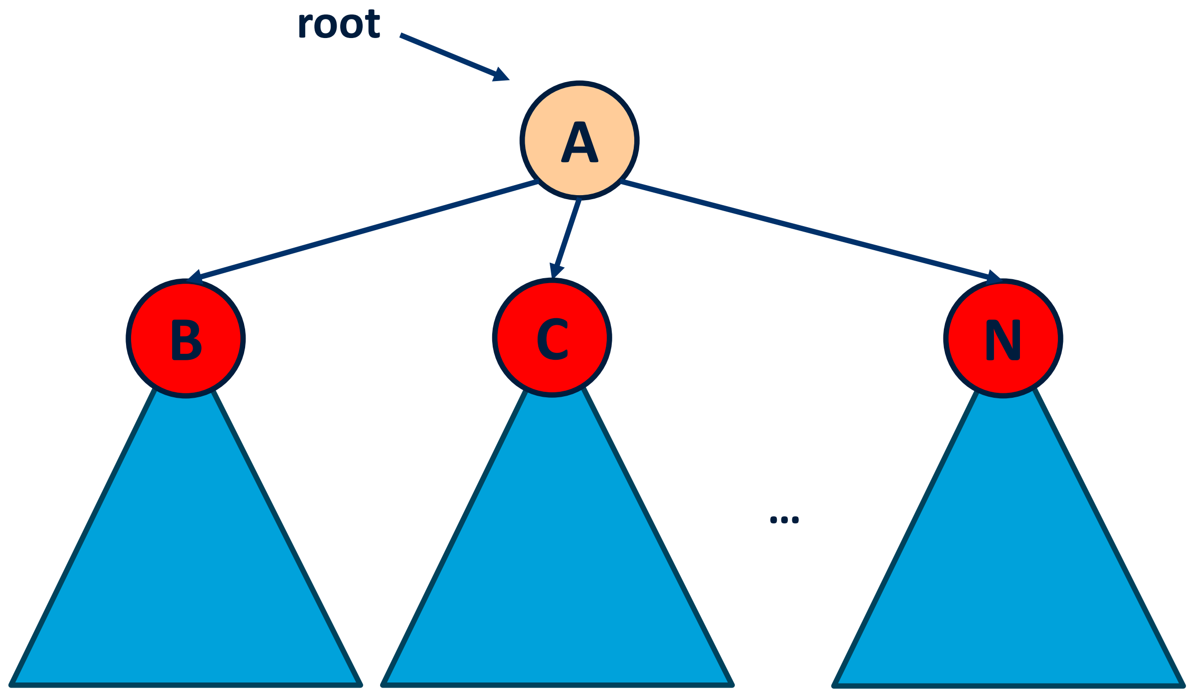
Programming with Trees

- **Post-order** visit
  - If the node is null -> nothing to do
  - If the node is not null



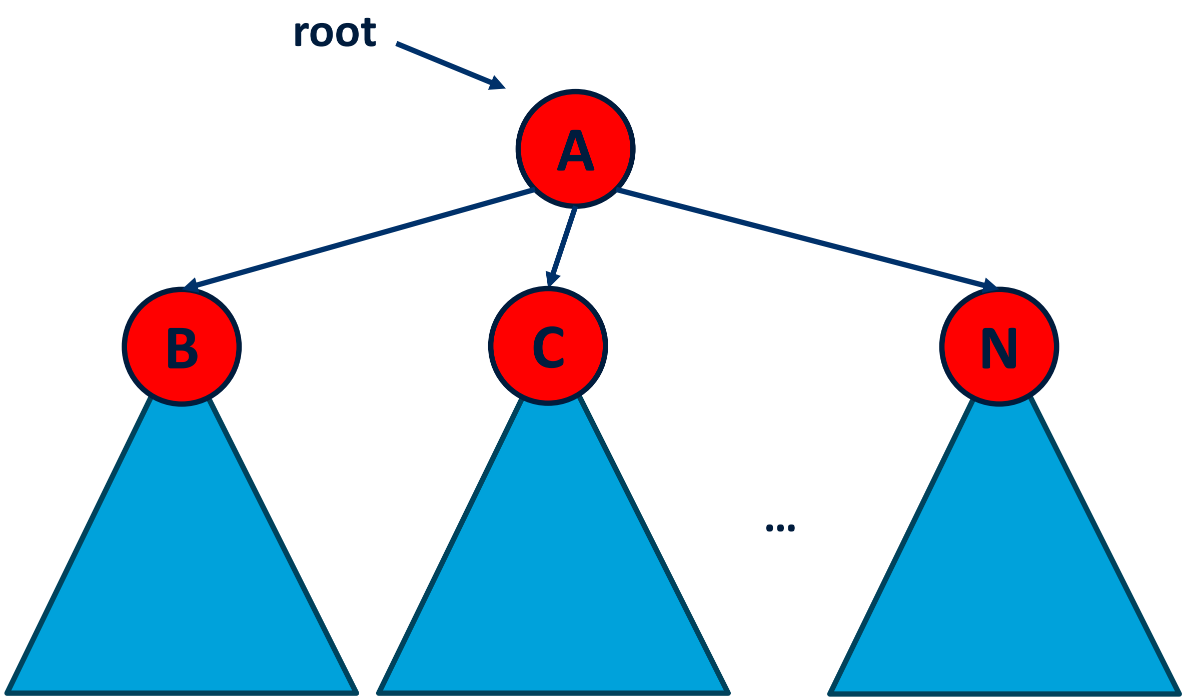
Programming with Trees

- **Post-order** visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit recursively all the children one by one



Programming with Trees

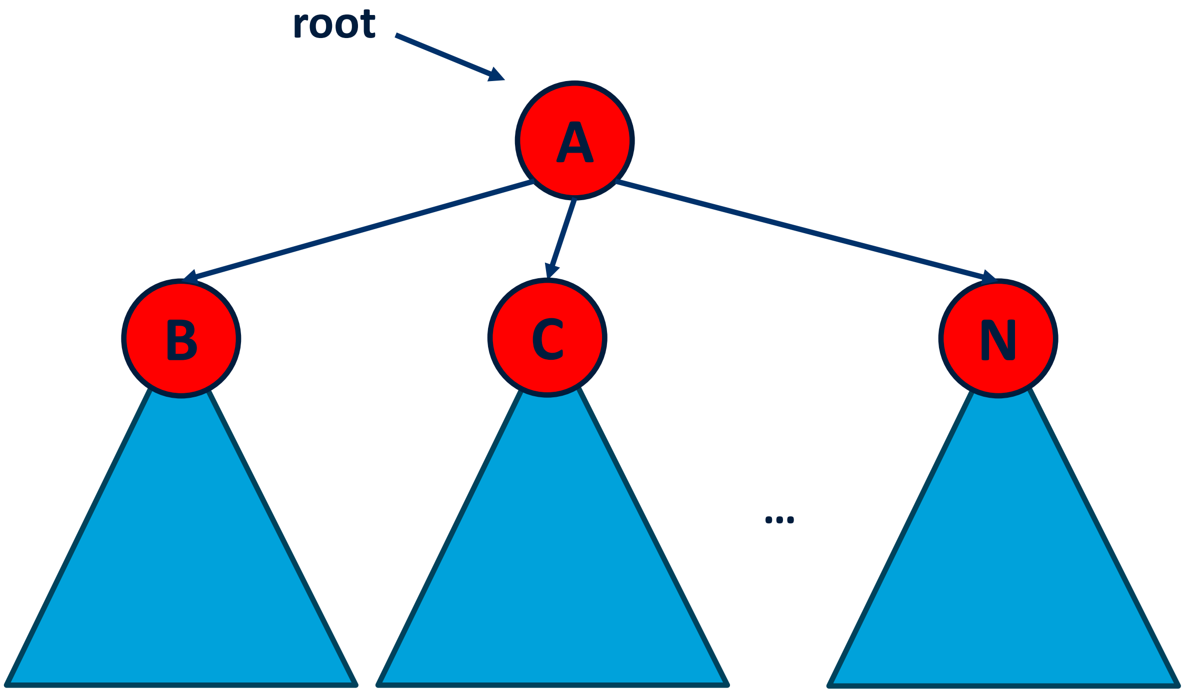
- **Post-order** visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit recursively all the children one by one
    - Visit the node



Programming with Trees

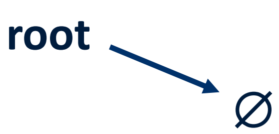
- **Post-order** visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit recursively all the children one by one
    - Visit the node

postOrder(v)
    if $v \neq \emptyset$
        for each child w of v
            postOrder(w)
        visit(v)



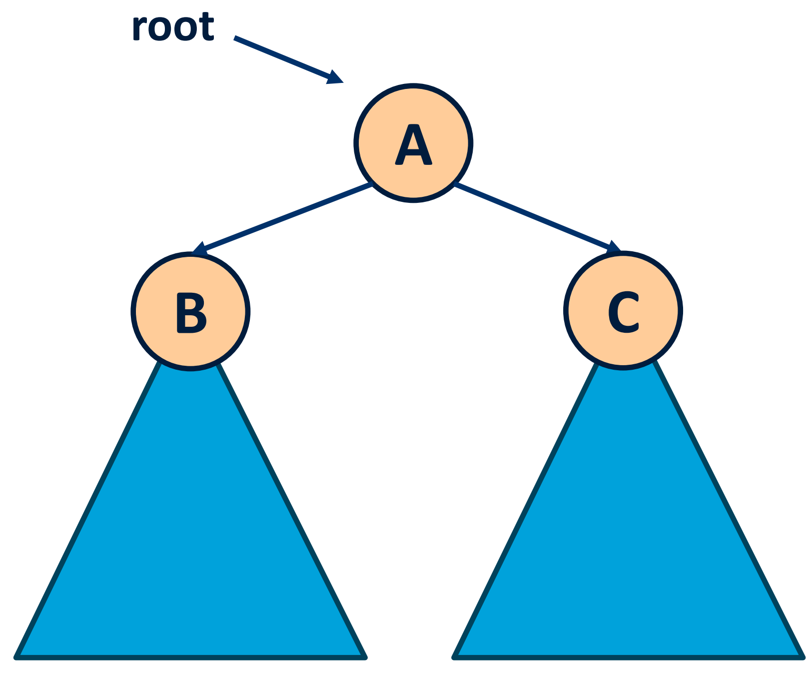
Programming with Trees

- In-order visit
  - If the node is null -> nothing to do



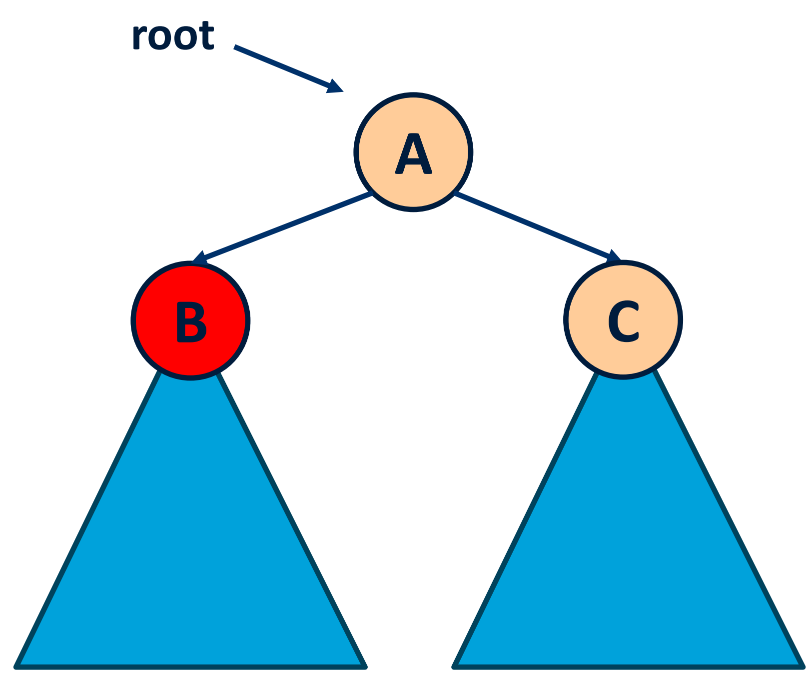
Programming with Trees

- In-order visit
  - If the node is null -> nothing to do
  - If the node is not null



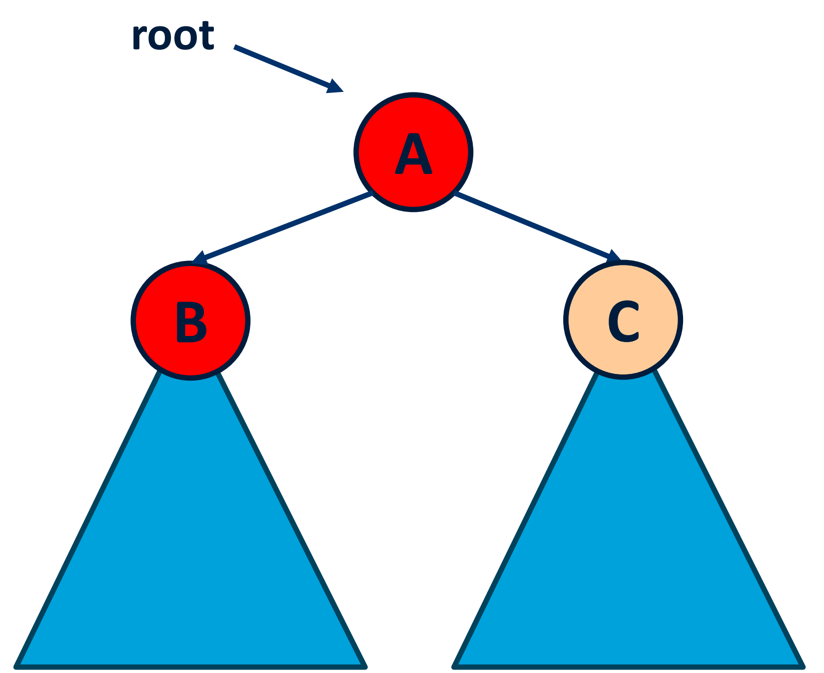
Programming with Trees

- In-order visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit recursively the left subtree



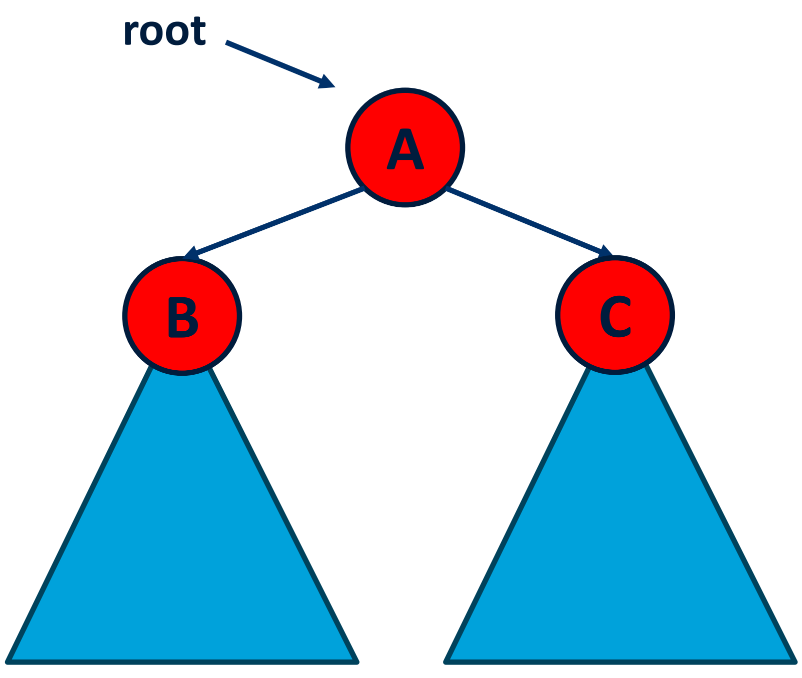
Programming with Trees

- In-order visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit recursively the left subtree
    - Visit the node



Programming with Trees

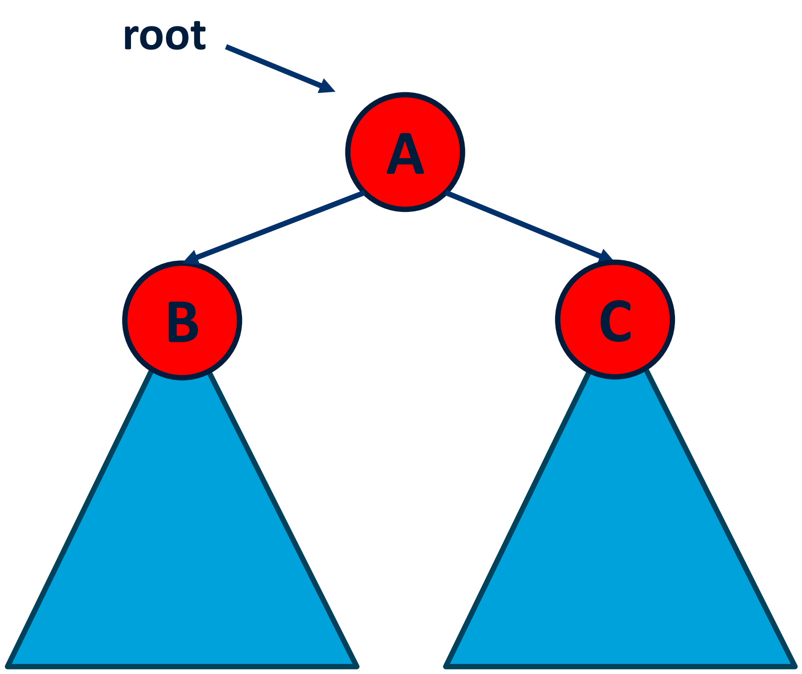
- In-order visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit recursively the left subtree
    - Visit the node
    - Visit recursively the right subtree



Programming with Trees

- In-order visit
  - If the node is null -> nothing to do
  - If the node is not null
    - Visit recursively the left subtree
    - Visit the node
    - Visit recursively the right subtree

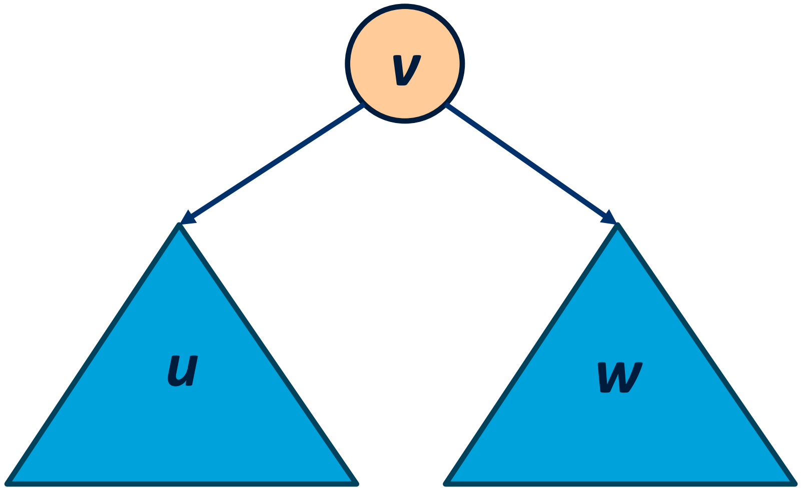
```
inOrder(v)
    if v ≠ ∅
        inOrder(left(v))
        visit(v)
        inOrder(right(v))
```



Binary Search Tree (BST)

Binary Search Trees (BST)

- A BST is a binary tree storing elements (keys) satisfying the following property:
  - Let ***u***, ***v***, and ***w*** be three nodes such that ***u is in the left subtree of v*** and ***w is in the right subtree of v***

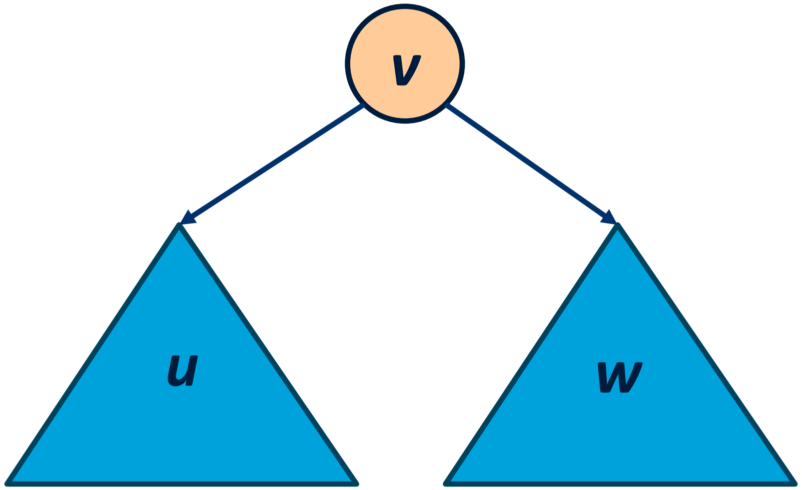


Binary Search Trees (BST)

- A BST is a binary tree storing elements (keys) satisfying the following property:
  - Let ***u***, ***v***, and ***w*** be three nodes such that ***u is in the left subtree of v*** and ***w is in the right subtree of v***

    Then

    key(u) ≤ key(v) ≤ key(w)

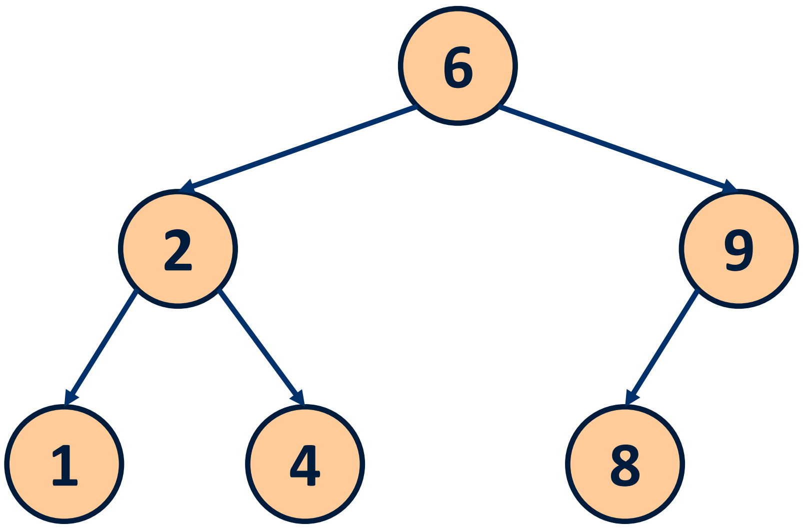


Binary Search Tree (BST)

- BSTs keep elements (keys) in sorted order
- Search, insertion, and deletion can use this info to find the elements or the position where to insert them
- Given a key to search, at each node, half of the node's subtrees can be eliminated

