uc3m | Universidad Carlos III de Madrid

Grado en Ciencia e Ingeniería de Datos, 2018-2019

Unit 5. Trees

Algorithms and Data Structures (ADS)

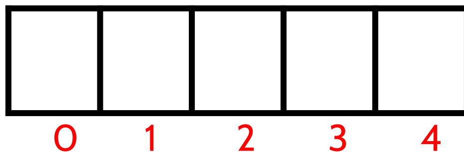
Author: Isabel Segura-Bedmar

BY NC SA

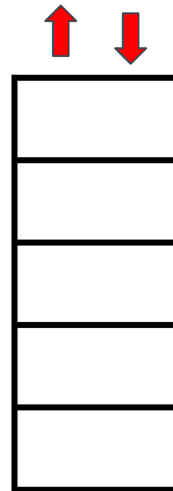
Index

- **Introduction (basic concepts)**
- ADT Binary Tree
- ADT Binary Search Tree
- Balanced trees

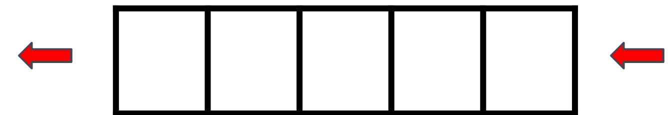
Introduction



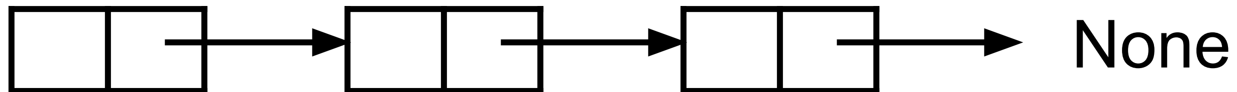
Python list (array)



Stack



Queue



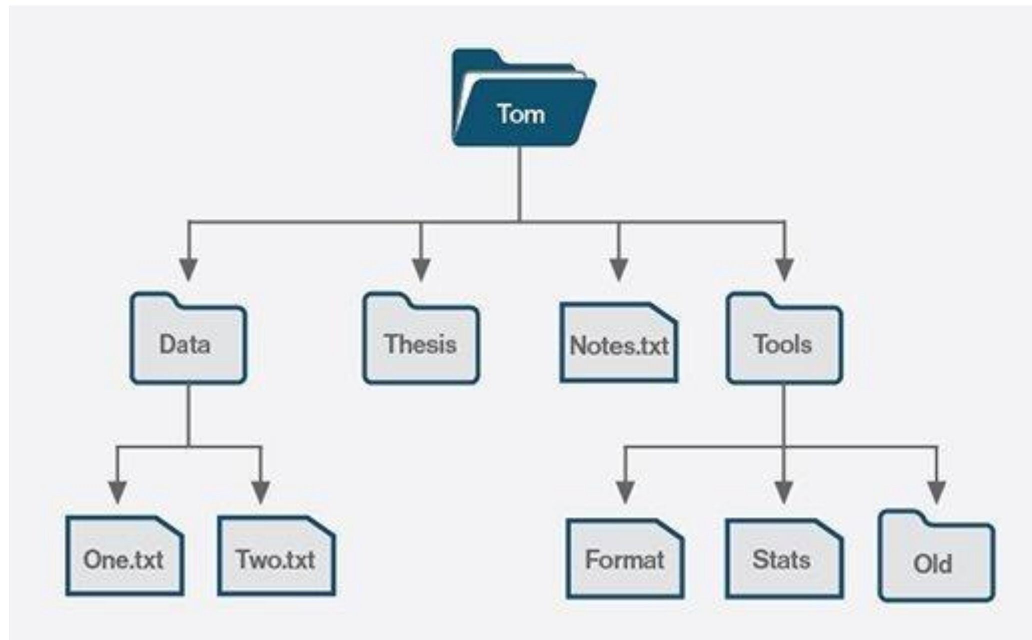
Linked List

Introduction

- How should we choose which data structure to use?
  - What kind of data we need to represent?
  - Time complexity of operations.
  - Space complexity
  - Ease of implementation