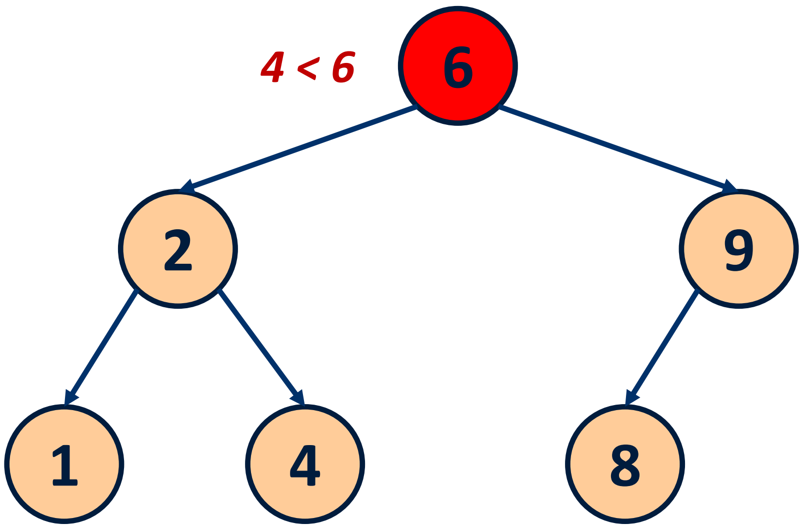
Search

- Example, *find(4)*



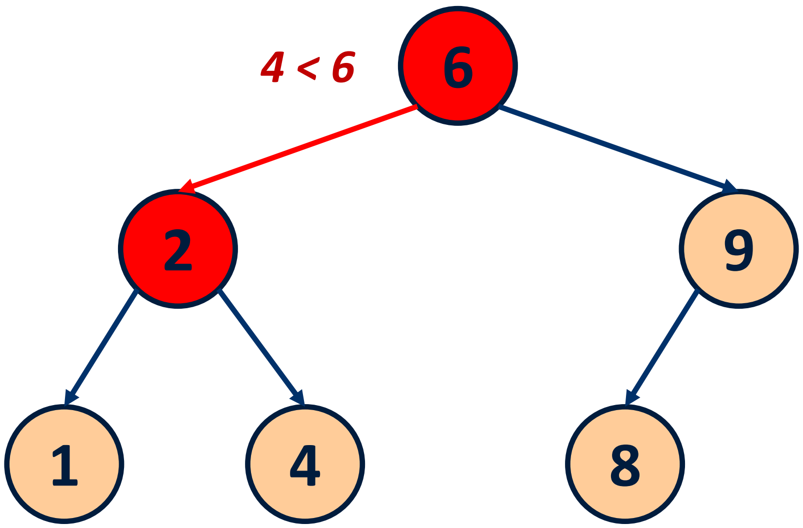
Search

- Example, *find(4)*



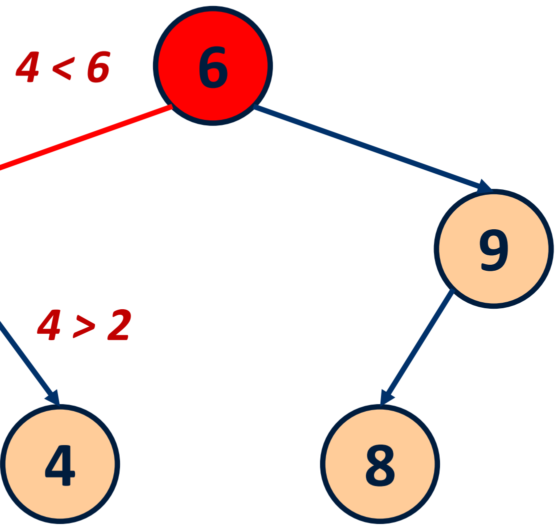
Search

- Example, *find(4)*



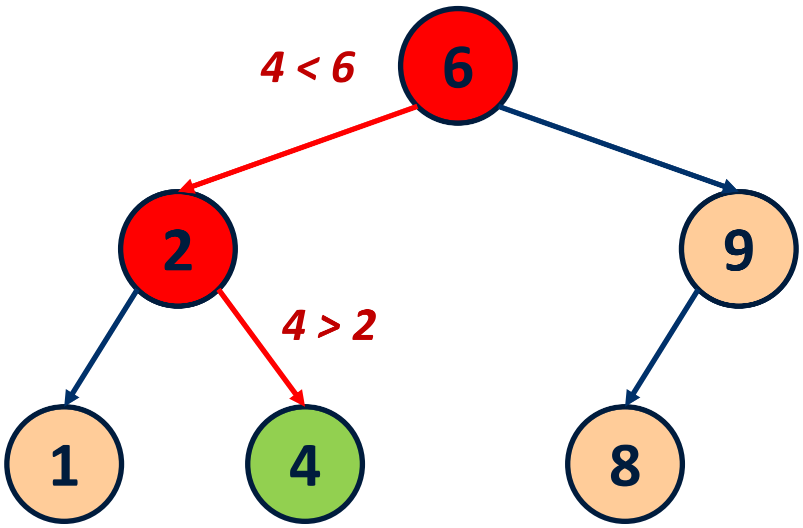
Search

- Example, *find(4)*



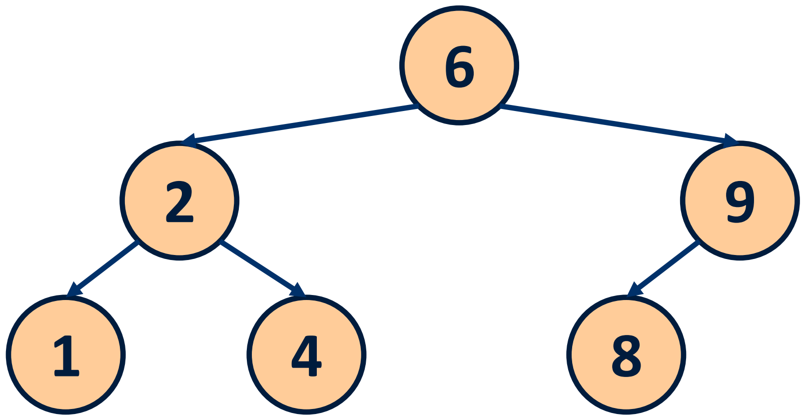
Search

- Example, *find(4)*



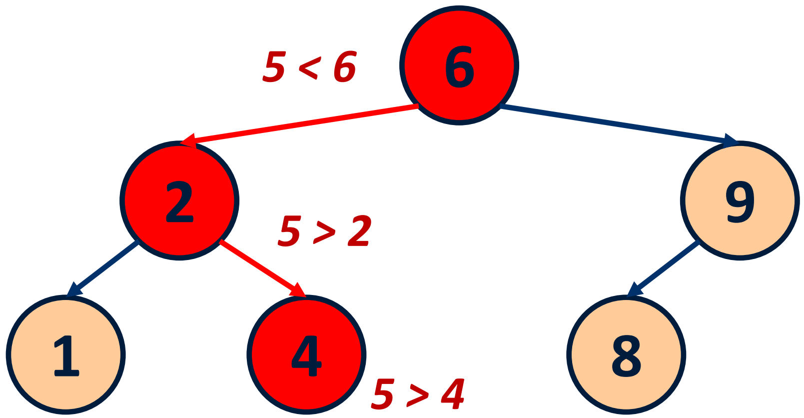
Insertion

- Example: *insert(5)*

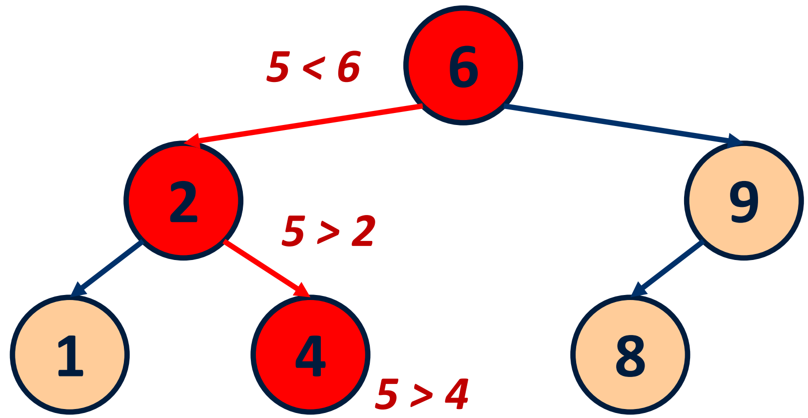


Insertion

- Example: *insert(5)*
  First, we *find(5)*



Insertion



- Example: *insert(5)*
  First, we *find(5)*

- Assume *5* is not already in the tree, 4 is the leaf we reach during the search

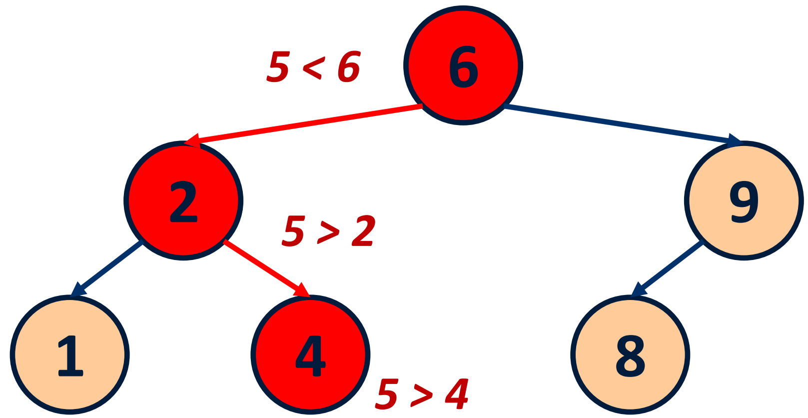
- As 5 > 4, we insert 5 as right child of 4

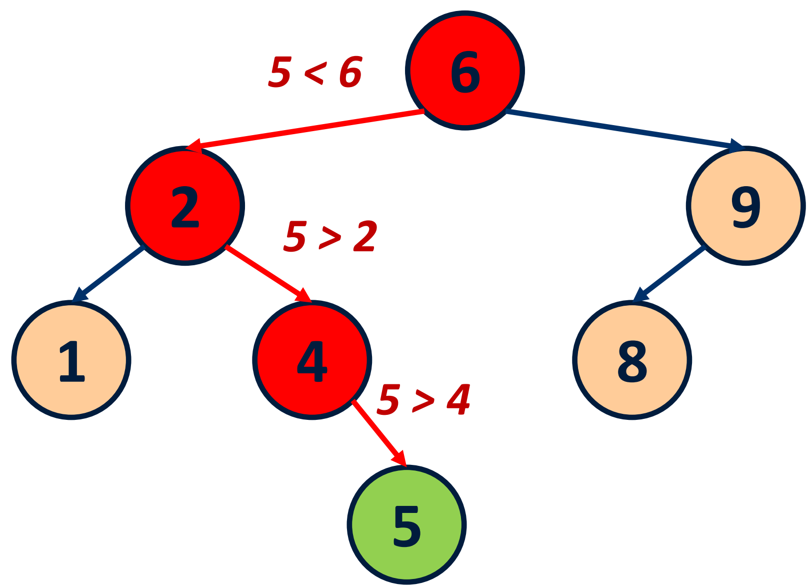
Insertion

- Example: *insert(5)*
  First, we *find(5)*

- Assume *5* is not already in the tree, 4 is the leaf we reach during the search

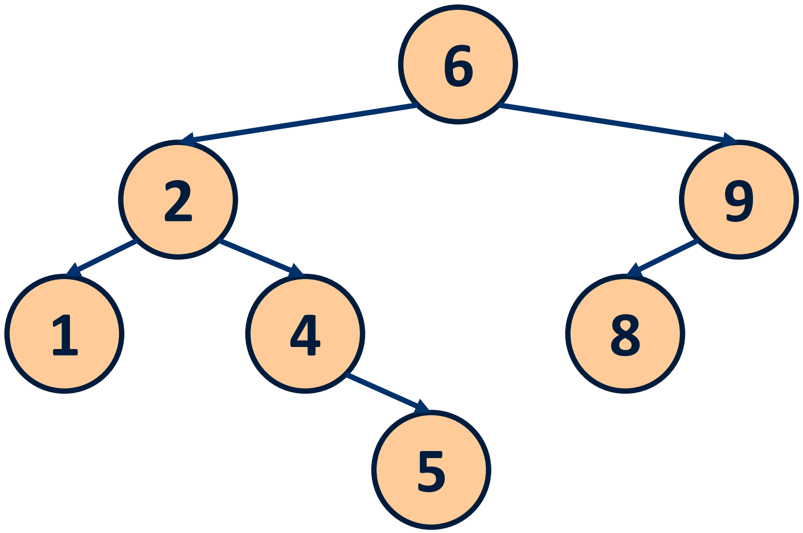
- As 5 > 4, we insert 5 as right child of 4





Deletion
Case 1 – leaf

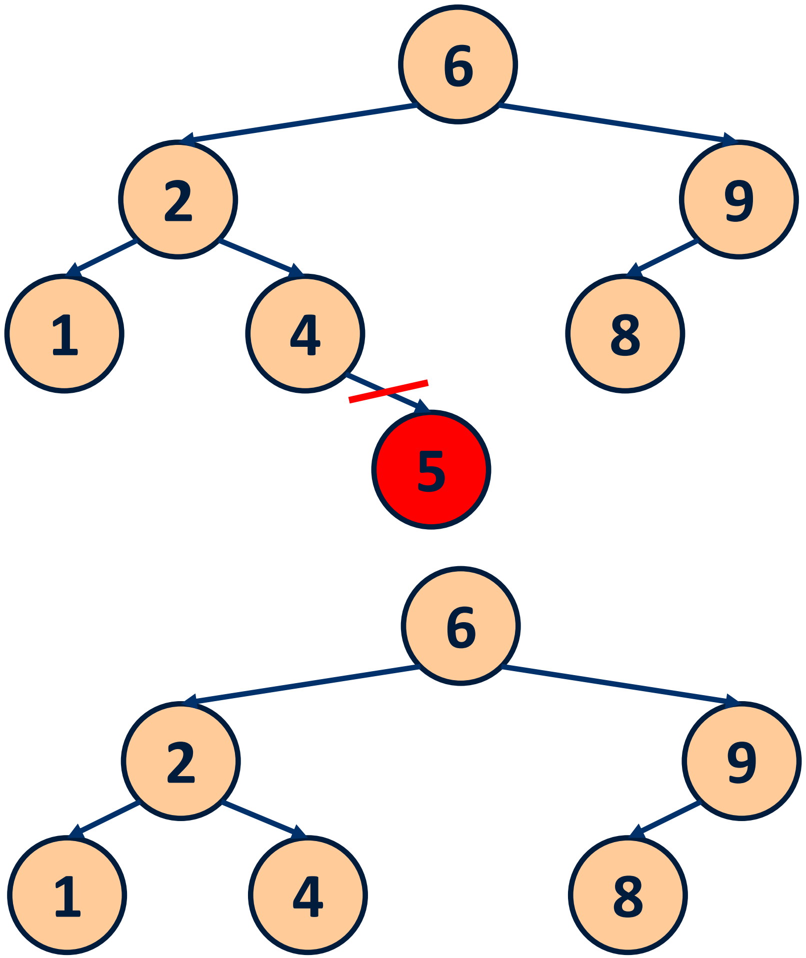
- Example: *remove(5)*



Deletion
Case 1 – leaf

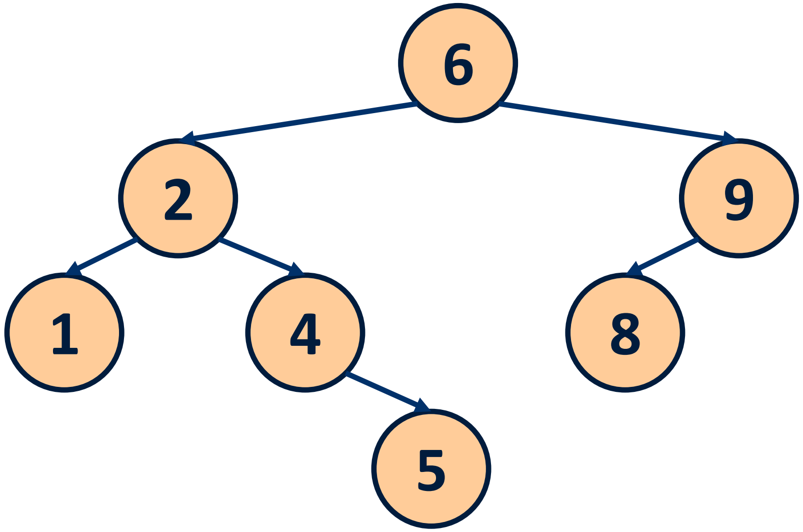
- Example: *remove(5)*

- As 5 is a leaf, we can just remove it



Deletion
Case 2 – internal node with one child

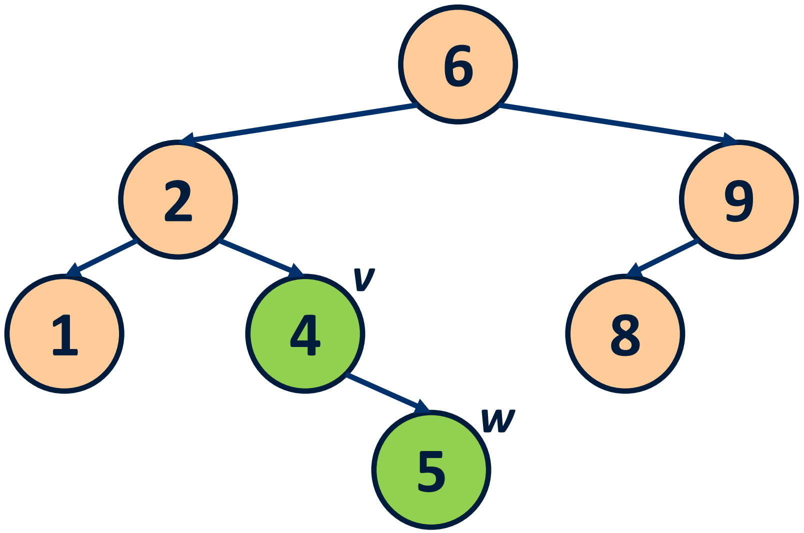
- Example: *remove(4)*



Deletion
Case 2 – internal node with one child

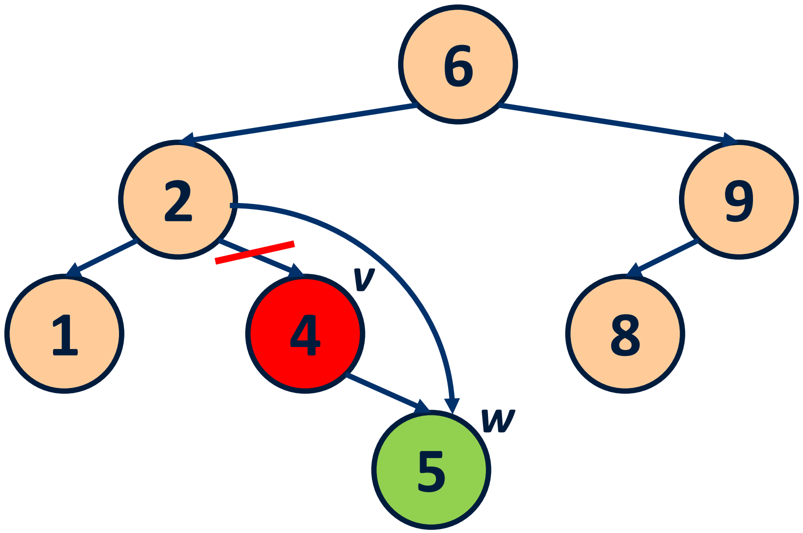
- Example: *remove(4)*

- Let ***v*** be the node storing 4 and ***w*** its only child



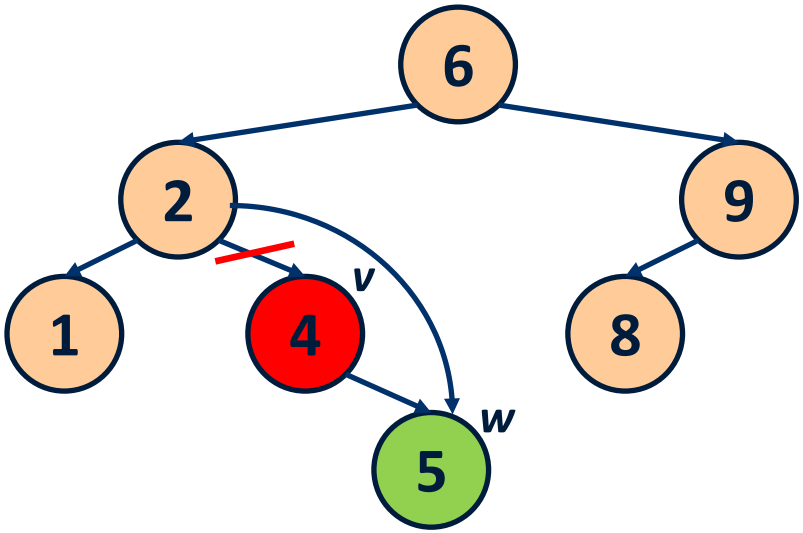
Deletion
Case 2 – internal node with one child

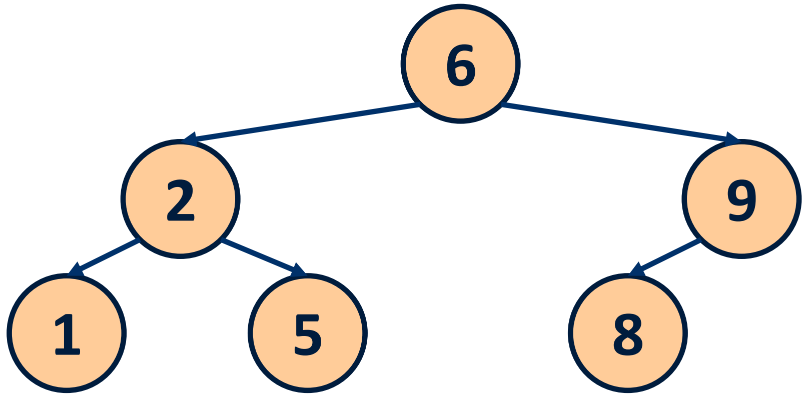
- Example: *remove(4)*
- Let ***v*** be the node storing 4 and ***w*** its only child
- We remove ***v*** by assigning ***w*** to its parent



Deletion
Case 2 – internal node with one child

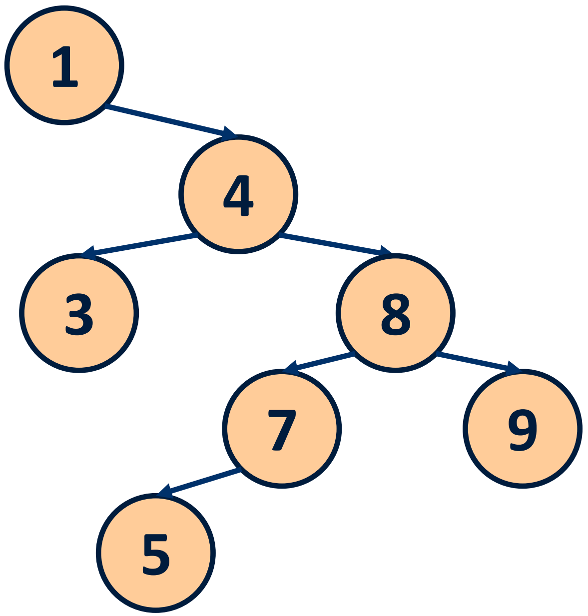
- Example: *remove(4)*
- Let ***v*** be the node storing 4 and ***w*** its only child
- We remove ***v*** by assigning ***w*** to its parent





Deletion
Case 3 – internal node with two children

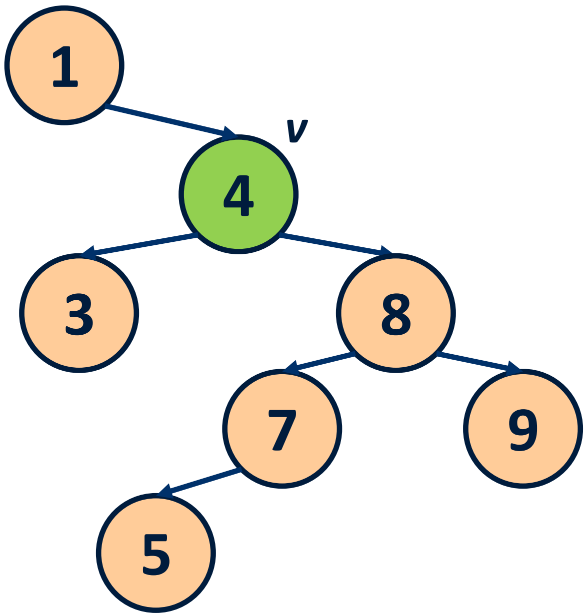
- Example: *remove(4)*



Deletion
Case 3 – internal node with two children

- Example: *remove(4)*

- Let ***v*** be the node storing 4 having 2 children

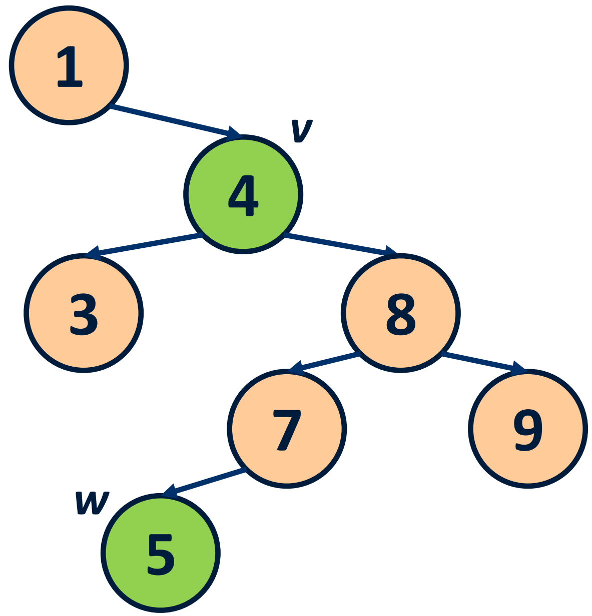


Deletion
Case 3 – internal node with two children

- Example: *remove(4)*

- Let ***v*** be the node storing 4 having 2 children

  1. Find the node ***w*** that follows ***v*** in an **inorder** traversal

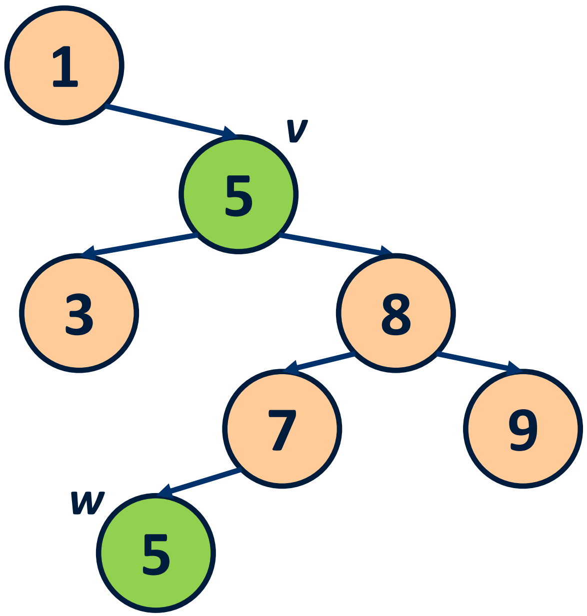


Deletion
Case 3 – internal node with two children

- Example: *remove(4)*

- Let ***v*** be the node storing 4 having 2 children

  1. Find the node ***w*** that follows ***v*** in an **inorder** traversal
  2. Replace ***v*** with ***w***

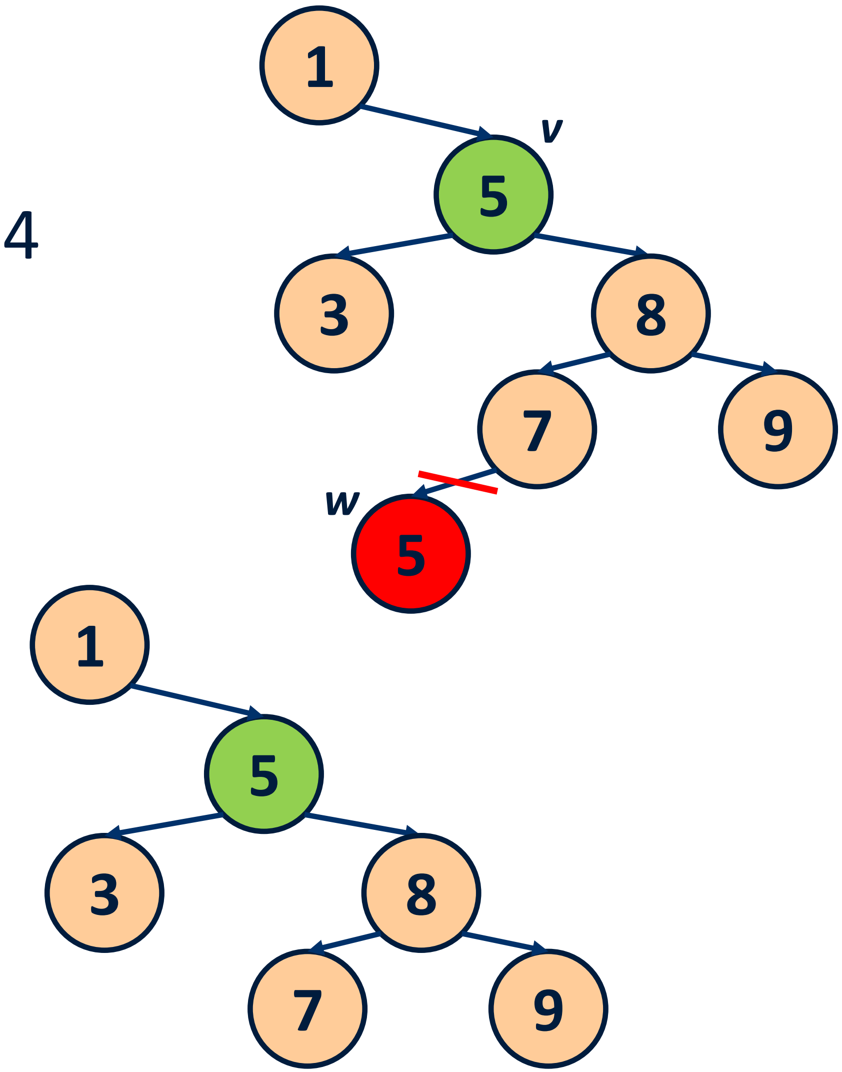


Deletion
Case 3 – internal node with two children

- Example: *remove(4)*

- Let ***v*** be the node storing 4 having 2 children

  1. Find the node ***w*** that follows ***v*** in an **inorder** traversal
  2. Replace ***v*** with ***w***
  3. Remove node ***w***

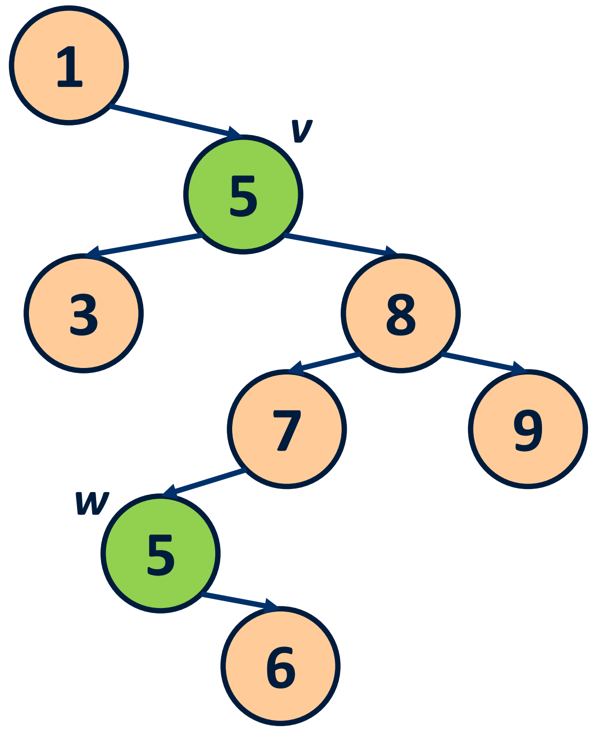


Deletion
Case 3 – if the successor has a child?

- Example: *remove(4)*

- Let ***v*** be the node storing 4 having 2 children

  1. Find the node ***w*** that follows ***v*** in an **inorder** traversal
  2. Replace ***v*** with ***w***
  3. Remove node ***w***

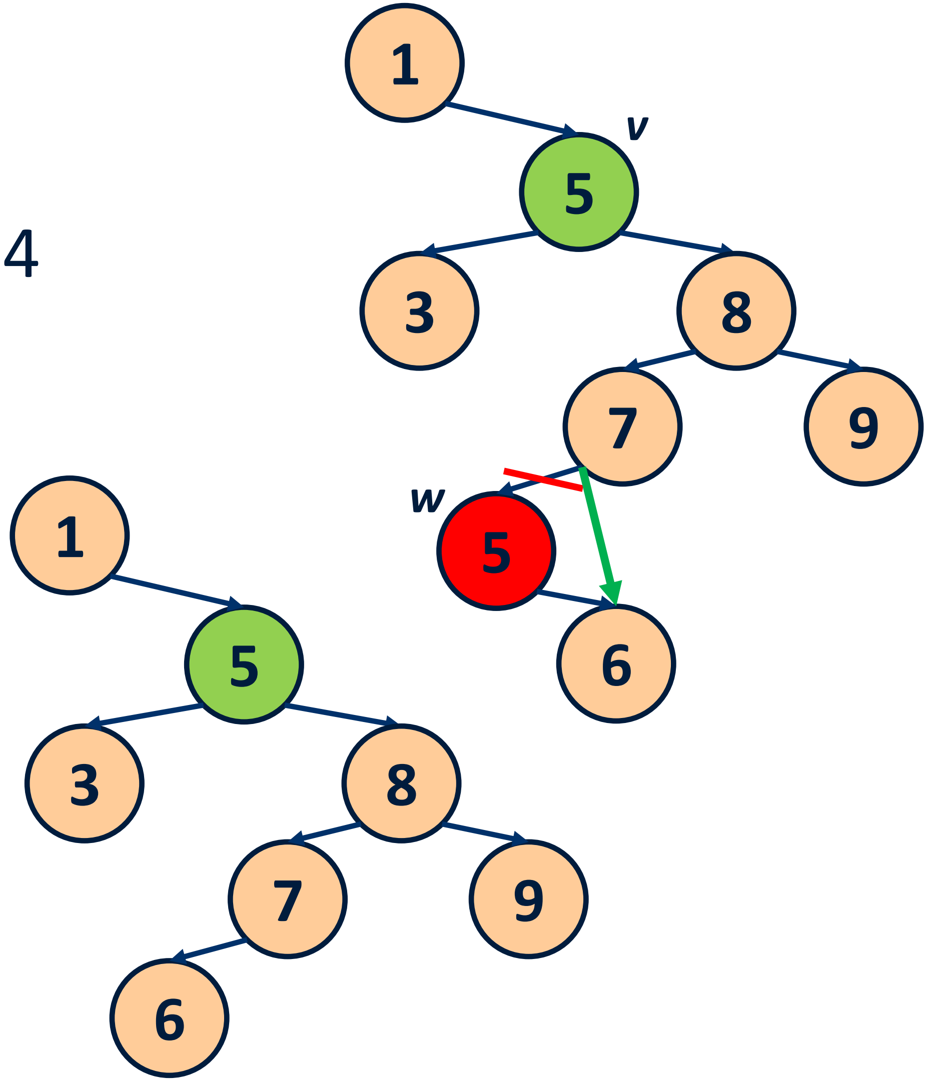


Deletion
Case 3 – if the successor has a child?

- Example: *remove(4)*

- Let ***v*** be the node storing 4 having 2 children

  1. Find the node ***w*** that follows ***v*** in an **inorder** traversal
  2. Replace ***v*** with ***w***
  3. Remove node ***w***



Range Queries

- An additional operation that can be answered by a binary search tree is a range query

- ***findInRange(k1, k2, node)***:
  find all elements ***k*** stored in the BST rooted in ***node*** such that:

  $$k1 \leq k \leq k2$$

Range Queries

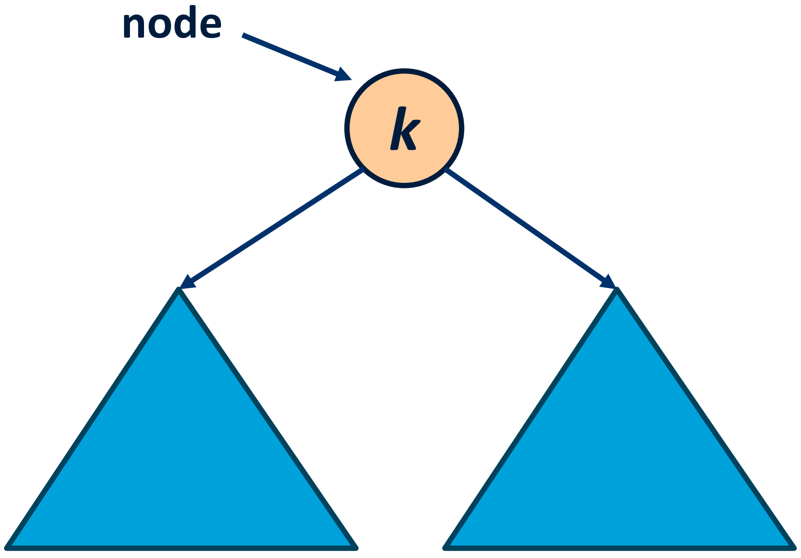
findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do

Range Queries

findInRange(k1, k2, node)

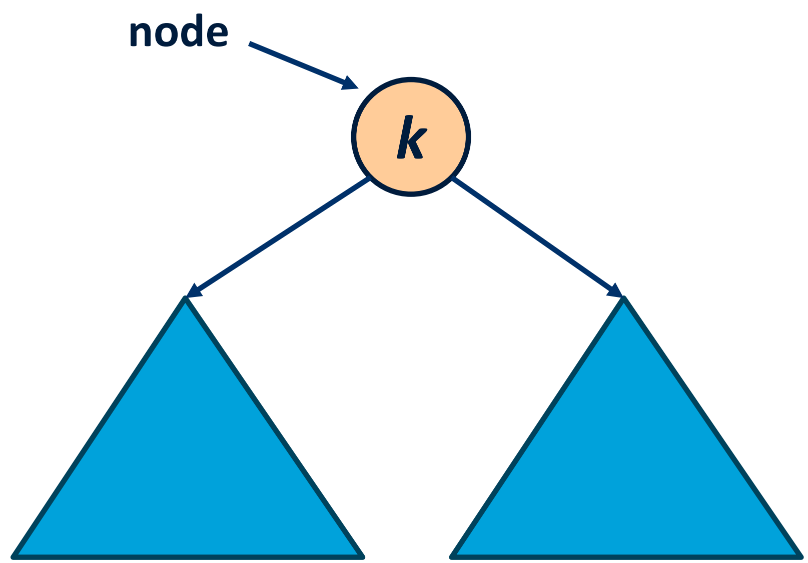
- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node



Range Queries

findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If $k1 \leq k \leq k2$:

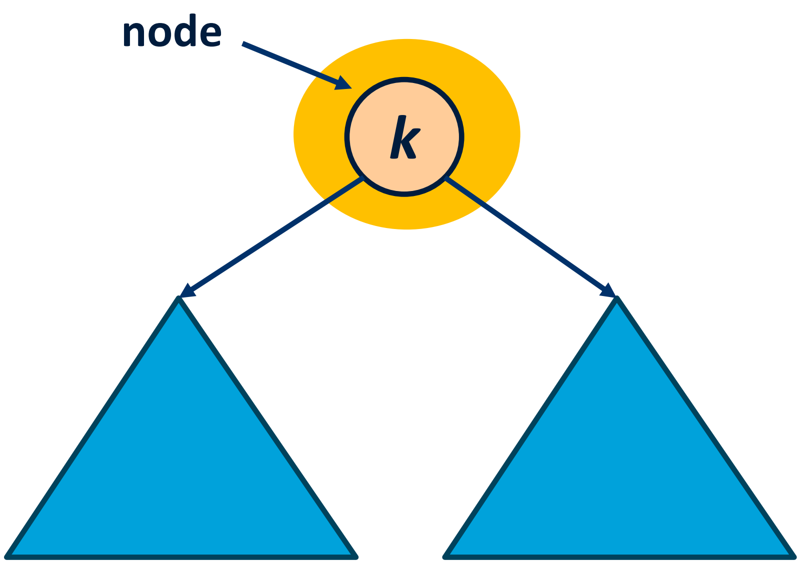


Range Queries

- ***k*** is in the interval and we need to return it
- Where can other elements to return be?

findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If ***k1 ≤ k ≤ k2***:

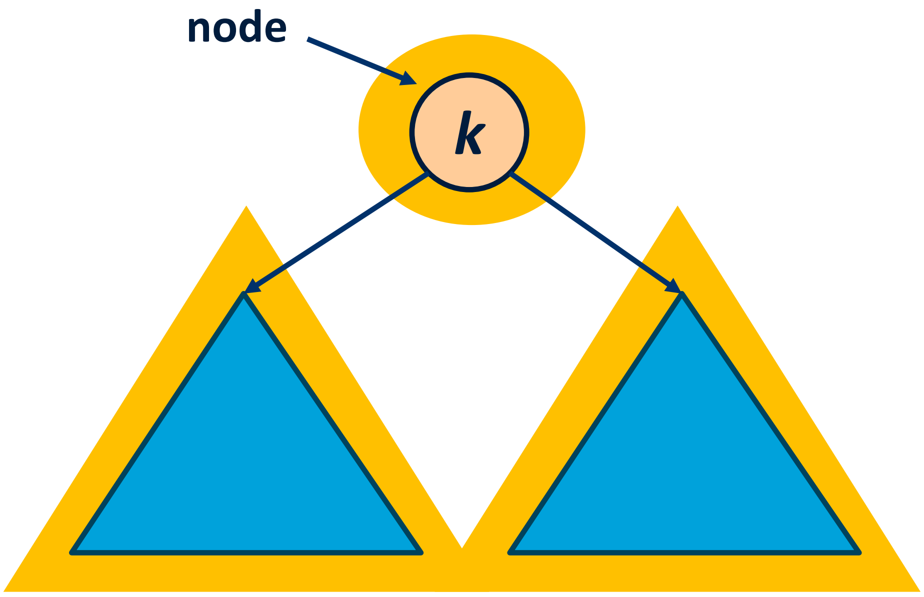


Range Queries

- ***k*** is in the interval and we need to return it
- Where can other elements to return be?
  - In both subtrees!

findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If ***k1 ≤ k ≤ k2***:



Range Queries

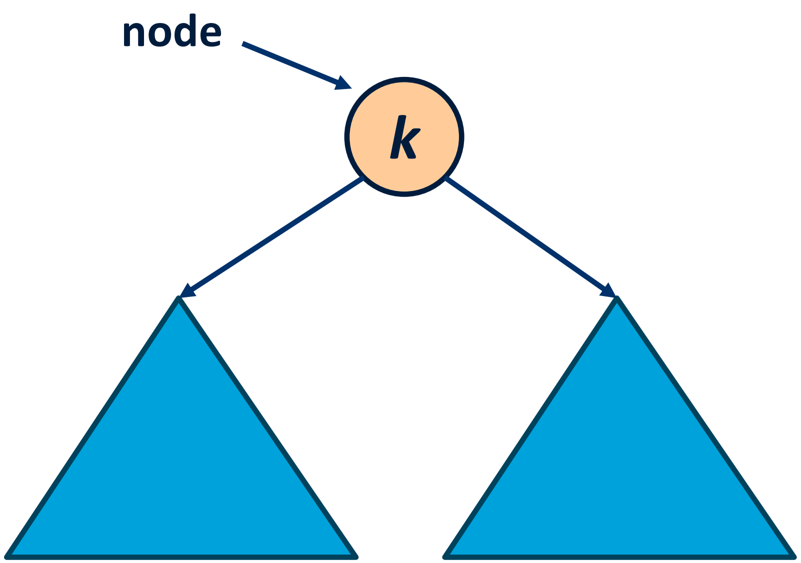
findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If ***k1 ≤ k ≤ k2***:
    - we return the result of both recursive calls plus the element ***k***

Range Queries

findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If ***k1 ≤ k ≤ k2***:
    - we return the result of botl
  - If ***k < k1***:

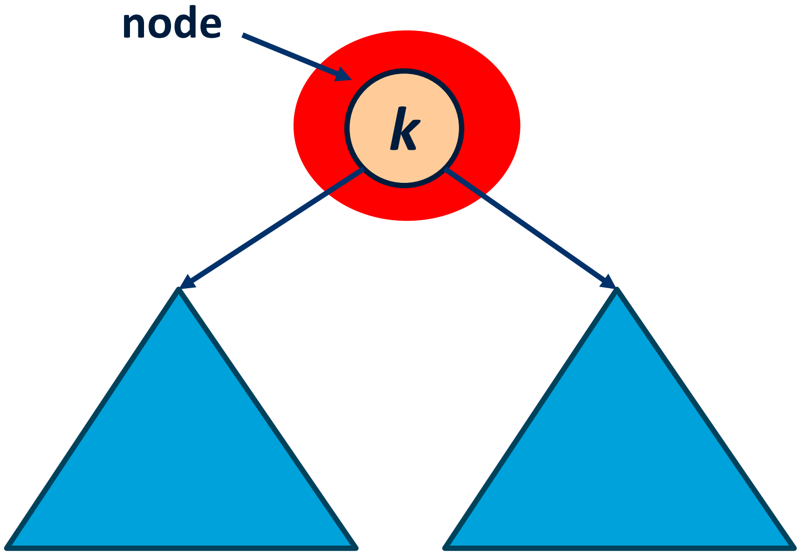


Range Queries

findInRange(k1, k2, node)

- *k* is **NOT** in the interval and we don't return it
- Where can other elements to return be?

- If *node* is null -> nothing to do
- If *node* is not null, let *k* be the element in node
  - If $k1 \leq k \leq k2$:
    - we return the result of botl
  - If $k < k1$:

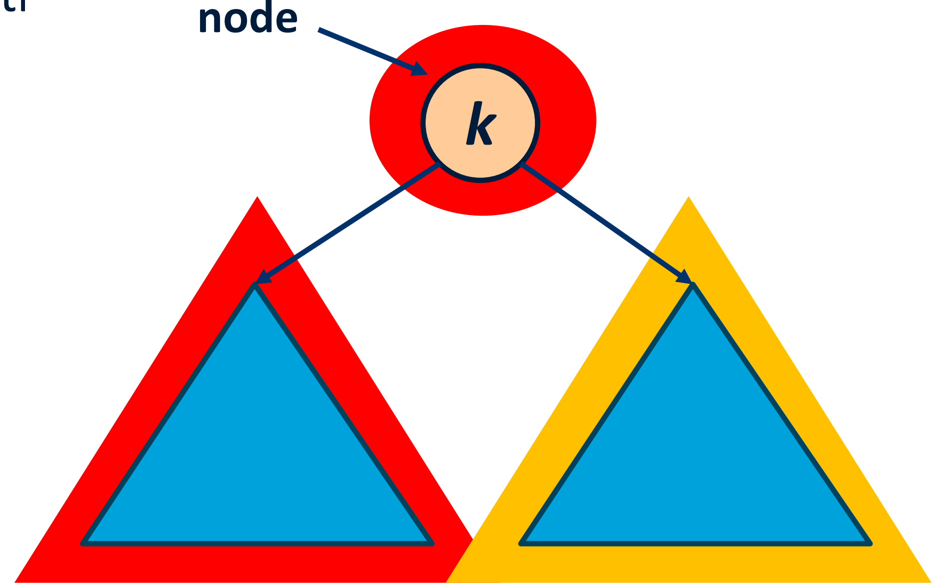


Range Queries

- ***k*** is **NOT** in the interval and we don't return it
- Where can other elements to return be?
  - Only in the right subtree!

findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If $k1 \leq k \leq k2$:
    - we return the result of botl
  - If $k < k1$:



Range Queries

findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If $k1 \leq k \leq k2$:
    - we return the result of both recursive calls plus the element ***k***
  - If $k < k1$:
    - we return the result of the recursive call in the ***right subtree***

Range Queries

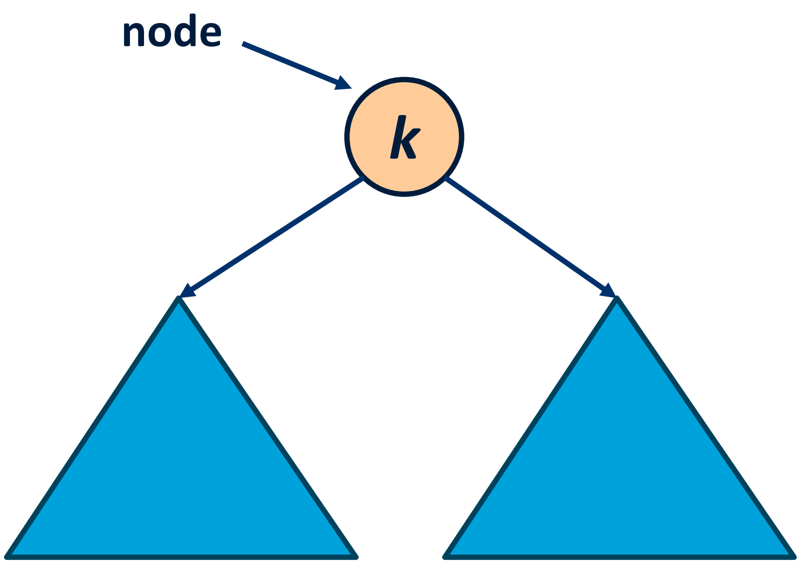
findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If $k1 \leq k \leq k2$:
    - we return the result of both recursive calls plus the element ***k***
  - If $k < k1$:
    - we return the result of the recursive call in the ***right subtree***
  - If $k2 < k$:

Range Queries

findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If $k1 \leq k \leq k2$:
    - we return the result of botl
  - If $k < k1$:
    - we return the result of the
  - If $k2 < k$:

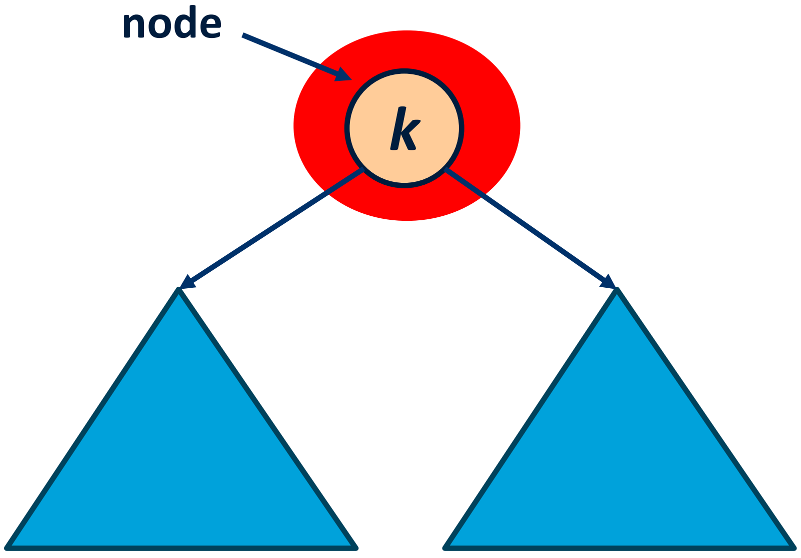


Range Queries

findInRange(k1, k2, node)

- **k** is **NOT** in the interval and we don't return it
- Where can other elements to return be?

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If $k1 \le k \le k2$:
    - we return the result of botl
  - If $k < k1$:
    - we return the result of the
  - If $k2 < k$:

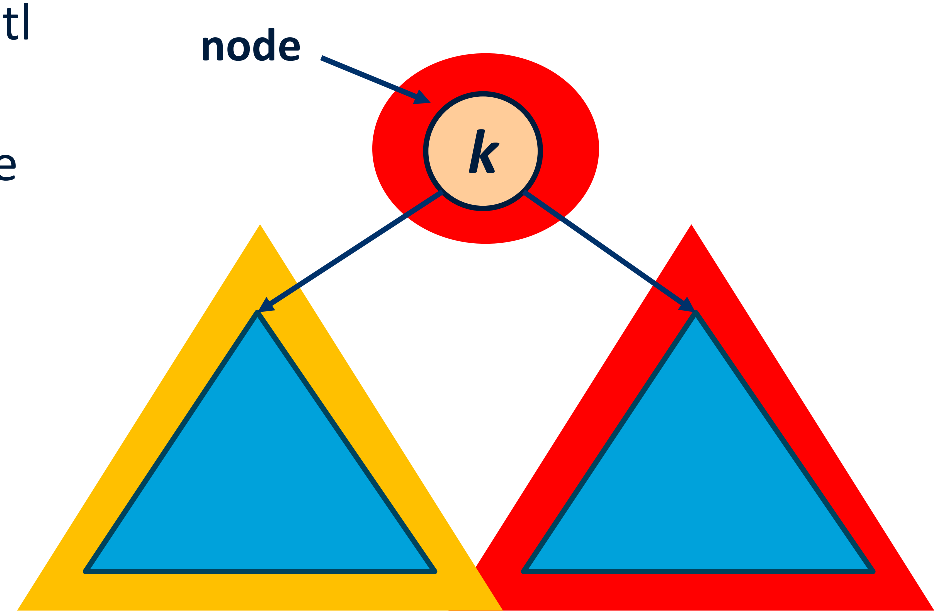


Range Queries

- ***k*** is **NOT** in the interval and we don't return it
- Where can other elements to return be?
  - Only in the left subtree!

findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If ***k1 ≤ k ≤ k2***:
    - we return the result of botl
  - If ***k < k1***:
    - we return the result of the
  - If ***k2 < k***:



Range Queries

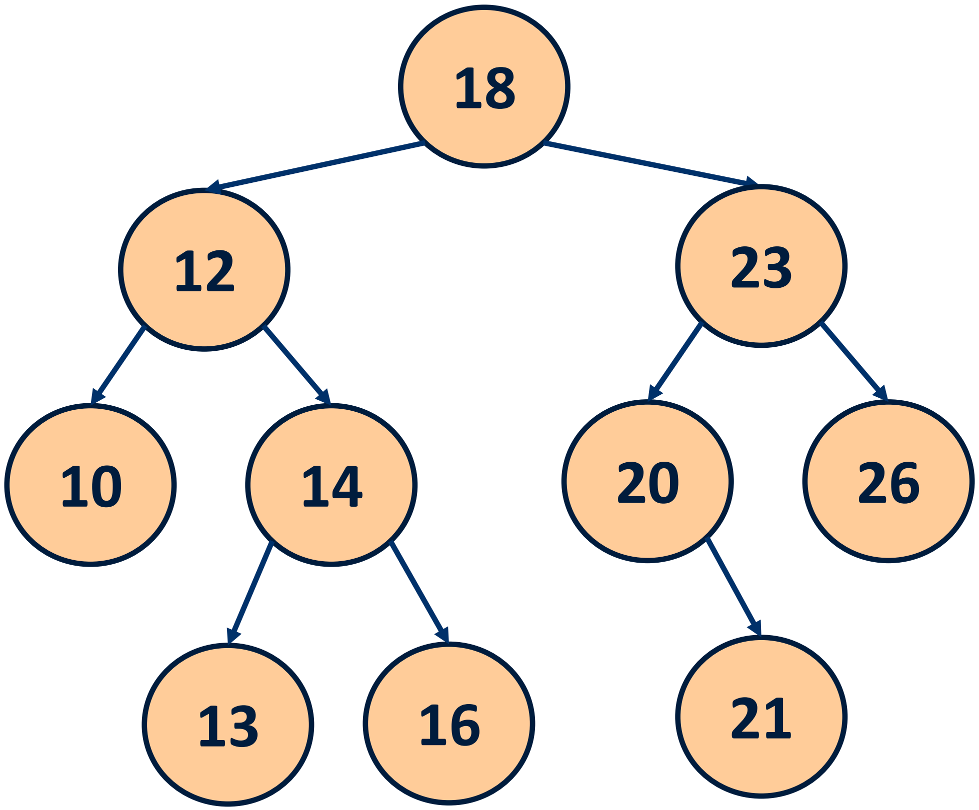
findInRange(k1, k2, node)

- If ***node*** is null -> nothing to do
- If ***node*** is not null, let ***k*** be the element in node
  - If $k1 \leq k \leq k2$:
    - we return the result of both recursive calls plus the element ***k***
  - If $k < k1$:
    - we return the result of the recursive call in the ***right subtree***
  - If $k2 < k$:
    - we return the result of the recursive call in the ***left subtree***