Introduction

Trees can be useful to represent hierarchical data



File system

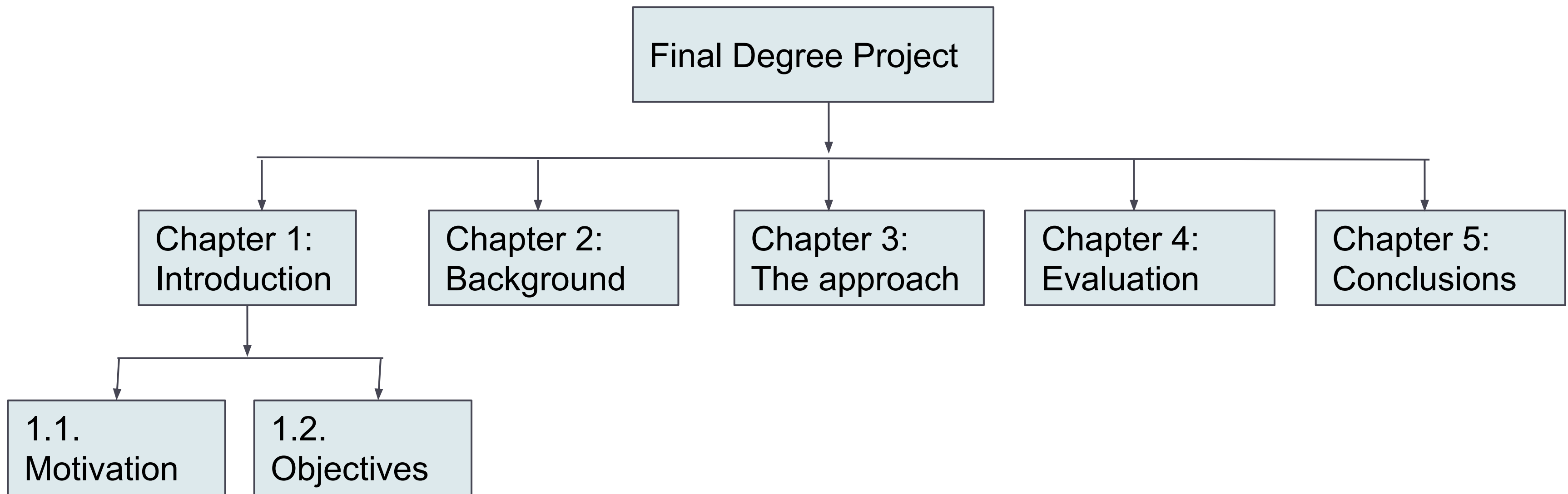
Introduction

Trees can be useful to represent hierarchical data

Family tree

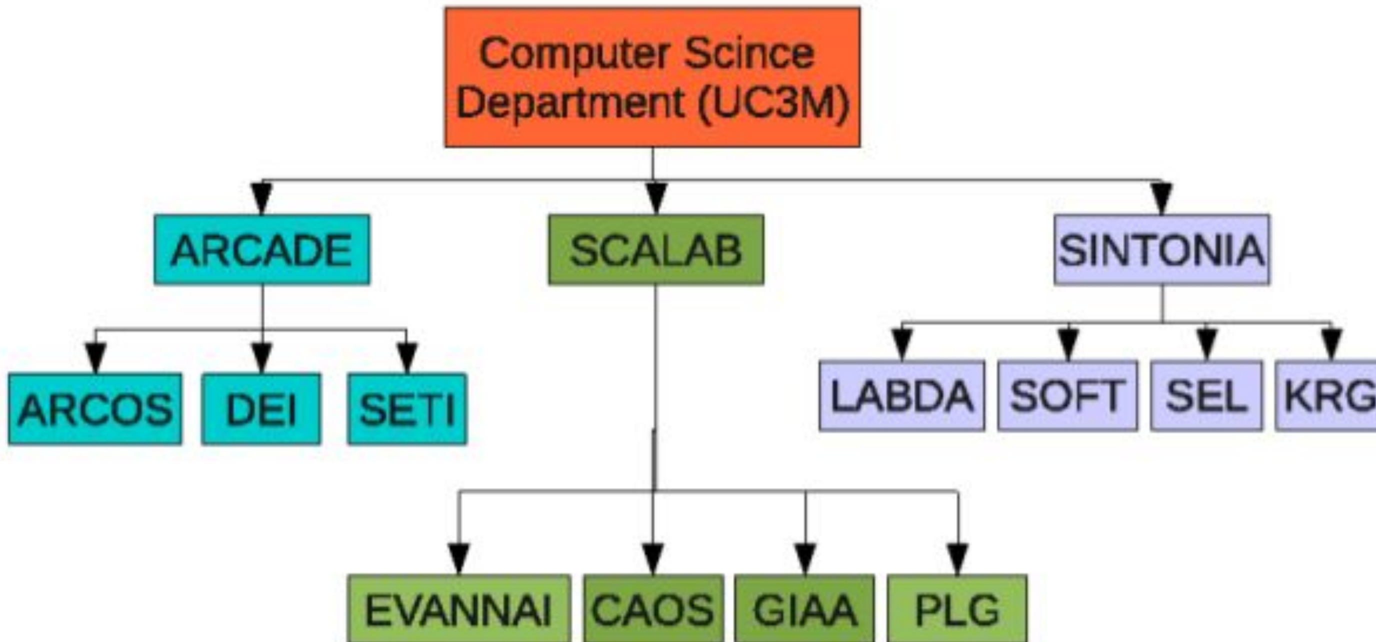