Range Query Algorithm

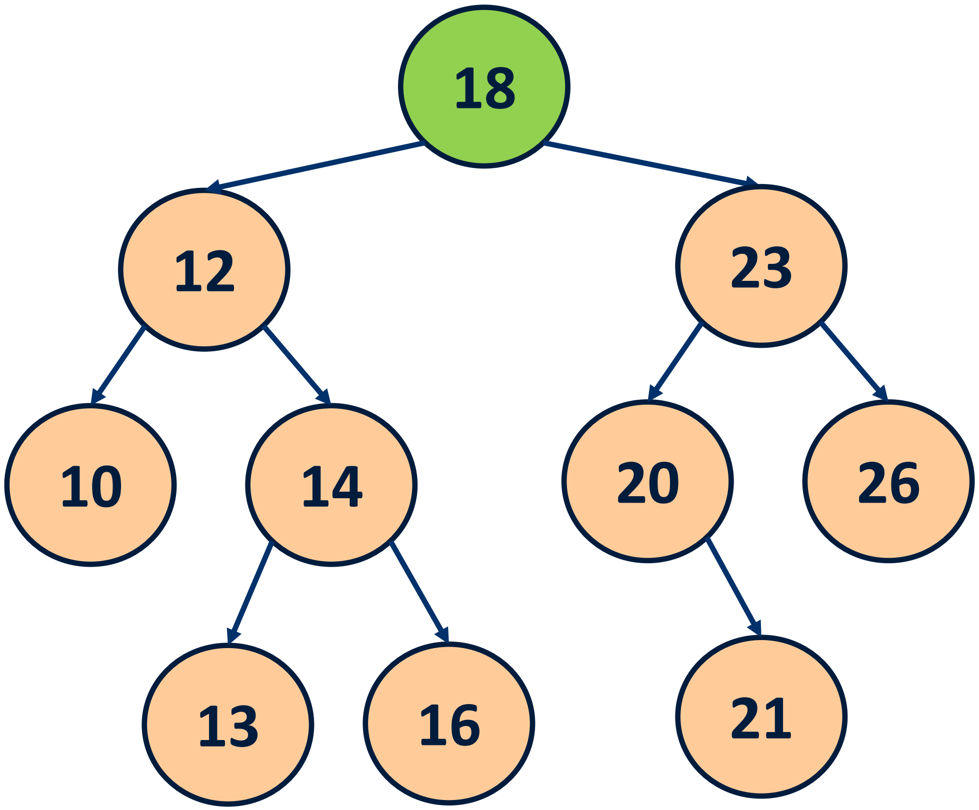
```
findInRange(k1, k2, node)
    if v ≠ ∅
        if k1 ≤ key(node) ≤ k2
            L = findInRange(k1, k2, left(node))
            R = findInRange(k1, k2, right(node))
            return {element(node)} ∪ L ∪ R
        else if k < k1
            return findInRange(k1, k2, right(node))
        else if k2 < k
            return findInRange(k1, k2, left(node))
```

Range Query Algorithm - Example



k1 = 13
k2 = 22

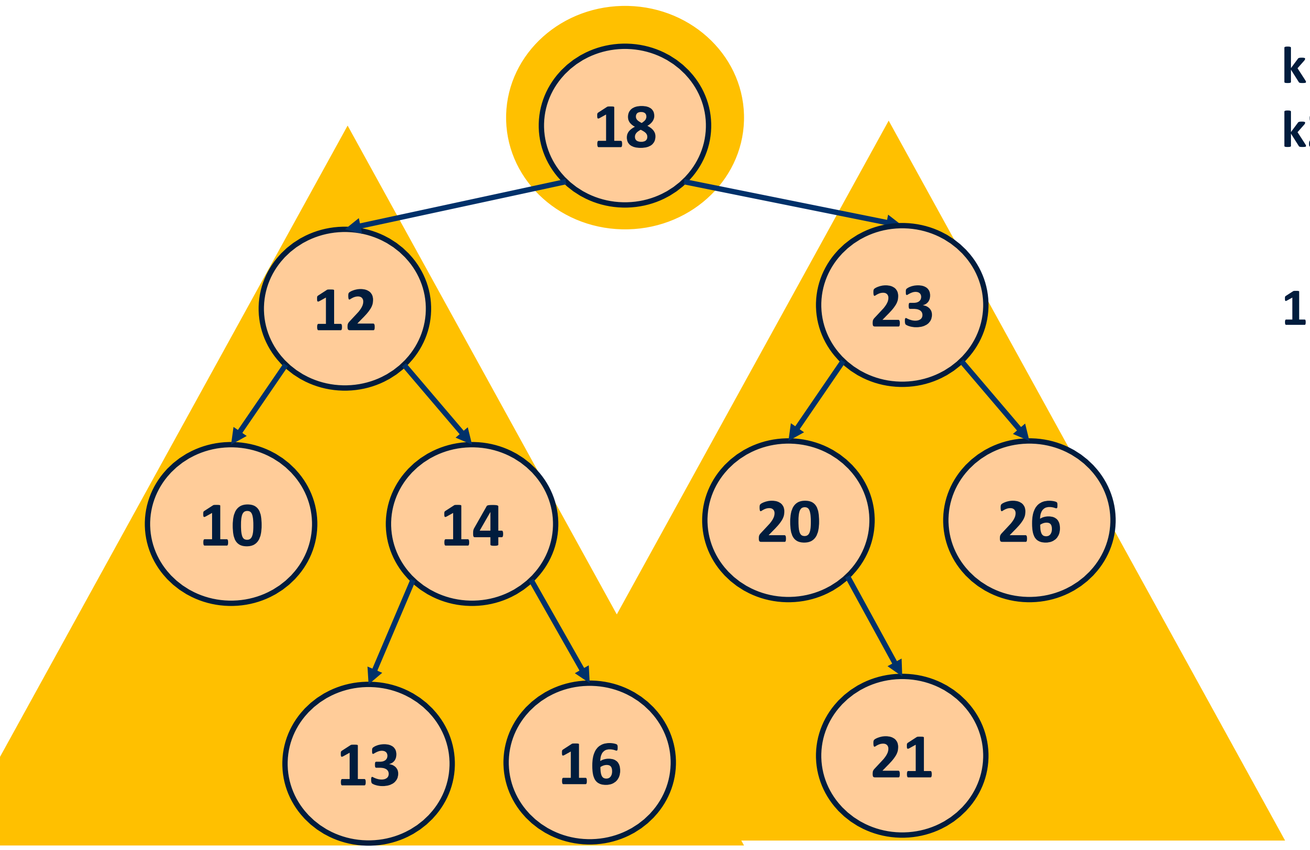
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22

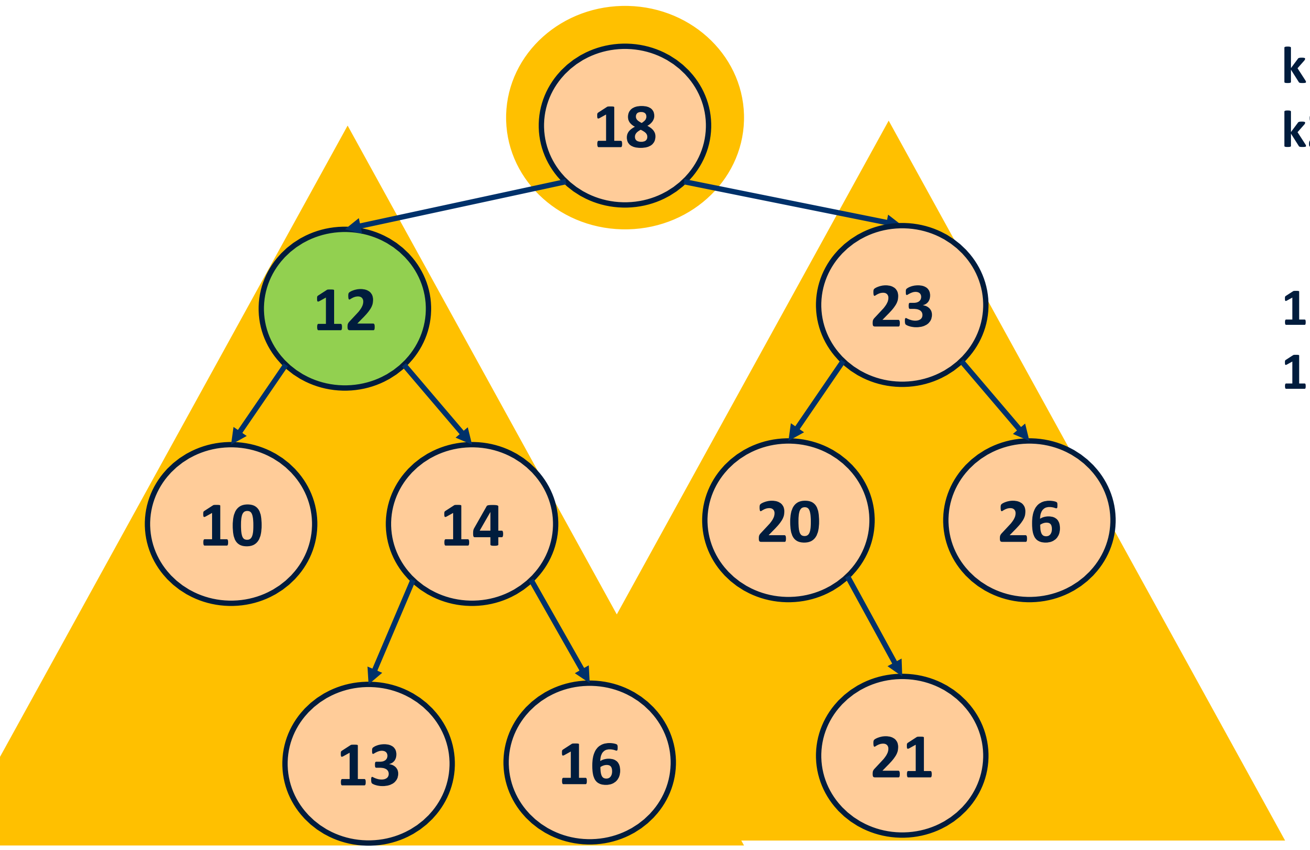
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22

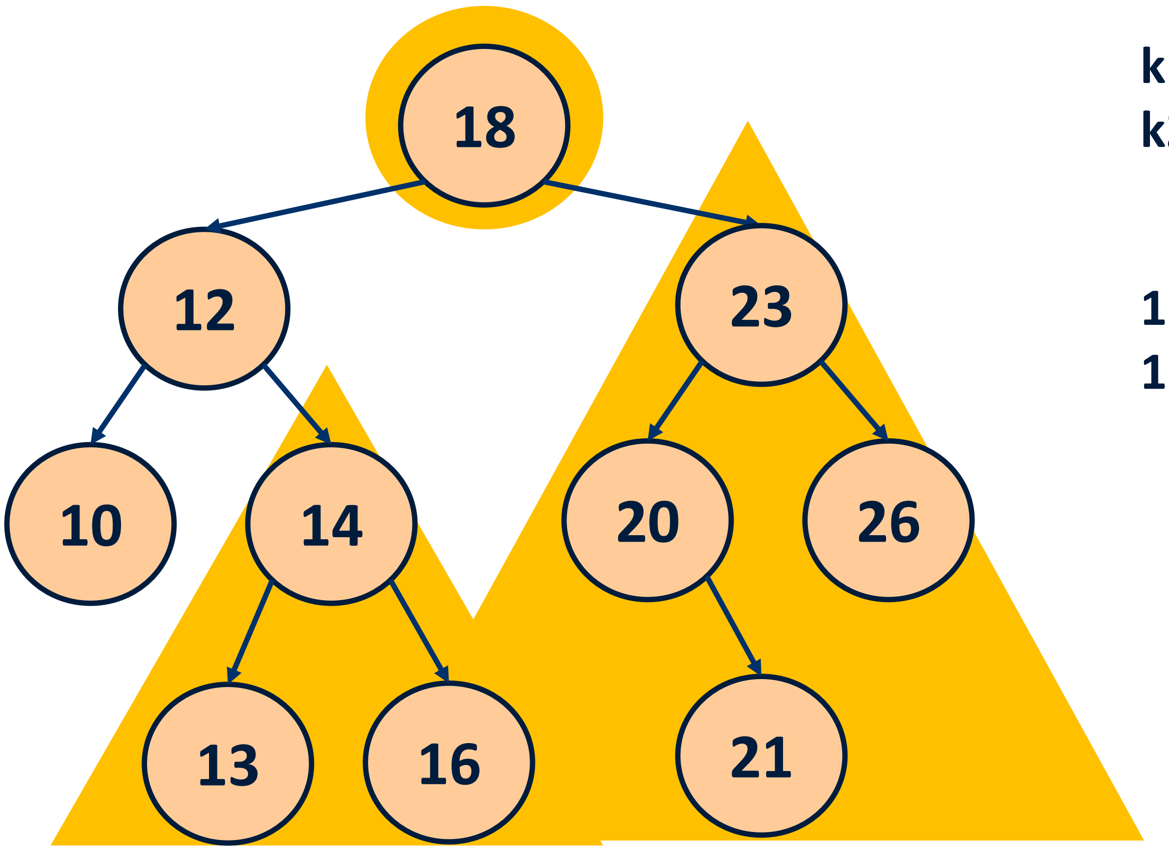
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13

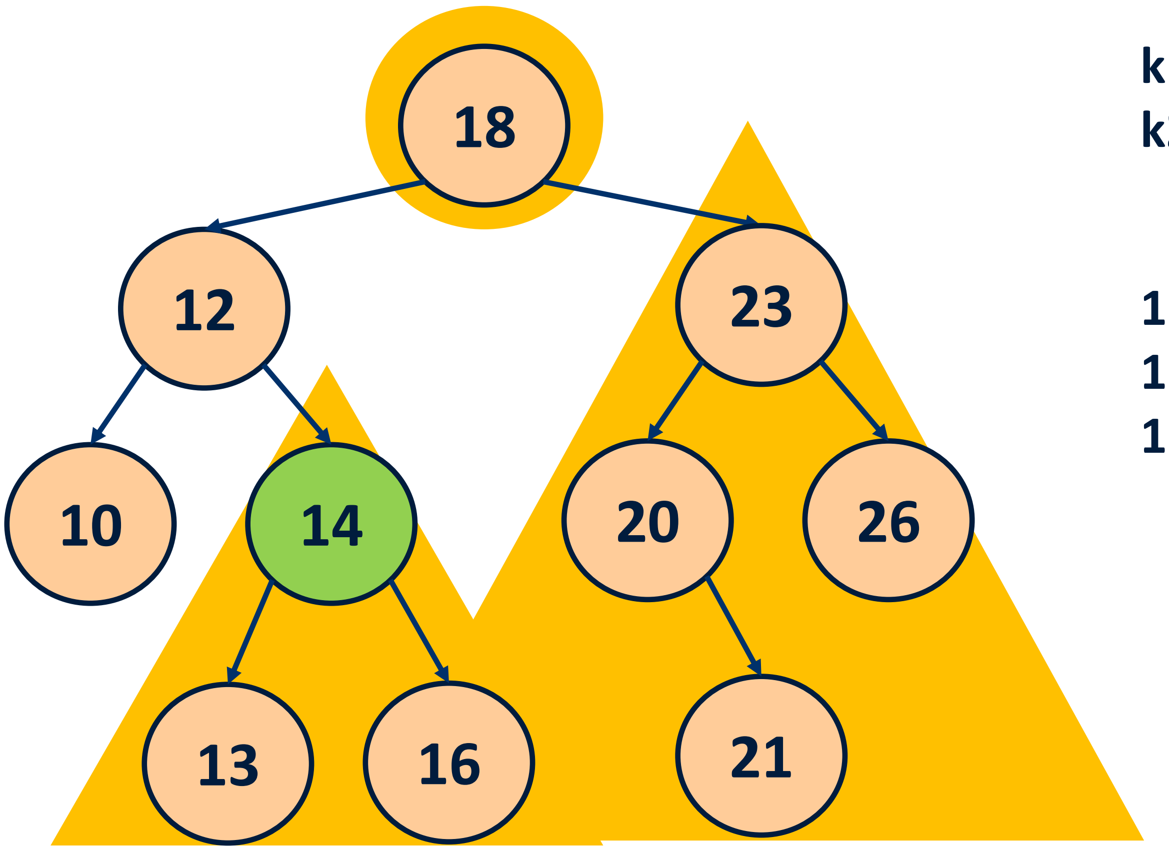
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13

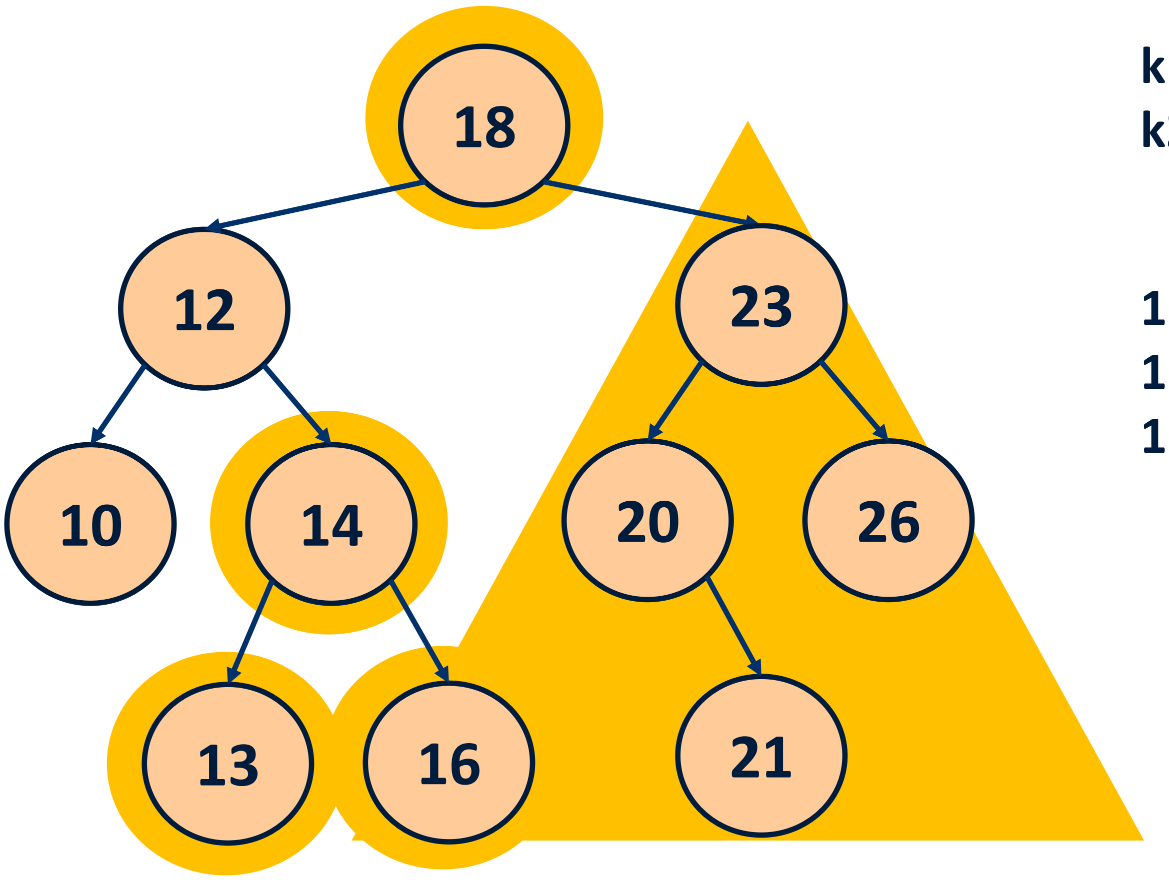
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13
13 ≤ 14 ≤ 22

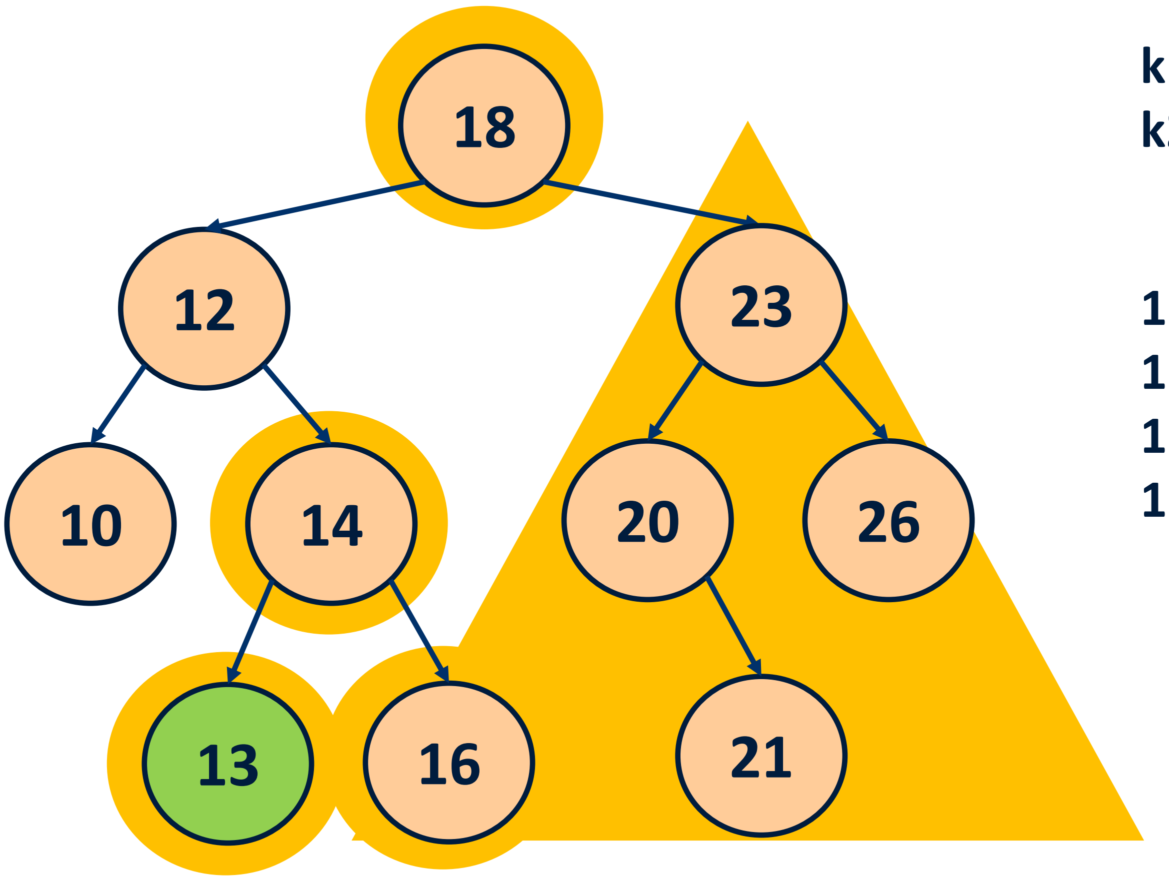
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13
13 ≤ 14 ≤ 22

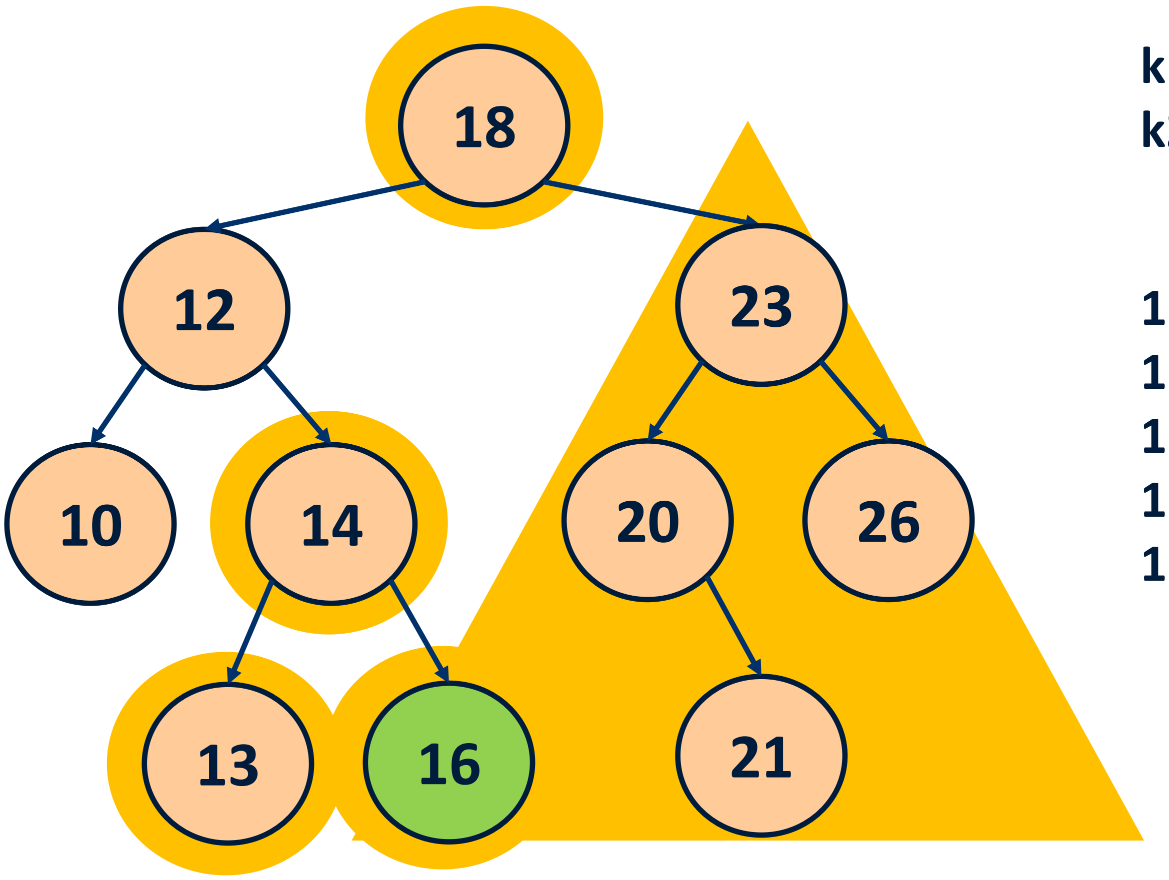
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13
13 ≤ 14 ≤ 22
13 ≤ 13 ≤ 22

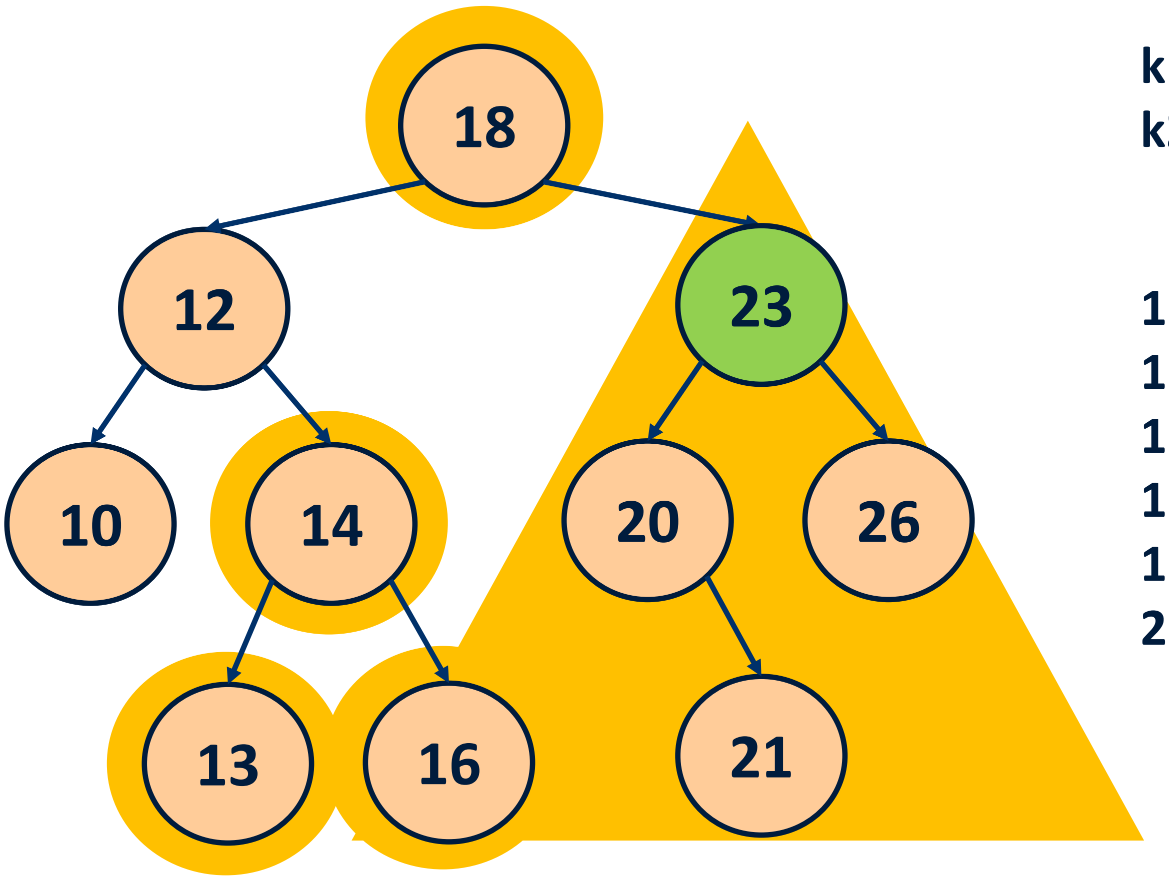
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13
13 ≤ 14 ≤ 22
13 ≤ 13 ≤ 22
13 ≤ 16 ≤ 22

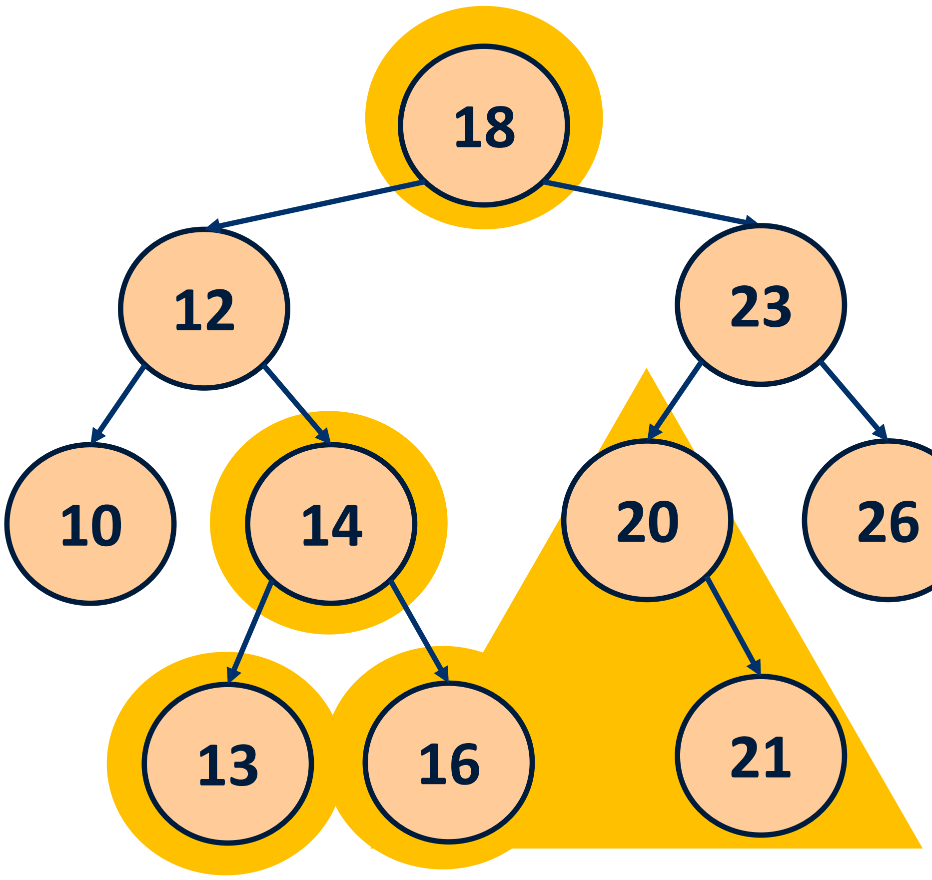
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13
13 ≤ 14 ≤ 22
13 ≤ 13 ≤ 22
13 ≤ 16 ≤ 22
22 < 23

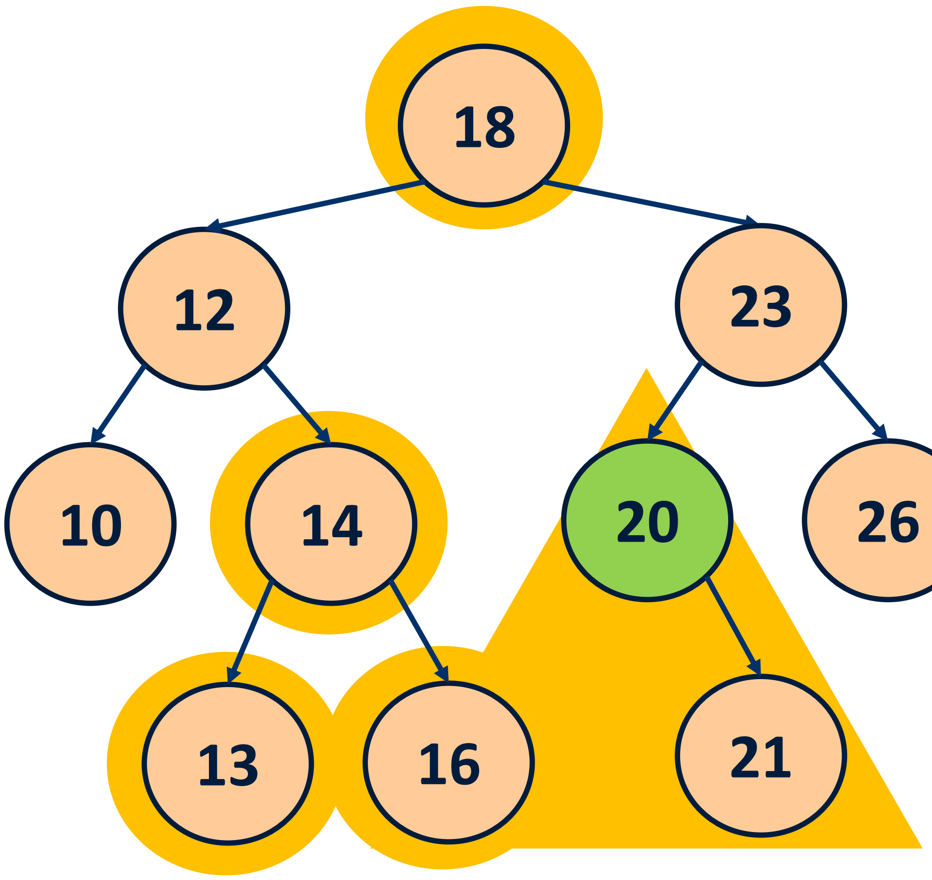
Range Query Algorithm - Example



k1 = 13
k2 = 22

$13 \le 18 \le 22$
$12 < 13$
$13 \le 14 \le 22$
$13 \le 13 \le 22$
$13 \le 16 \le 22$
$22 < 23$

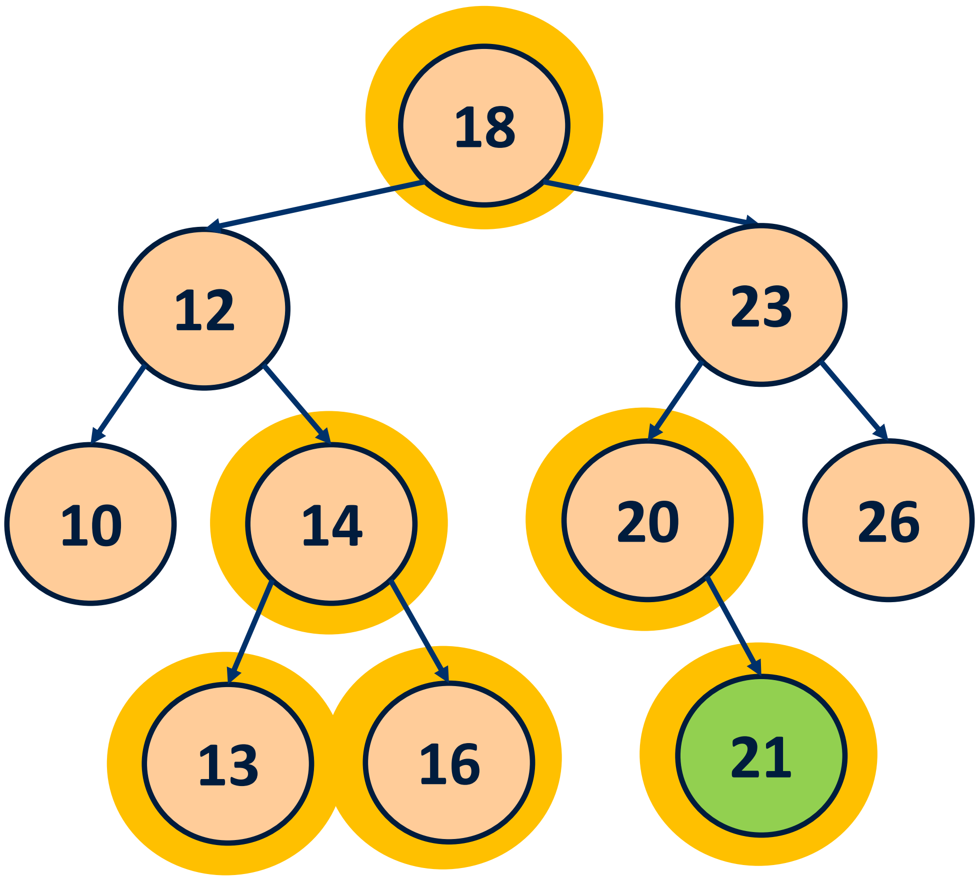
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13
13 ≤ 14 ≤ 22
13 ≤ 13 ≤ 22
13 ≤ 16 ≤ 22
22 < 23
13 ≤ 20 ≤ 22

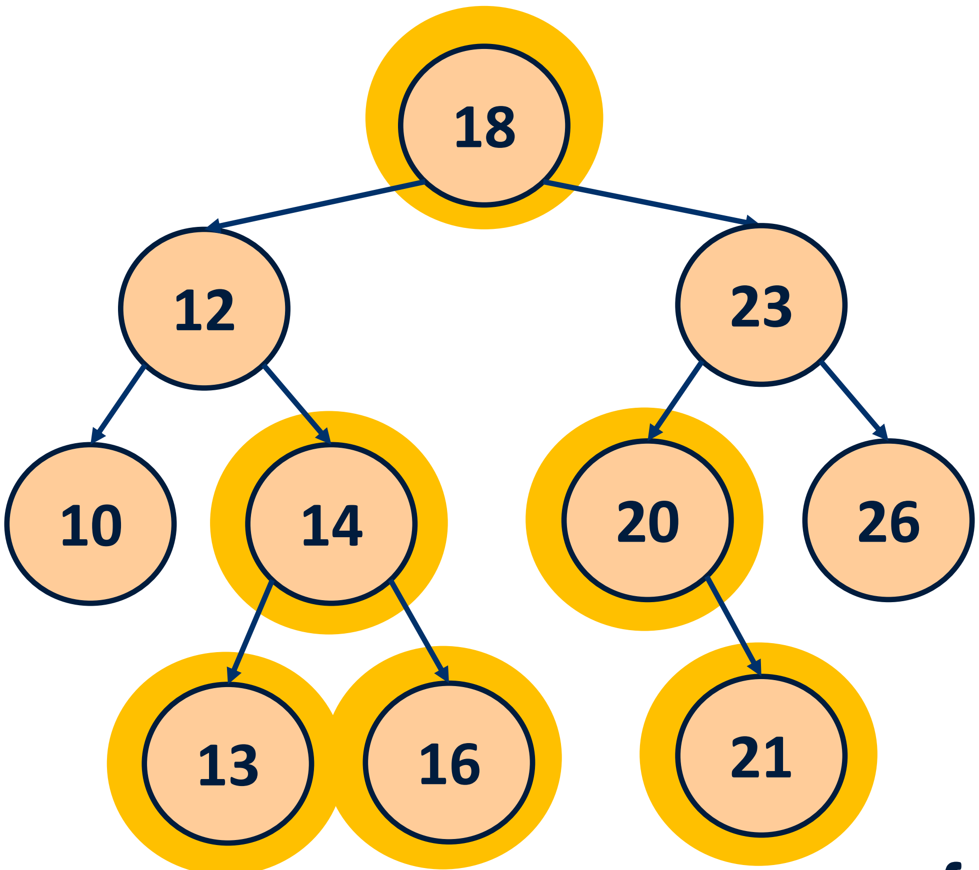
Range Query Algorithm - Example



k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13
13 ≤ 14 ≤ 22
13 ≤ 13 ≤ 22
13 ≤ 16 ≤ 22
22 < 23
13 ≤ 21 ≤ 22

Range Query Algorithm - Example



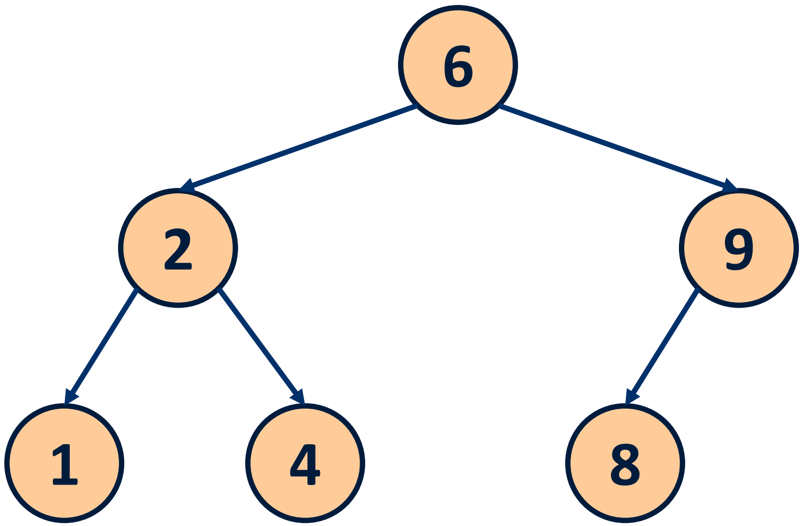
k1 = 13
k2 = 22

13 ≤ 18 ≤ 22
12 < 13
13 ≤ 14 ≤ 22
13 ≤ 13 ≤ 22
13 ≤ 16 ≤ 22
22 < 23
13 ≤ 21 ≤ 22

{13, 14, 16, 18, 20, 21}

Binary Search Tree – In-Order Traversal

- An in-order traversal of a binary search trees visits the keys in increasing order



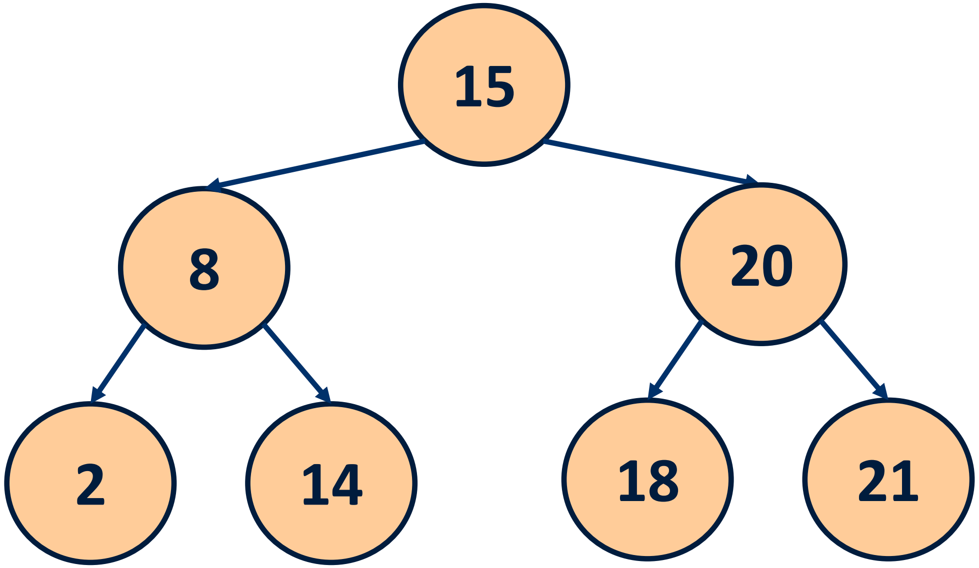
1 - 2 - 4 - 6 - 8 - 9

Let's make a BST!

Same values but added with different order

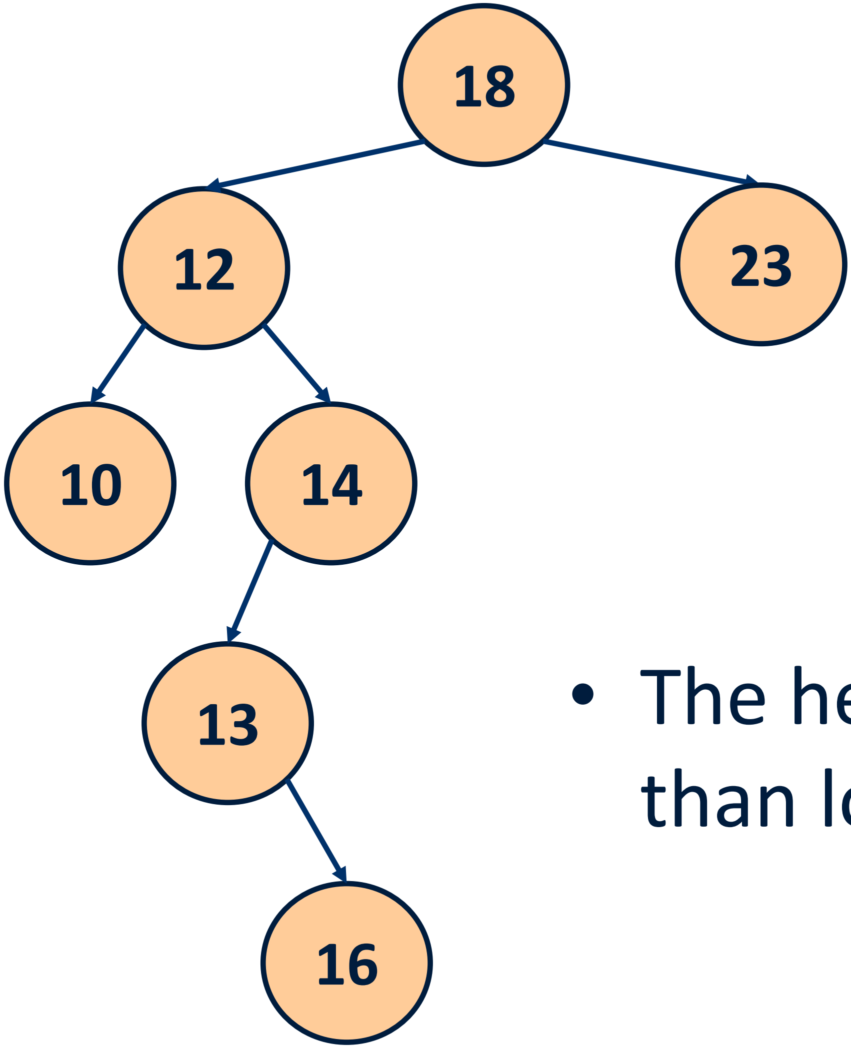
1) 15, 8, 2, 20, 21, 14, 18
2) 20, 8, 21, 18, 14, 15, 2
3) 2, 8, 14, 15, 18, 20, 21

1) A Balanced Tree



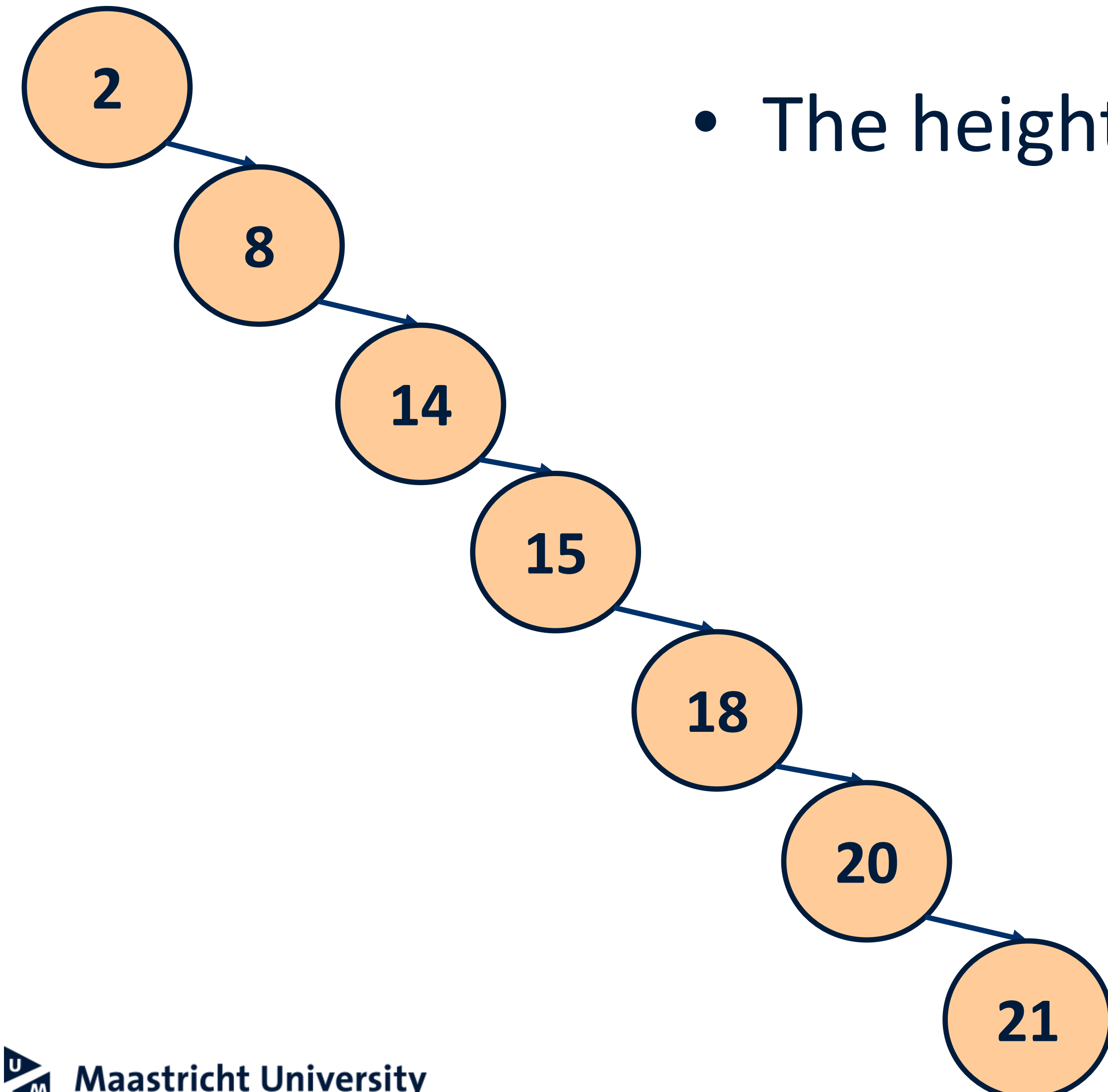
- The height of the tree is log(N)

1) A mostly-Balanced Tree



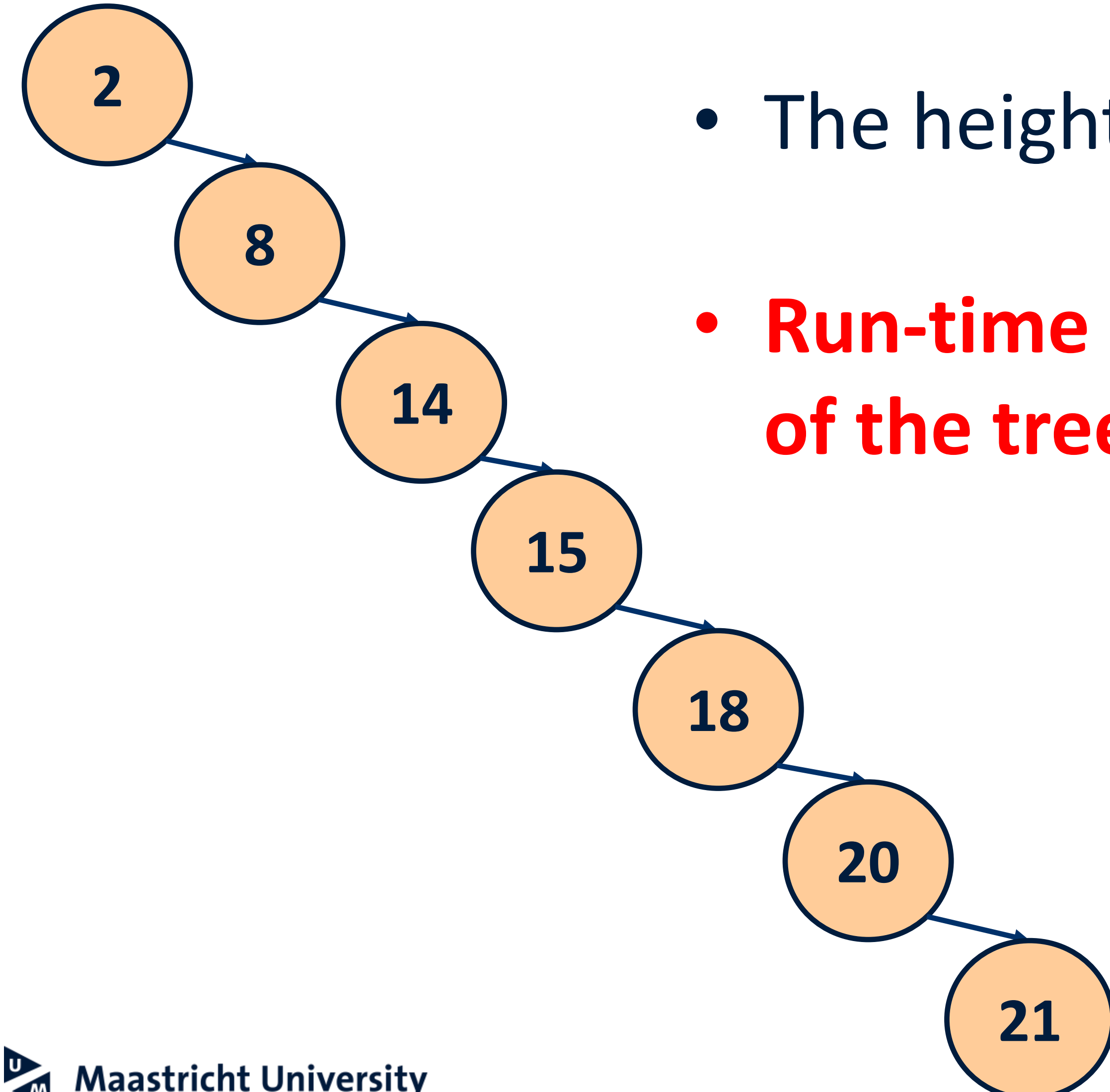
- The height of the tree is more than log(N)

1) A Balanced Tree



- The height of the tree is N

1) A Balanced Tree



- The height of the tree is N
- **Run-time depends on height of the tree!!!**

Binary Trees: Performance

- For binary tree of height ***h*** and with ***n*** nodes:
  - max # of leaves: 2^h
  - max # of nodes: $2^{(h+1)} - 1$
  - min # of leaves: **1**
  - min # of nodes: $h + 1$

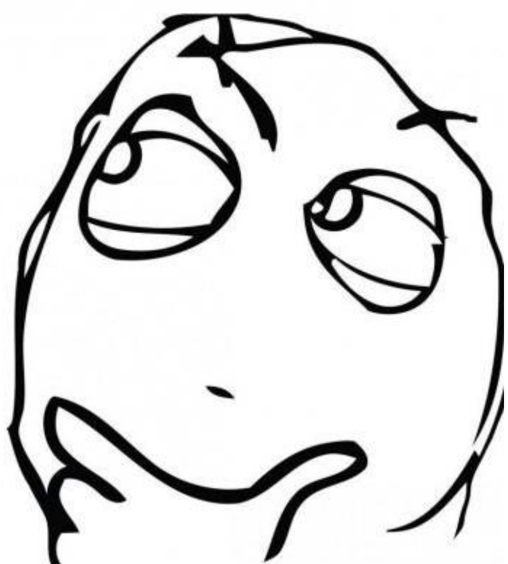
- Methods **find**, **insert** and **remove** have ***O(h)*** complexity
- The height ***h*** is ***O(n)*** in the worst case and ***O(log n)*** in the best case

Binary Trees: Performance

- What if we could make sure binary trees remain balanced???