Game of Thrones Family Tree by Matt Baker - UsefulCharts.com. For educational purposes only - NOT FOR SALE.

Introduction



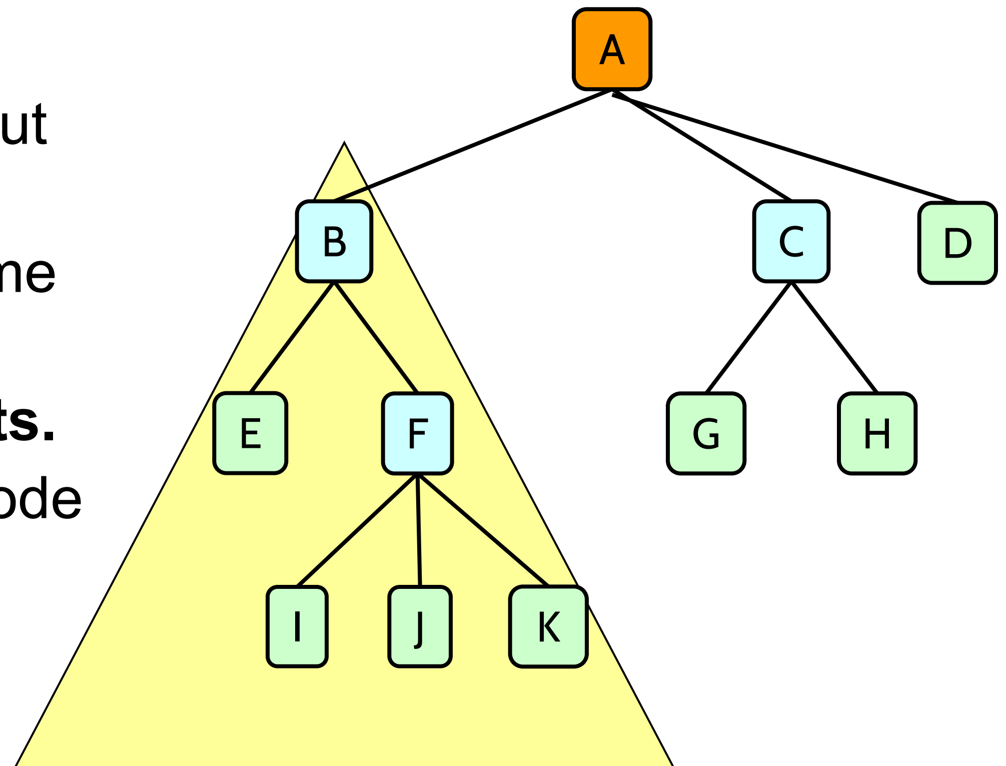
Common structure of a manuscript for a final degree project