  - Their ***height*** should be always ***O(log n)***
  - Then we can perform all operations in ***O(log n) time***

- Dream or reality?



AVL Trees

AVL Tree

- An AVL tree is a *self-balancing* binary search tree
- Named after its two Soviet inventors, Georgy Adelson-Velsky and Evgenii Landis

- Published it in the 1962 paper "An algorithm for the organization of information"

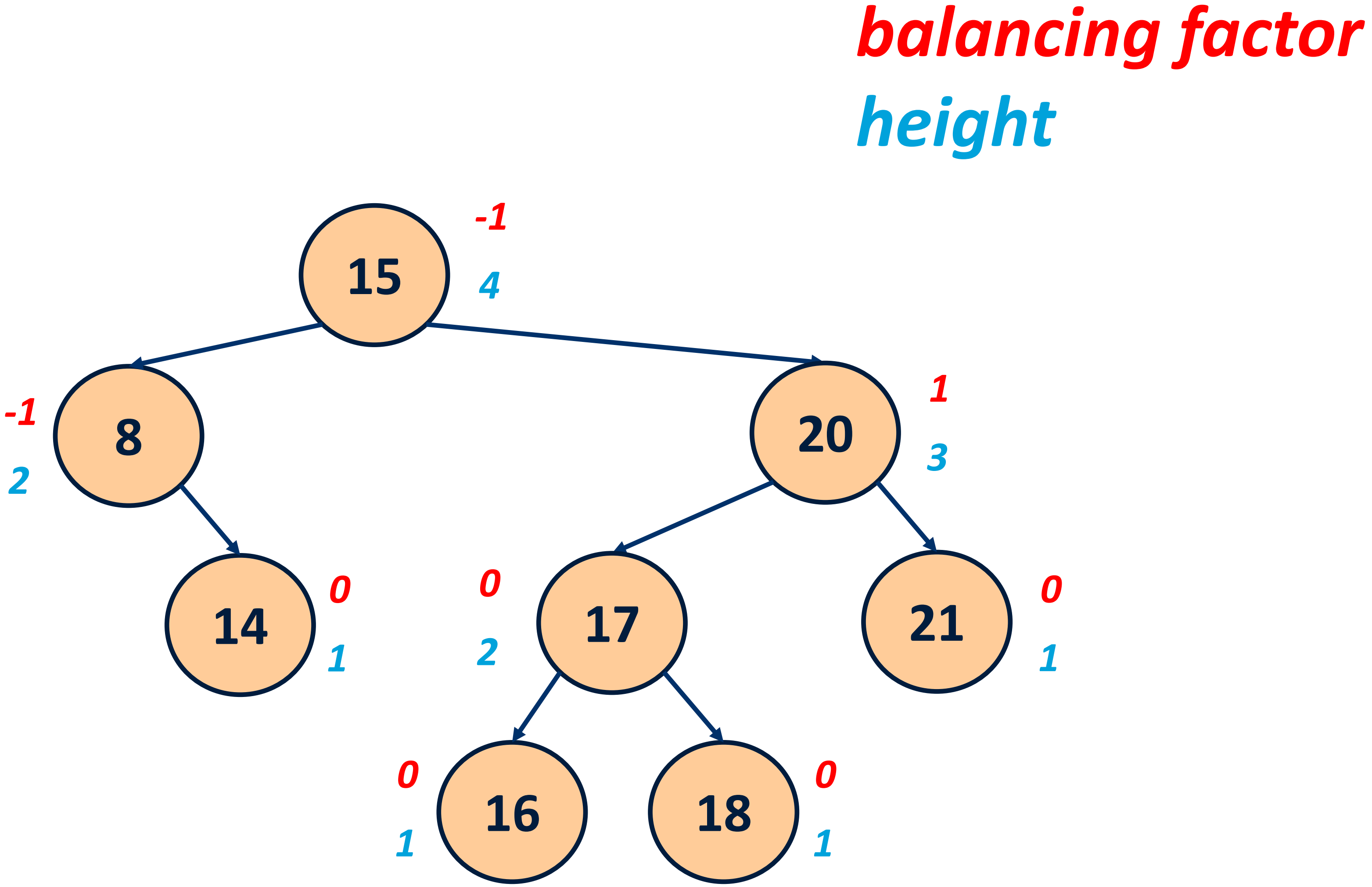
AVL Tree

- An AVL Tree is a BST such that
  - for every internal node ***v***, the **heights of the children of *v* can differ by at most 1**
- The height of an AVL tree with ***n*** nodes is ***O(log n)***

AVL Tree

- In each node we store
  - ***Height*** of the node
    - we assume here ***leafs have height=1***
  - ***Balancing factor***
    - ***height*** of left subtree – ***height*** of right subtree

AVL Tree



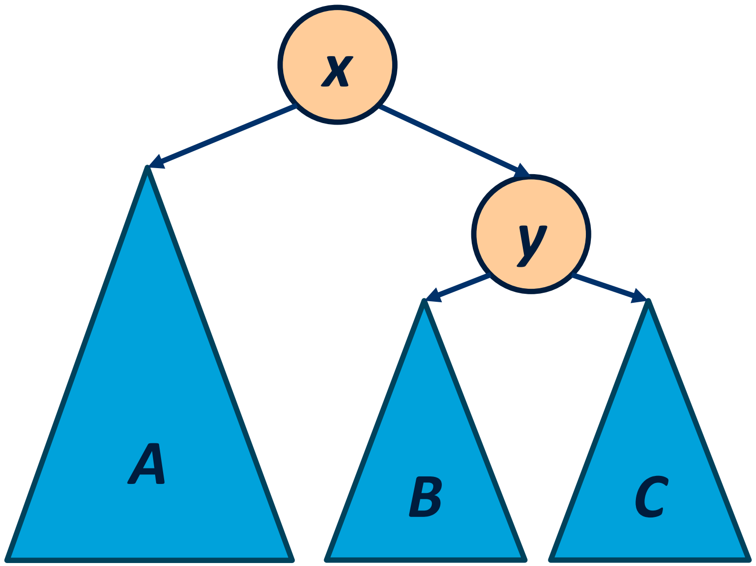
balancing factor
height
15
-1
4
8
-1
2
20
1
3
14
0
1
17
0
2
21
0
1
16
0
1
18
0
1

AVL Tree

- Assuming we have an AVL tree
  - When we modify the tree, we update ***height*** and ***balancing factor*** for all the nodes on the path to the root
    - from the parent to the root of the inserted/deleted node
- If we find a node with a ***balancing factor*** bigger than 2 (lower than -2)?
  - We restore the balancing using a **rotation**!

AVL Tree - ROTATION

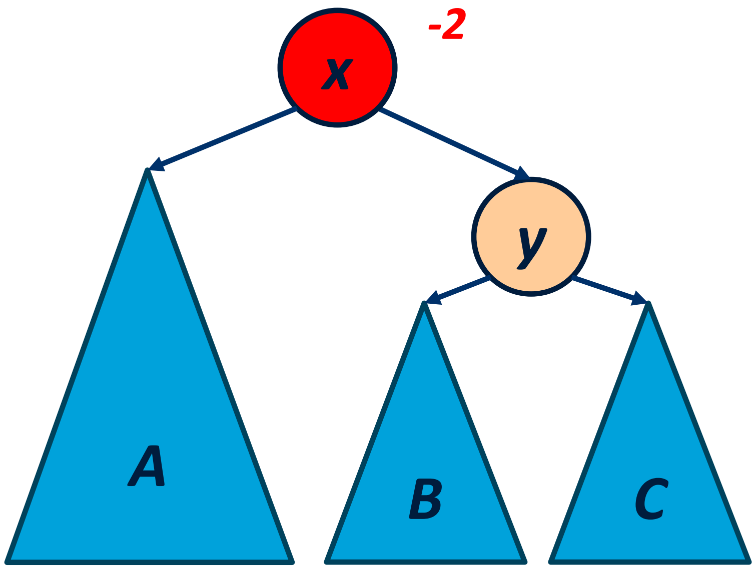
- Assume the following is a binary tree



$$A \leq x \leq B \leq y \leq C$$

AVL Tree - ROTATION

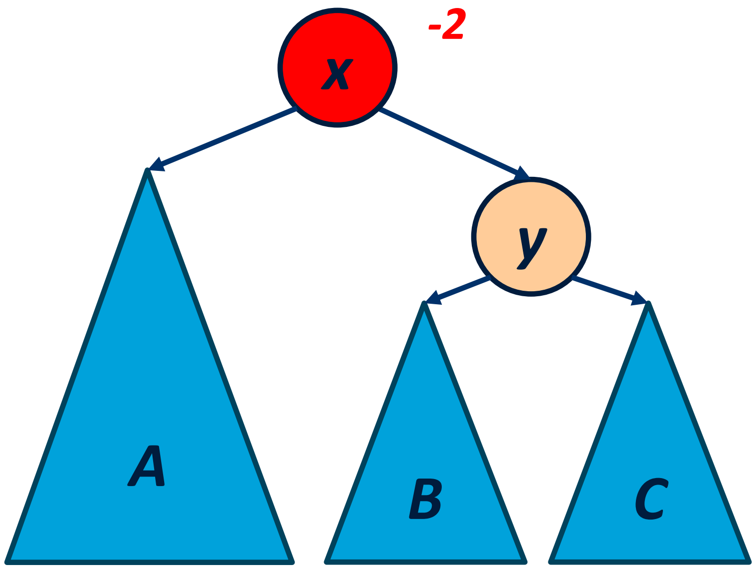
- Assuming we have a negative unbalance in x



$$A \leq x \leq B \leq y \leq C$$

AVL Tree - ROTATION

- Assuming we have a negative unbalance in x

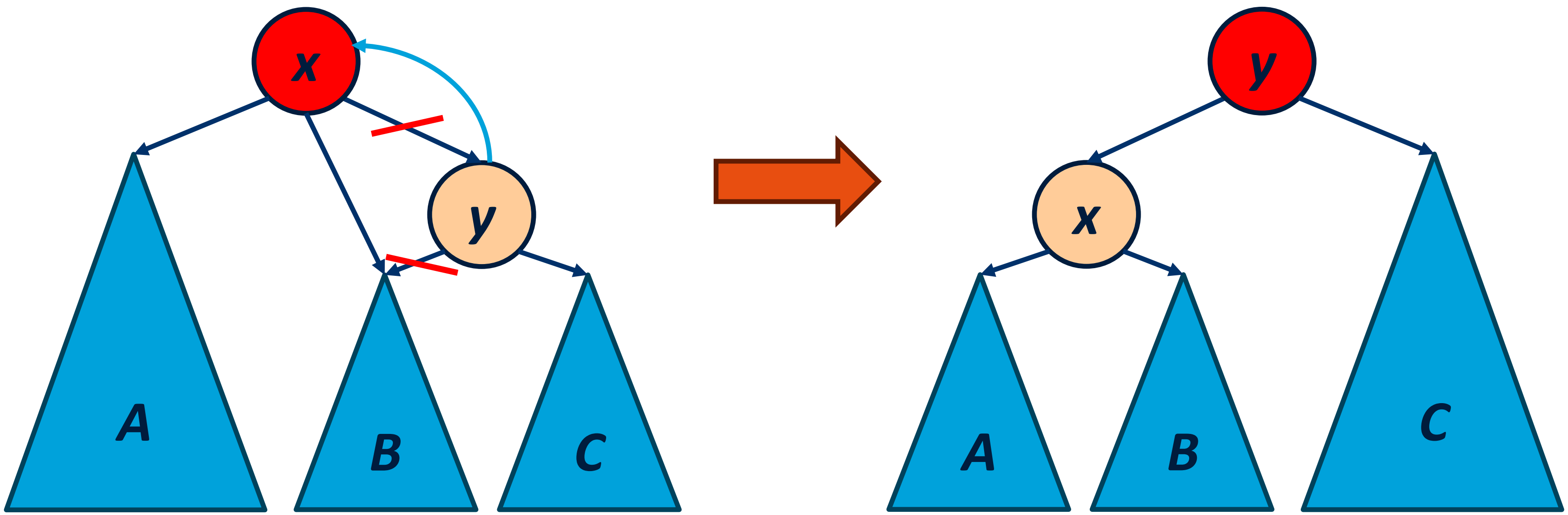


LEFT ROTATION

$$A \leq x \leq B \leq y \leq C$$

AVL Tree - ROTATION

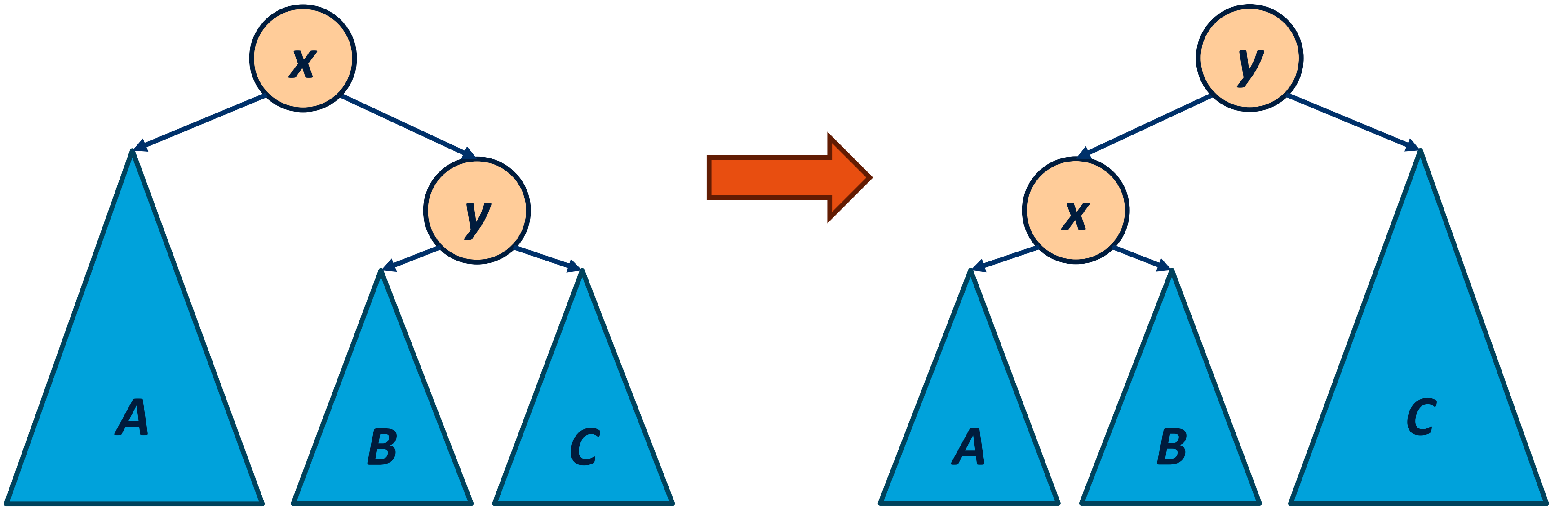
- Assuming we have a negative unbalance in x



$$A \le x \le B \le y \le C$$

AVL Tree - ROTATION

- Assuming we have a negative unbalance in x

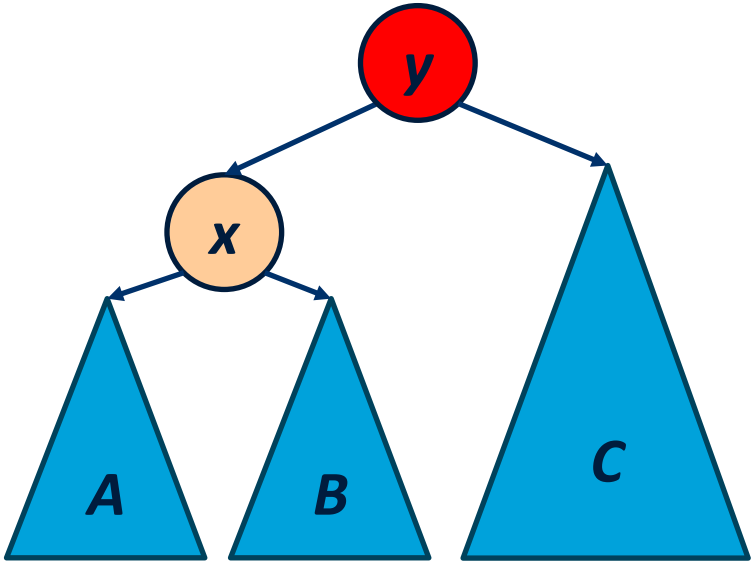


The ordering is preserved!

$$A \leq x \leq B \leq y \leq C$$

AVL Tree - ROTATION

- If we have a positive unbalance in y?

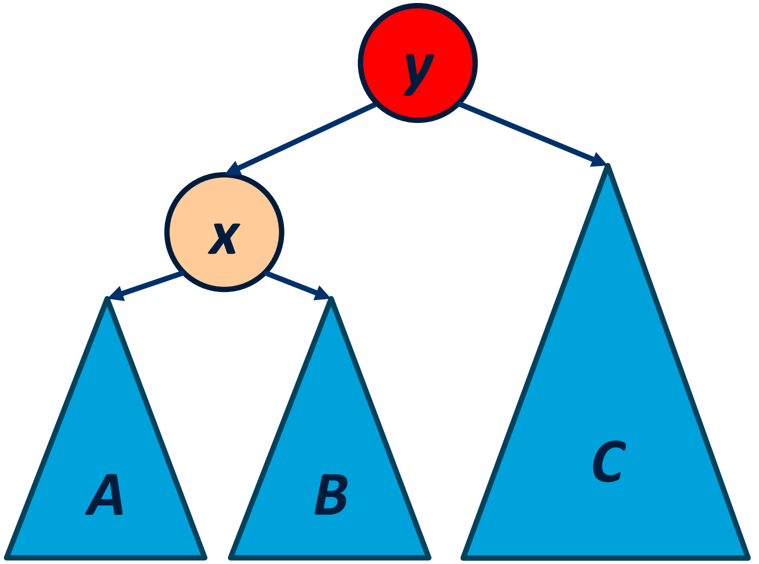


$$A \leq x \leq B \leq y \leq C$$

AVL Tree - ROTATION

- If we have a positive unbalance in y?

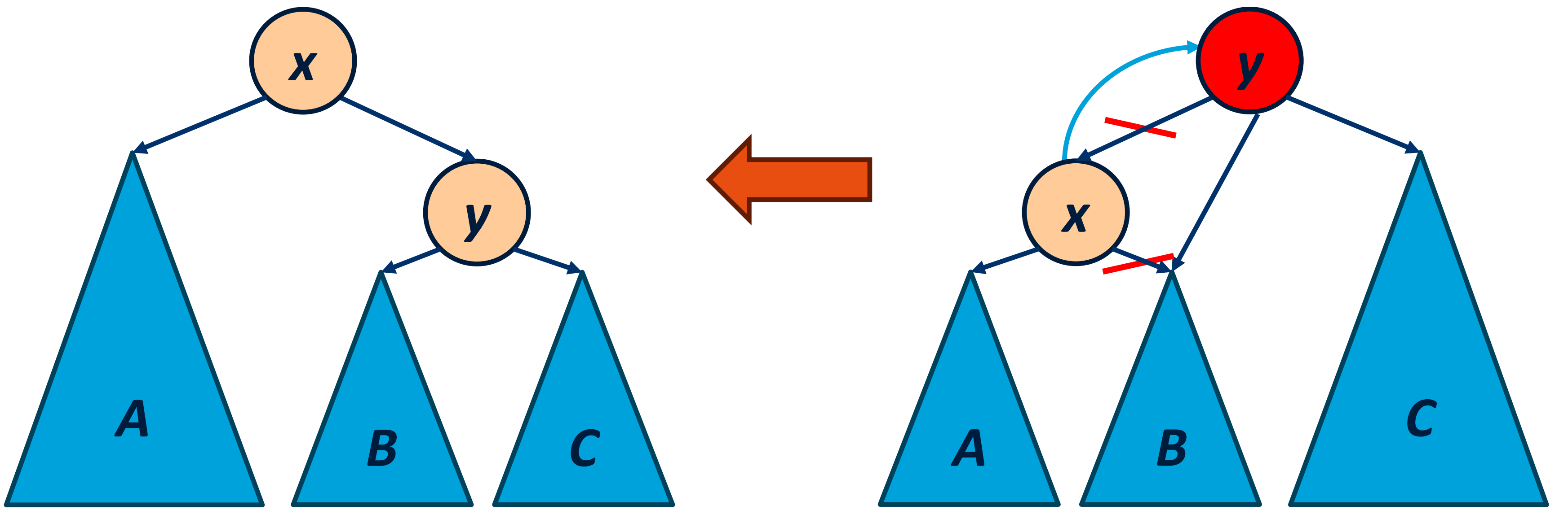
RIGHT ROTATION



$$A \leq x \leq B \leq y \leq C$$

AVL Tree - ROTATION

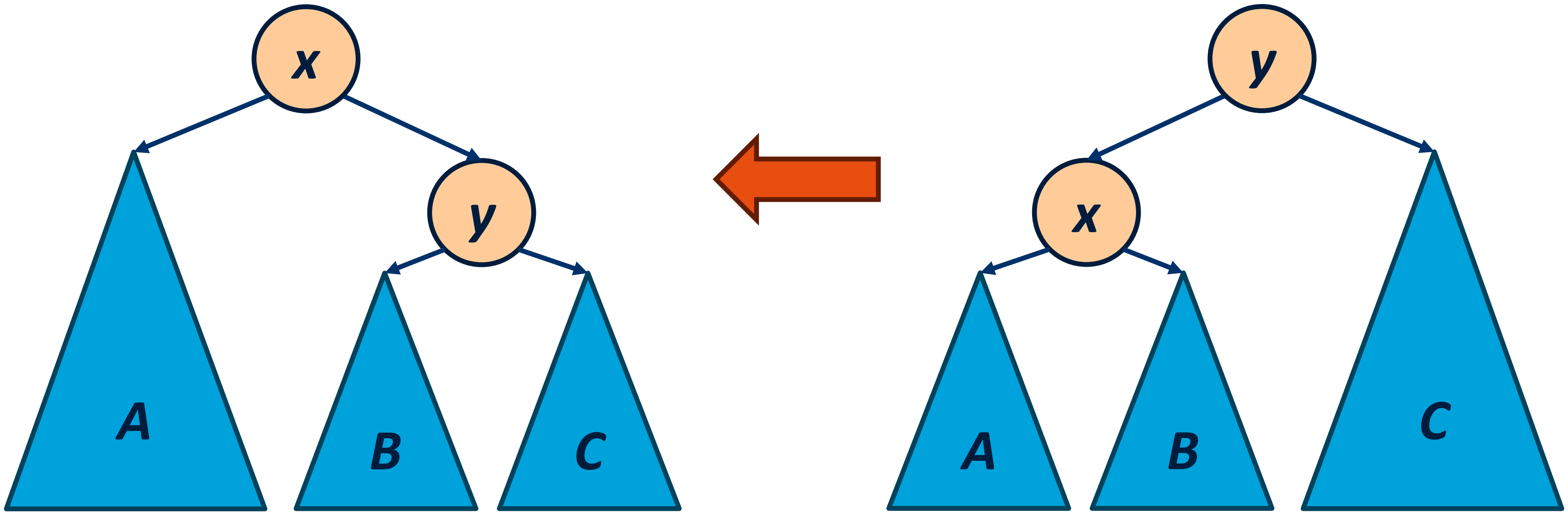
- If we have a positive unbalance in y?



$$A \leq x \leq B \leq y \leq C$$

AVL Tree - ROTATION

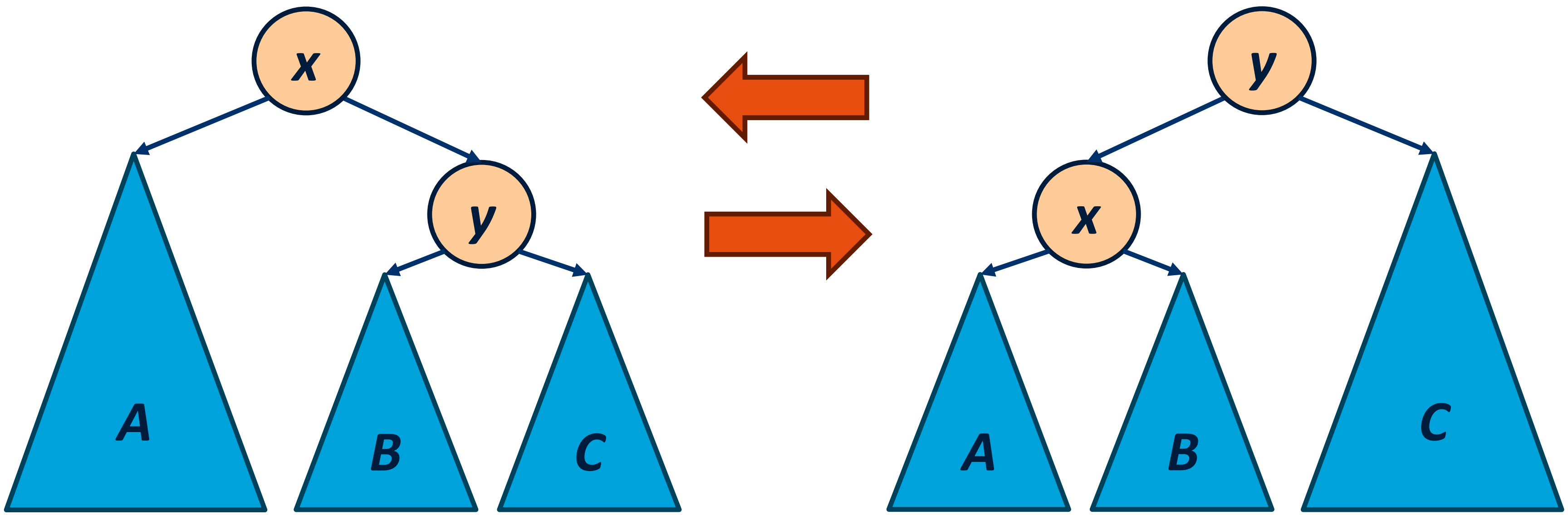
- Assuming we have a positive unbalance in x



The ordering is preserved!