Introduction

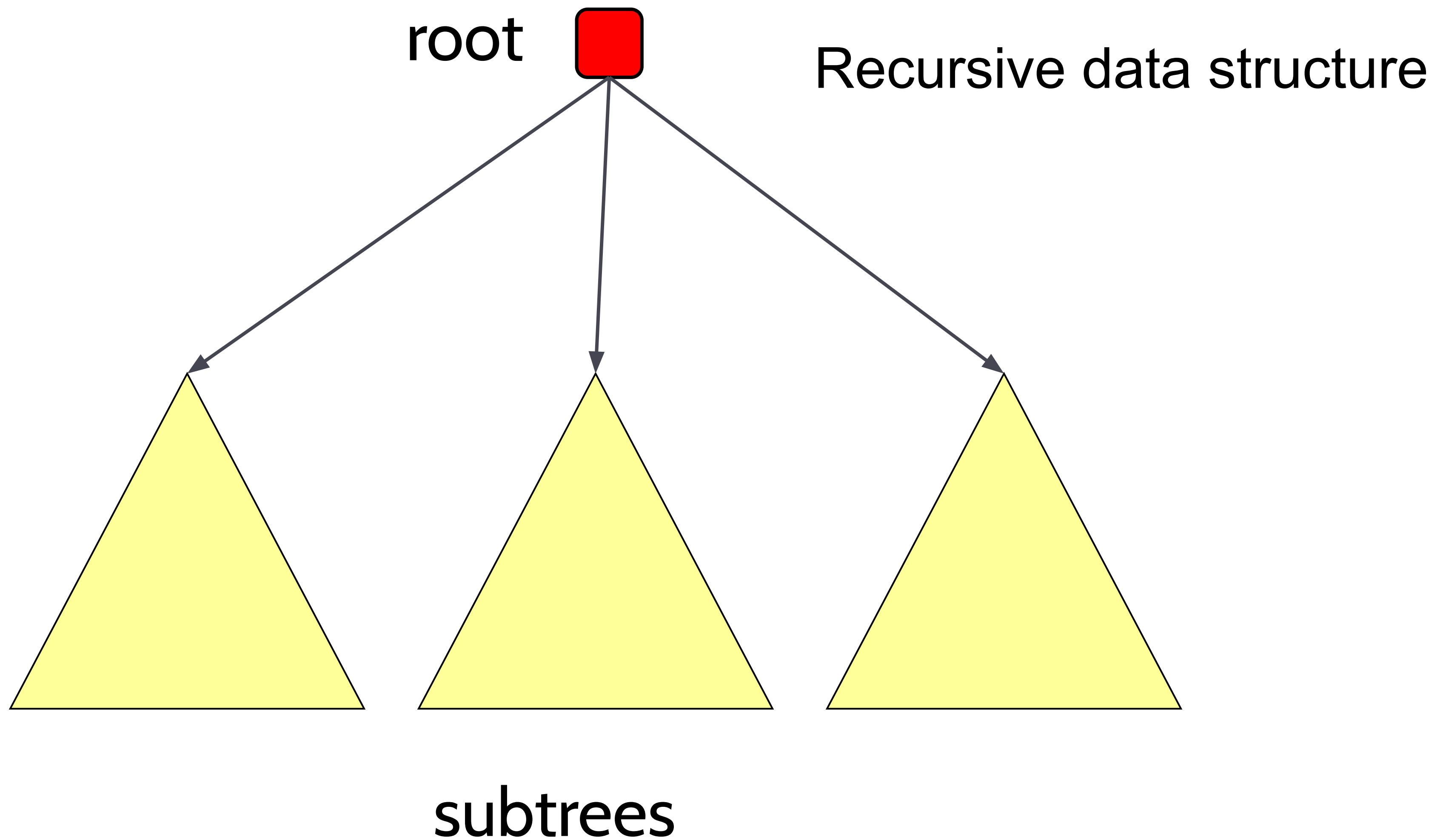


Introduction (basic concepts)

- **Root**: the only node without a father (A)
- **Internal Node**: at least one child (A, B, C, F)
- **Leaf Node (External)**: without children (E, I, J, K, G, H, D)
- **Siblings**: nodes with the same parent.
- **Ancestors and descendents.**
- **Subtree**: tree formed by a node and its descendents.