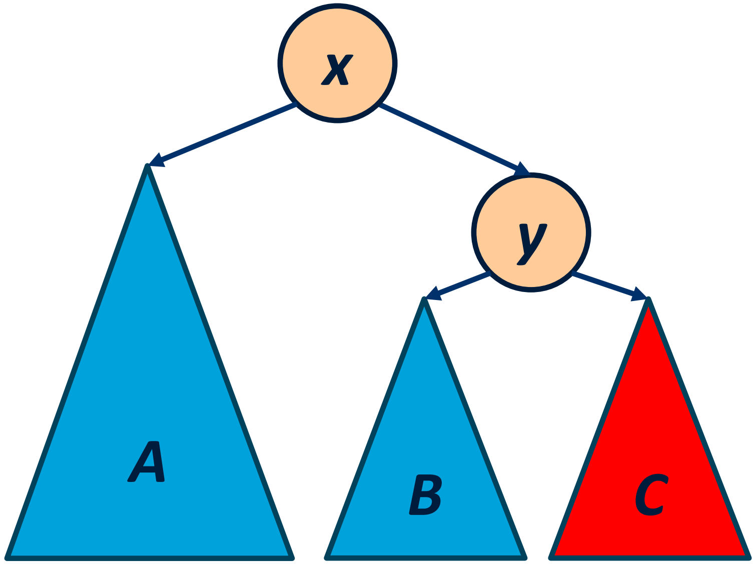
$$A \le x \le B \le y \le C$$

AVL Tree - ROTATION



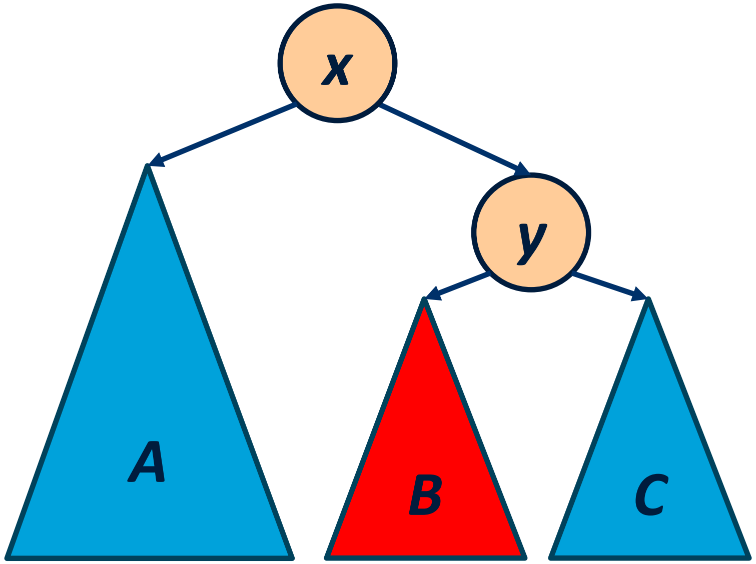
- Rotation operations keep the ordering while modifying the height

AVL Tree - ROTATION



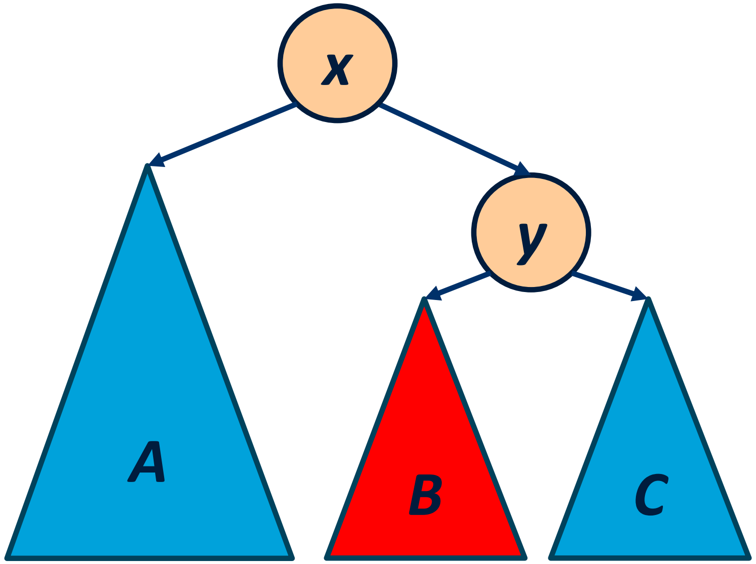
- A left rotation restores the balance if the imbalance comes from C

AVL Tree - ROTATION



- A left rotation restores the balance if the imbalance comes from C
- What if the imbalance is in B?

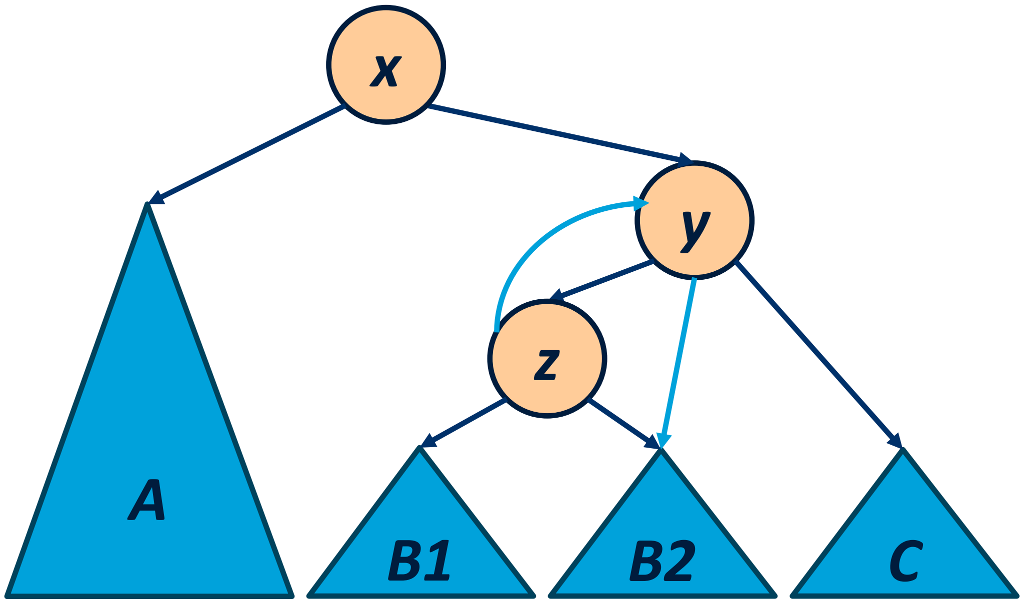
AVL Tree - ROTATION



- A left rotation restores the balance if the imbalance comes from C
- What if the imbalance is in B?

- **RIGHT-LEFT ROTATION!**

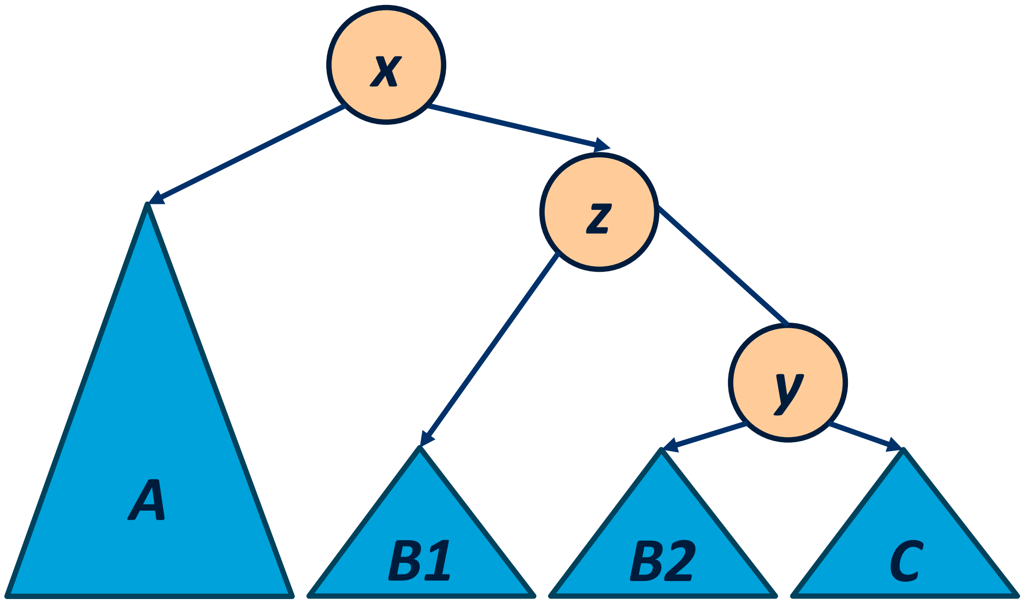
AVL Tree - ROTATION



- We first perform a right rotation on y

- **RIGHT-LEFT ROTATION!**

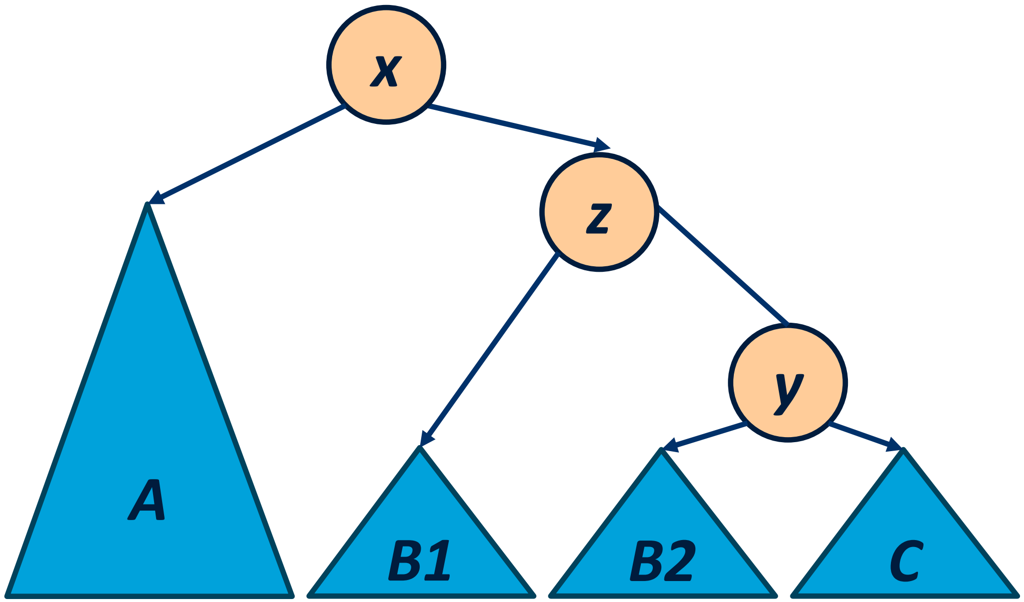
AVL Tree - ROTATION



- We first perform a right rotation on y

- **RIGHT-LEFT ROTATION!**

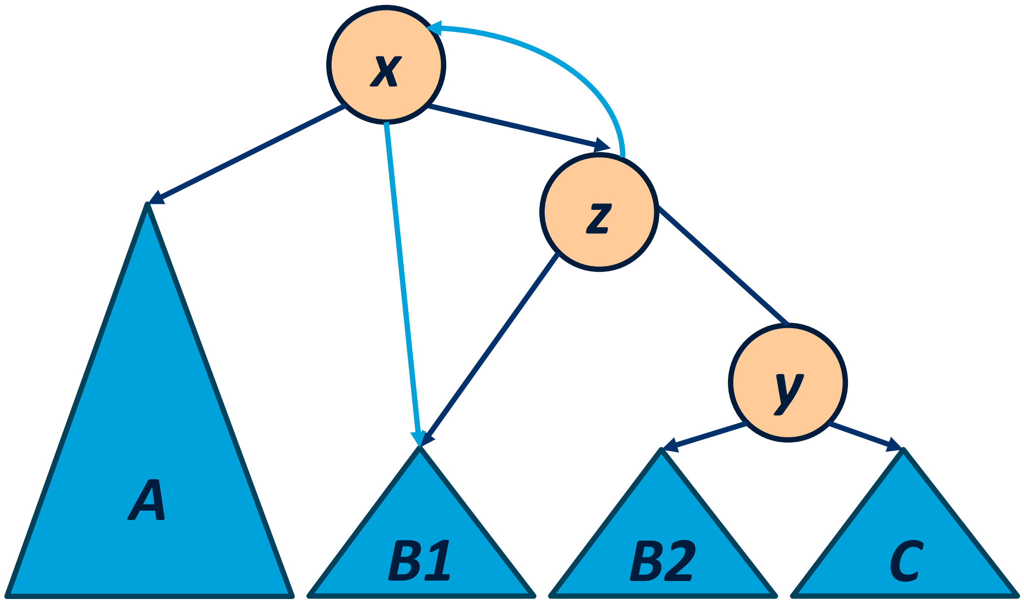
AVL Tree - ROTATION



- We first perform a right rotation on y
- Then we perform a left rotation on x

- **RIGHT-LEFT ROTATION!**

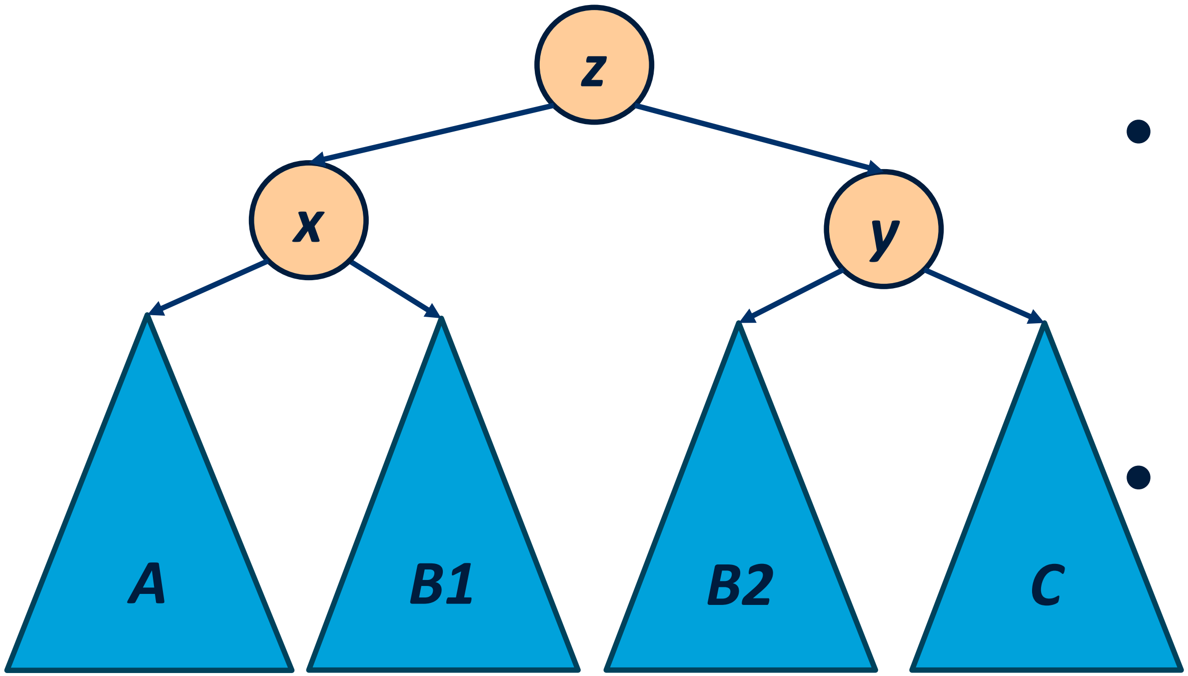
AVL Tree - ROTATION



- We first perform a right rotation on y
- Then we perform a left rotation on x

- **RIGHT-LEFT ROTATION!**

AVL Tree - ROTATION



- We first perform a right rotation on y
- Then we perform a left rotation on x

- **RIGHT-LEFT ROTATION!**

AVL Tree - Example

- Let’s insert values 1, 2, 3, 4, 5, 6

AVL Tree - Example

balancing factor
height

- Let’s insert values 1, 2, 3, 4, 5, 6

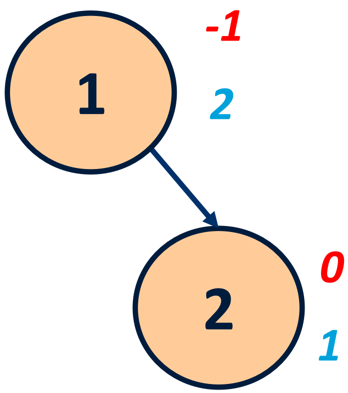
Insert(1)

AVL Tree - Example

balancing factor

height

- Let’s insert values 1, 2, 3, 4, 5, 6



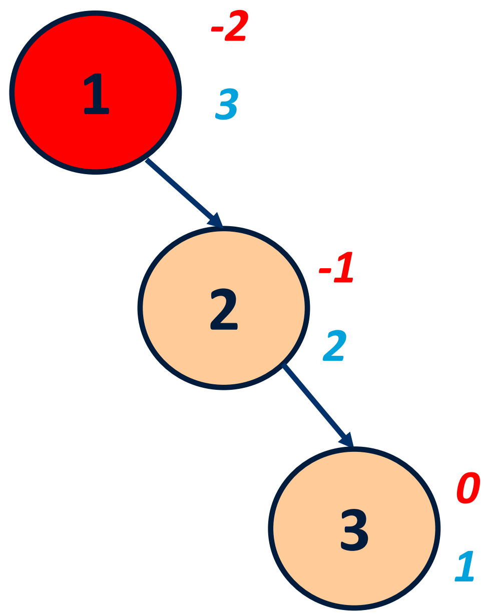
Insert(2)

AVL Tree - Example

balancing factor

height

- Let's insert values 1, 2, 3, 4, 5, 6

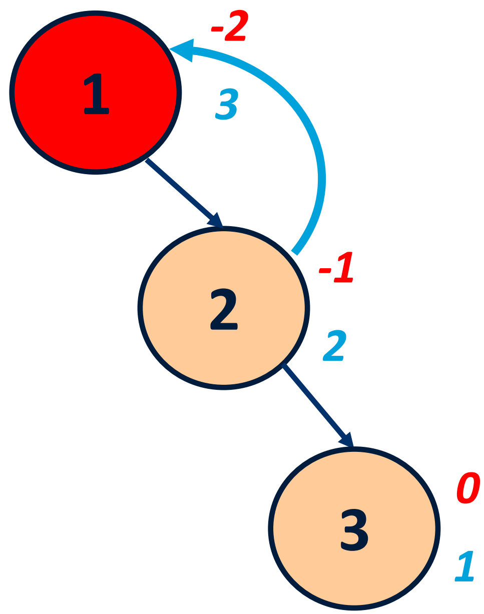


Insert(3)

AVL Tree - Example

balancing factor
height

- Let's insert values 1, 2, 3, 4, 5, 6



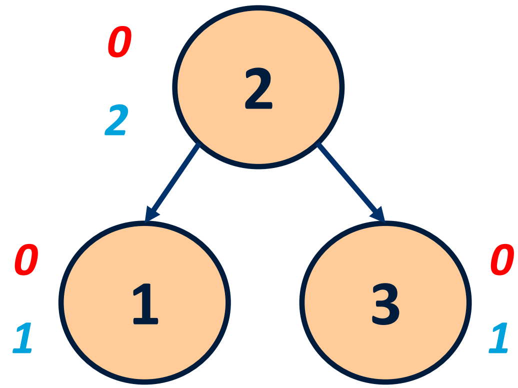
LEFT ROTATION

Insert(3)

AVL Tree - Example

balancing factor
height

- Let's insert values 1, 2, 3, 4, 5, 6

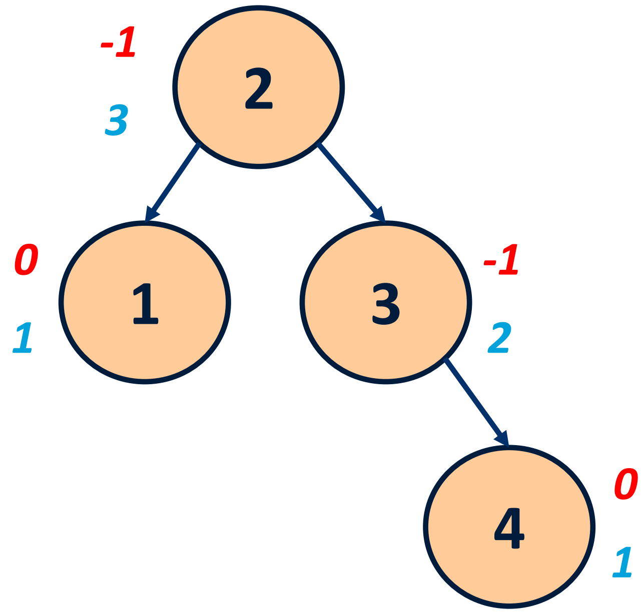


Insert(3)

AVL Tree - Example

balancing factor
height

- Let’s insert values 1, 2, 3, 4, 5, 6



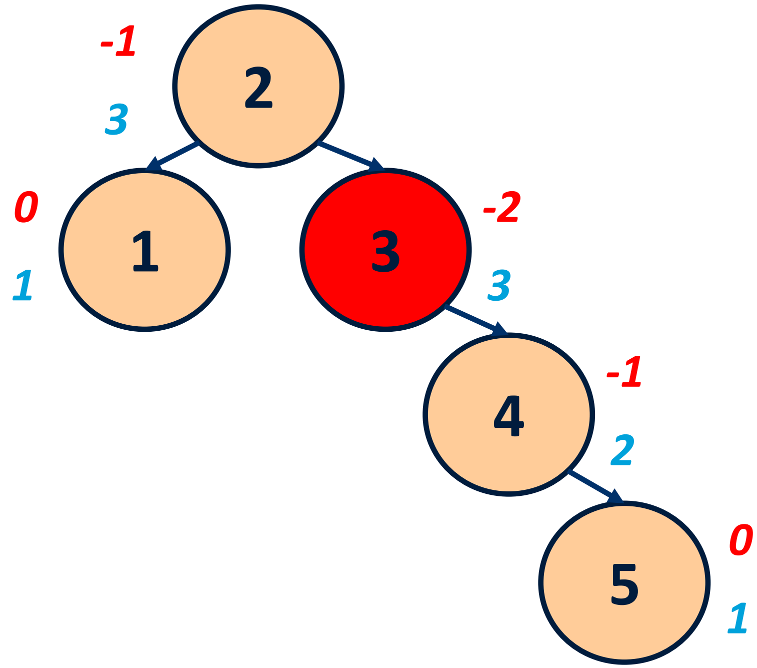
Insert(4)

AVL Tree - Example

balancing factor

height

- Let's insert values 1, 2, 3, 4, 5, 6



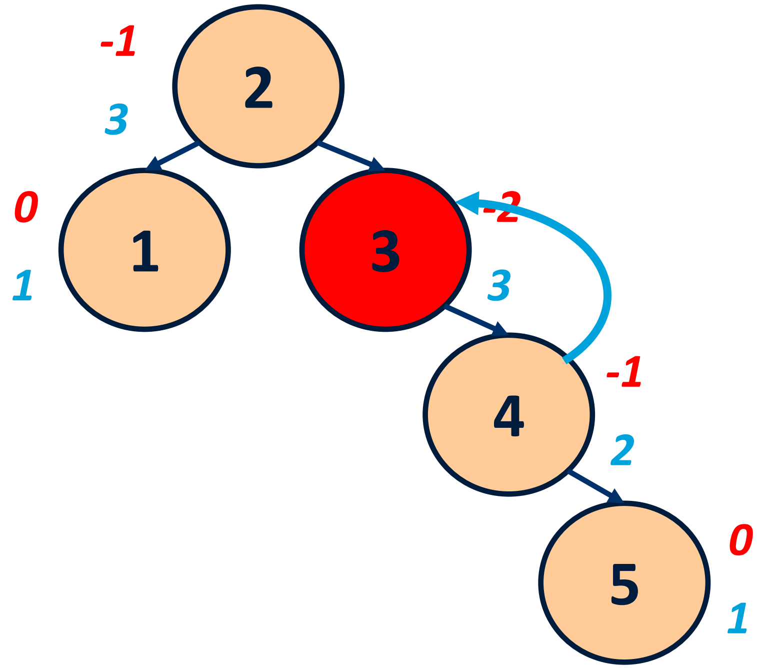
Insert(5)

AVL Tree - Example

balancing factor

height

- Let's insert values 1, 2, 3, 4, 5, 6



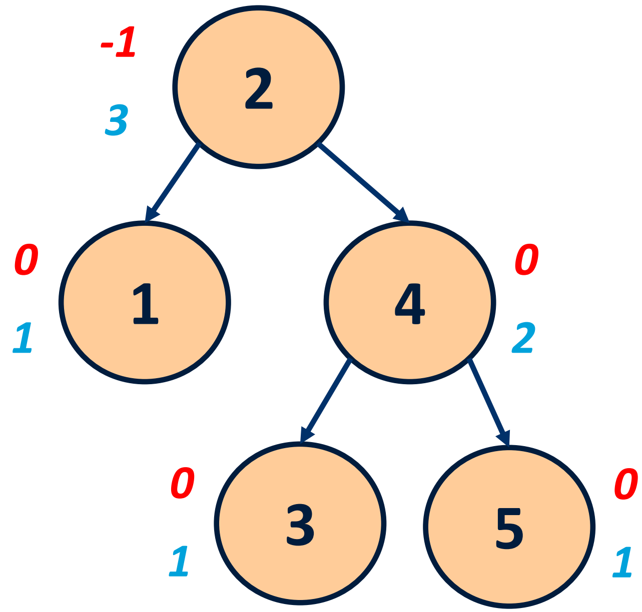
LEFT ROTATION

Insert(5)

AVL Tree - Example

balancing factor
height

- Let’s insert values 1, 2, 3, 4, 5, 6

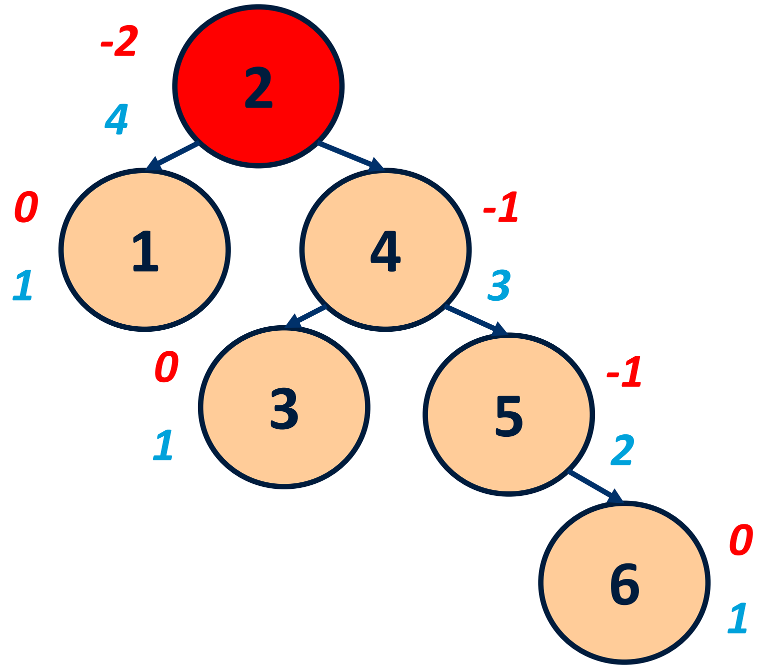


Insert(5)

AVL Tree - Example

balancing factor
height

- Let's insert values 1, 2, 3, 4, 5, 6

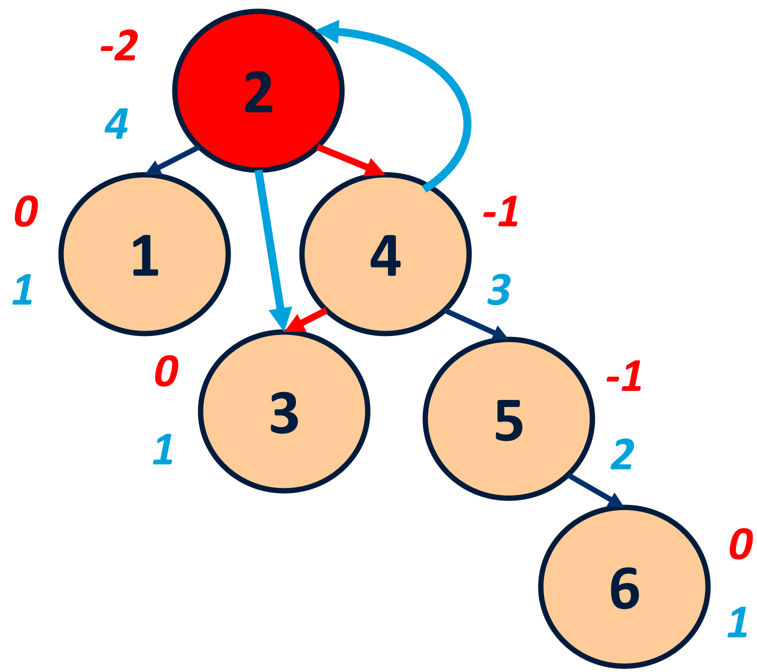


Insert(6)

AVL Tree - Example

balancing factor
height

- Let's insert values 1, 2, 3, 4, 5, 6



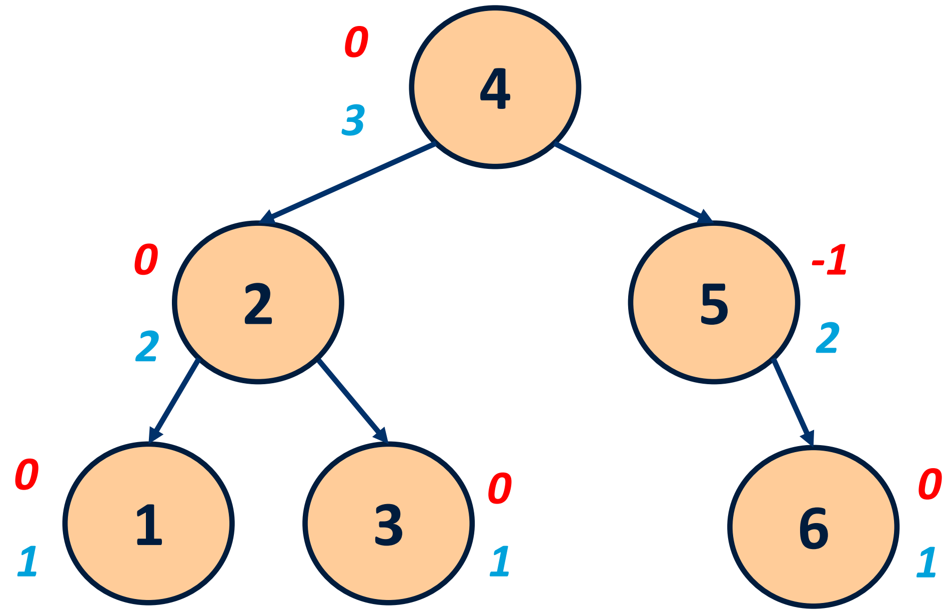
LEFT ROTATION

Insert(6)

AVL Tree - Example

balancing factor
height

- Let's insert values 1, 2, 3, 4, 5, 6

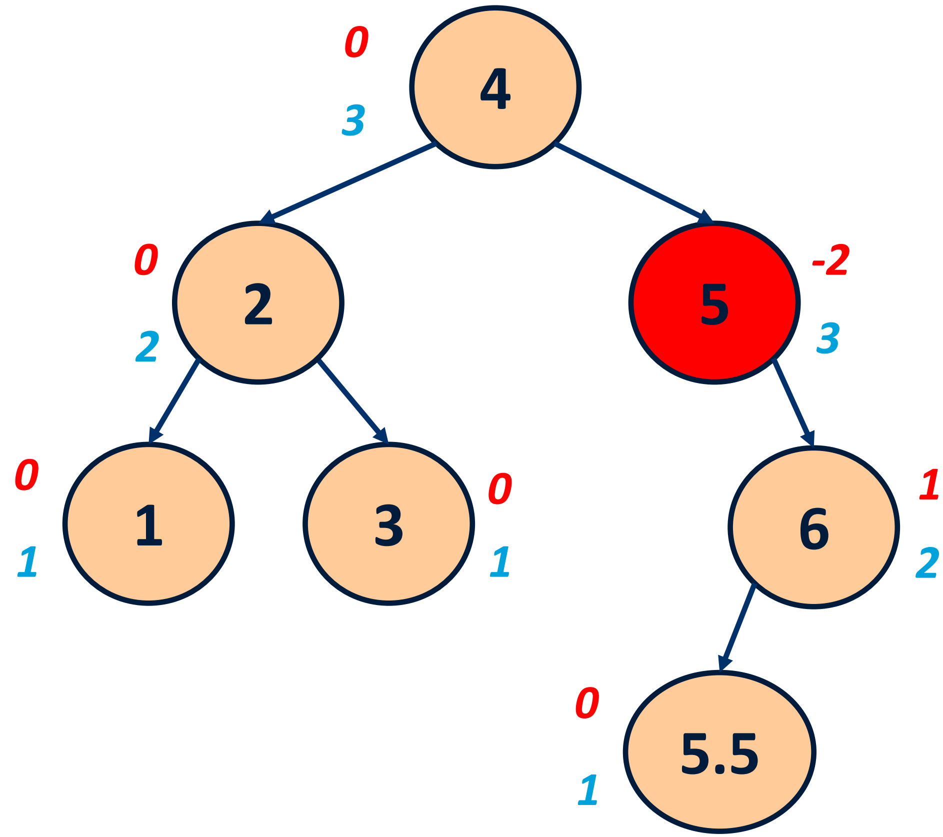


Insert(6)

AVL Tree - Example

balancing factor
height

- Now we insert 5.5



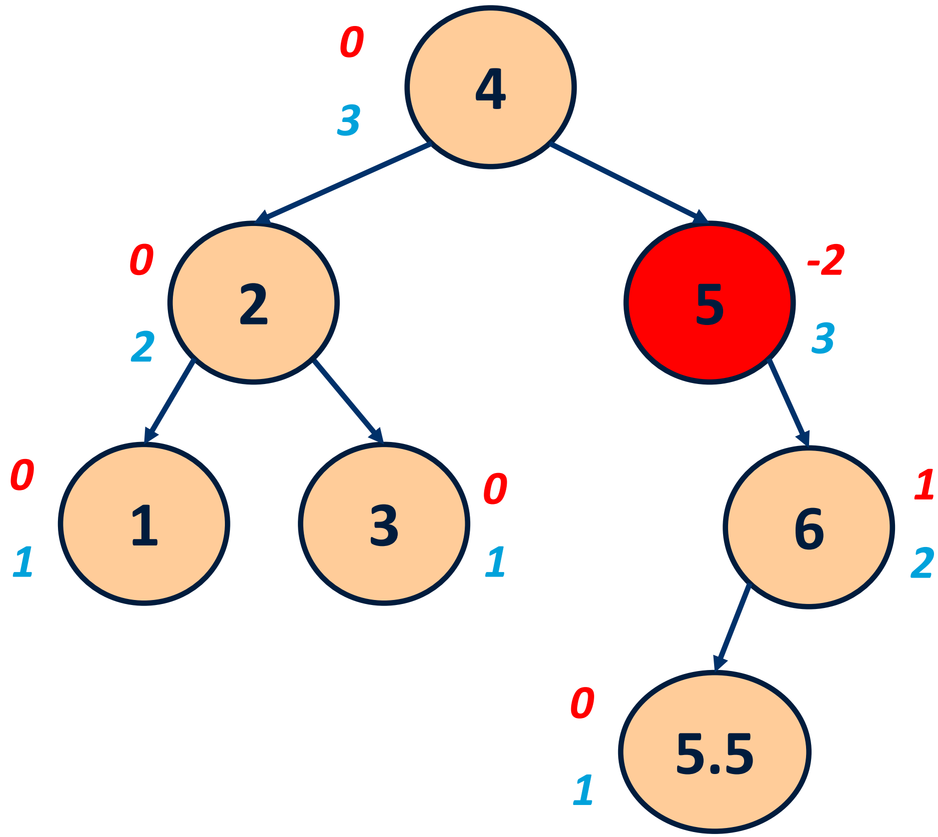
- 5 has balancing factor -2
- The previous updated node has balancing factor 1

Insert(5.5)

AVL Tree - Example

balancing factor
height

- Now we insert 5.5



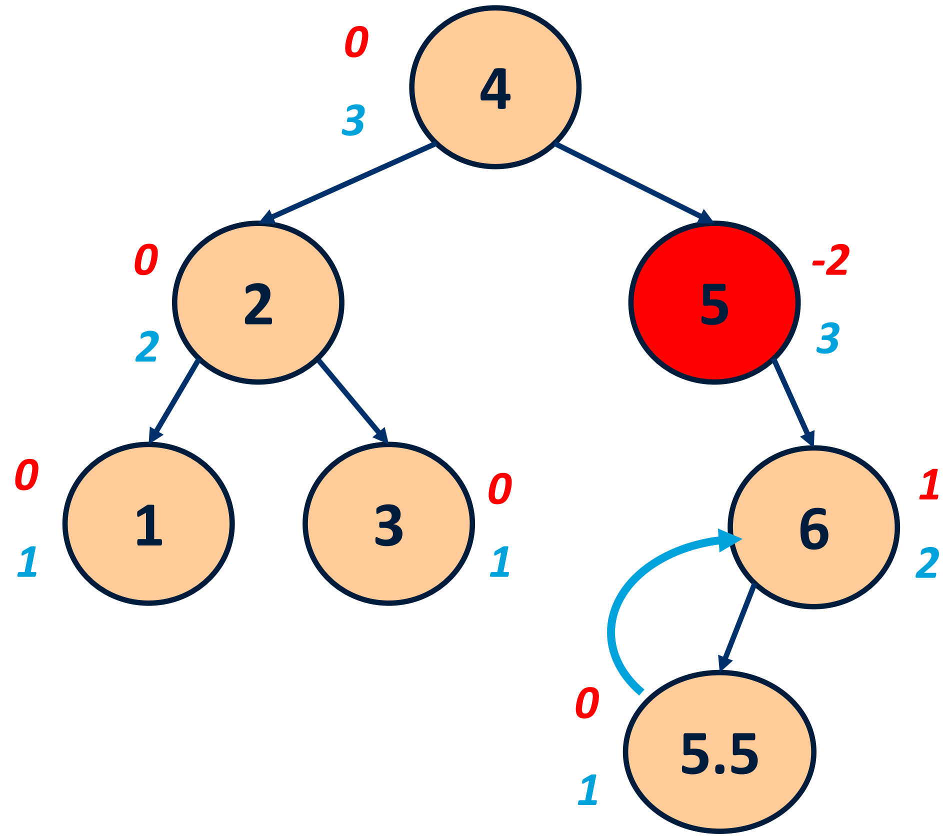
- 5 has **balancing factor -2**
- The previous updated node has **balancing factor 1**

Insert(5.5)

AVL Tree - Example

balancing factor
height

- Now we insert 5.5



- If the signs are not the same, we need a **right-left rotation!**

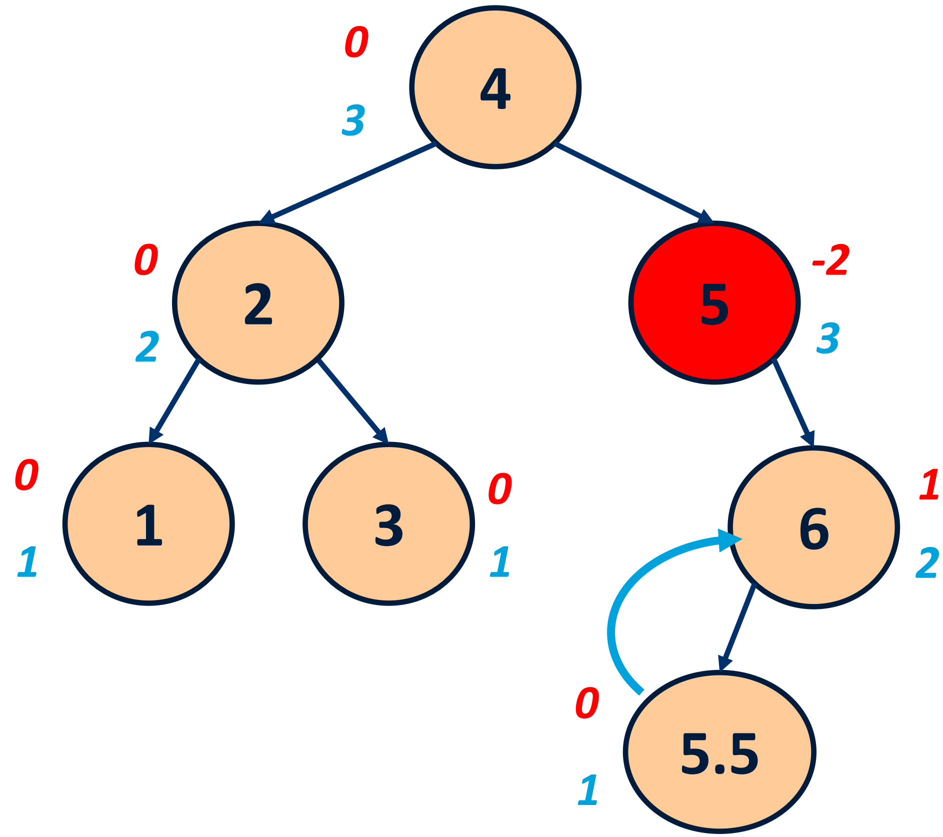
Insert(5.5)

AVL Tree - Example

balancing factor

height

- Now we insert 5.5



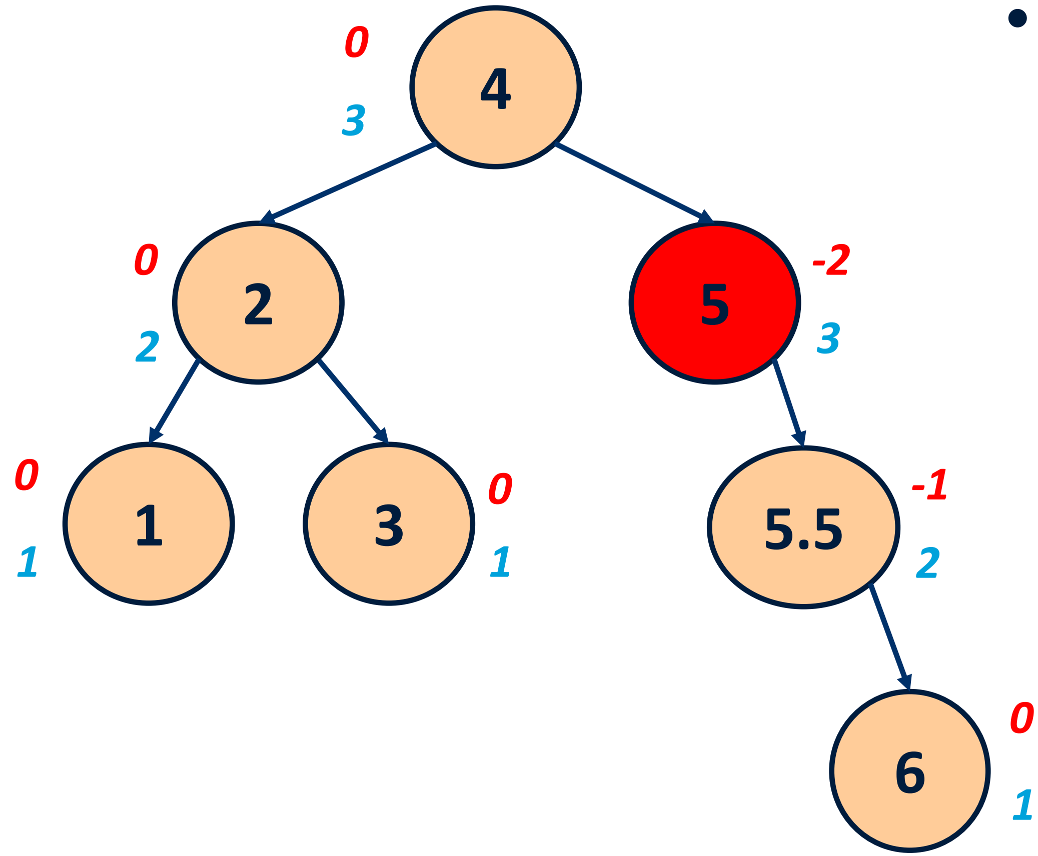
- Right rotation on 6

Insert(5.5)

AVL Tree - Example

balancing factor
height

- Now we insert 5.5



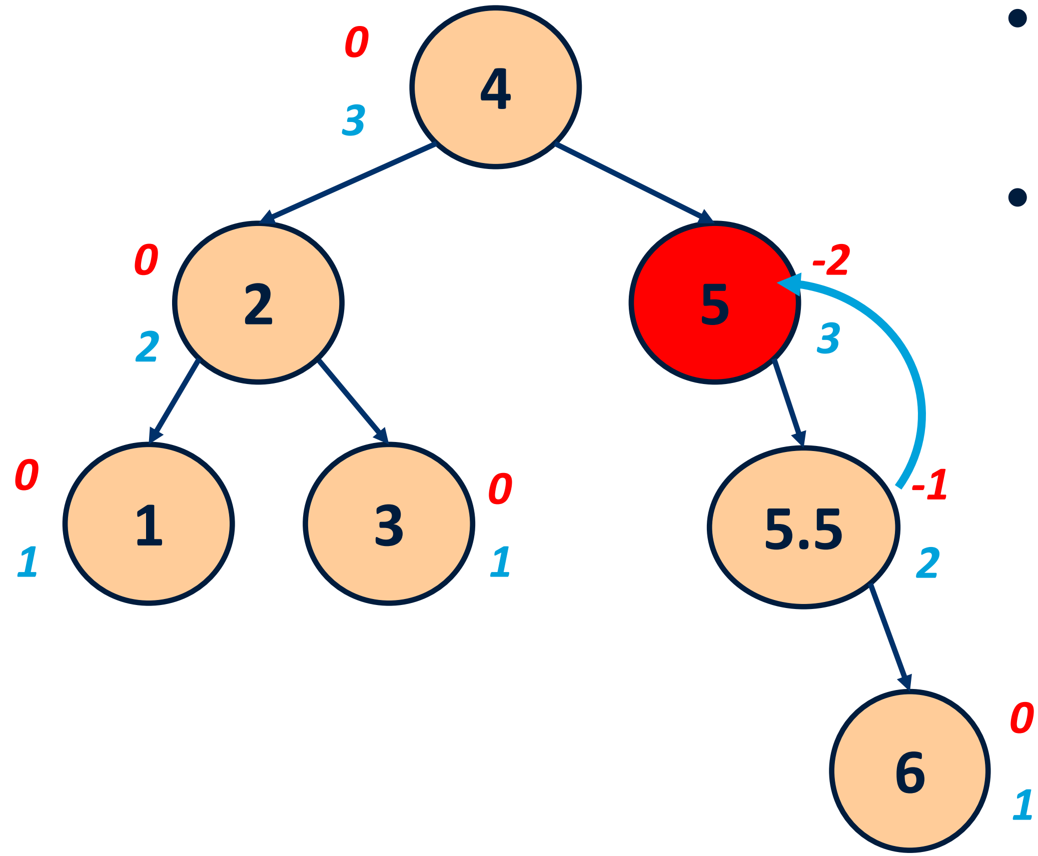
- Right rotation on 6

Insert(5.5)

AVL Tree - Example

balancing factor
height

- Now we insert 5.5



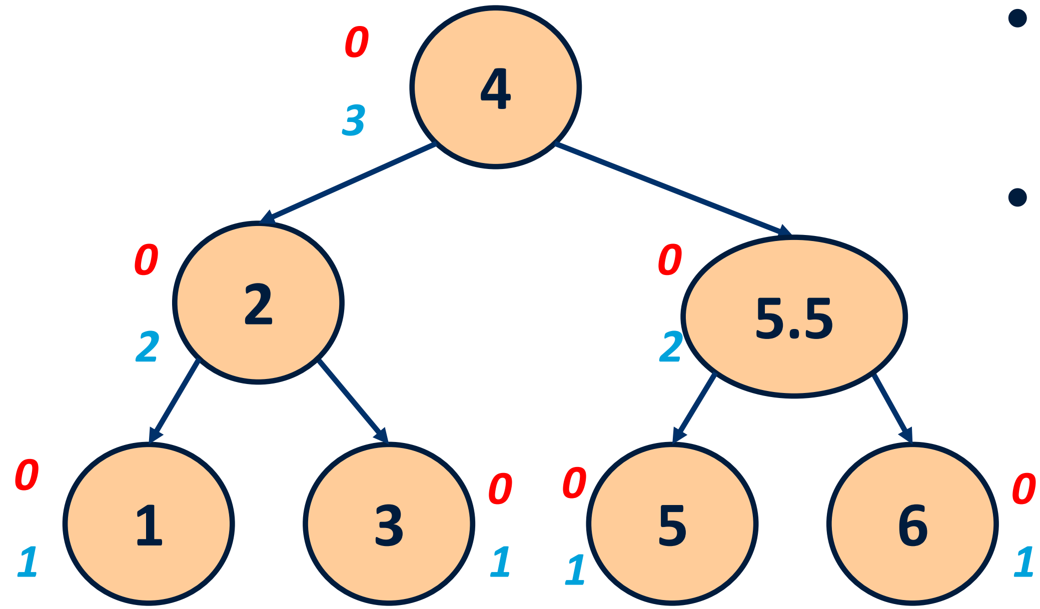
- Right rotation on 6
- Left rotation on 5

Insert(5.5)

AVL Tree - Example

balancing factor
height

- Now we insert 5.5



- Right rotation on 6
- Left rotation on 5

Insert(5.5)

AVL Tree Performance

Given an AVL tree storing *n* items:

- The data structure uses ***O(n)*** space
- A single restructuring takes ***O(1)*** time using a linked-structure binary tree
- Searching takes ***O(log n)*** time
- Insertion takes ***O(log n)*** time
- Removal takes ***O(log n)*** time

• Downside is complex implementation

Some complexities revisited

	Insert	Remove	Search
Unsorted array	O(1)	O(*n*)	O(*n*)
Sorted array	O(*n*)	O(*n*)	O(log(*n*))
Linked list	O(1)	O(*1*)	O(*n*)
BST (if balanced)	O(log n)	O(log n)	O(log n)