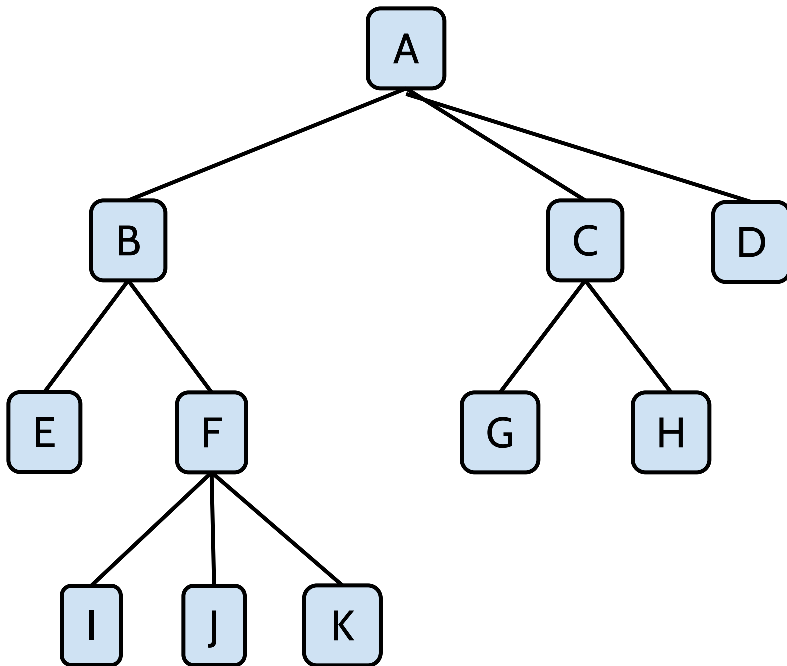
Introduction (tree data structure)



Introduction (formal tree definition)

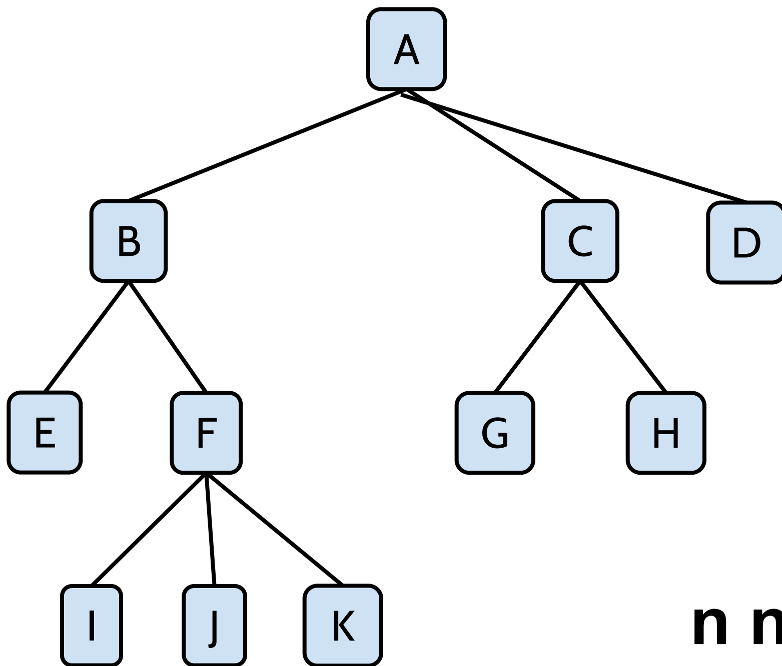
- A tree T is a set of nodes with parent-child relationships, which satisfies:
  - If T is not empty, it only has one root. The root has no parent.
  - Each node of T (no root) has a unique parent

Introduction (some tree properties)



If the tree has n nodes, how many edges does it have?

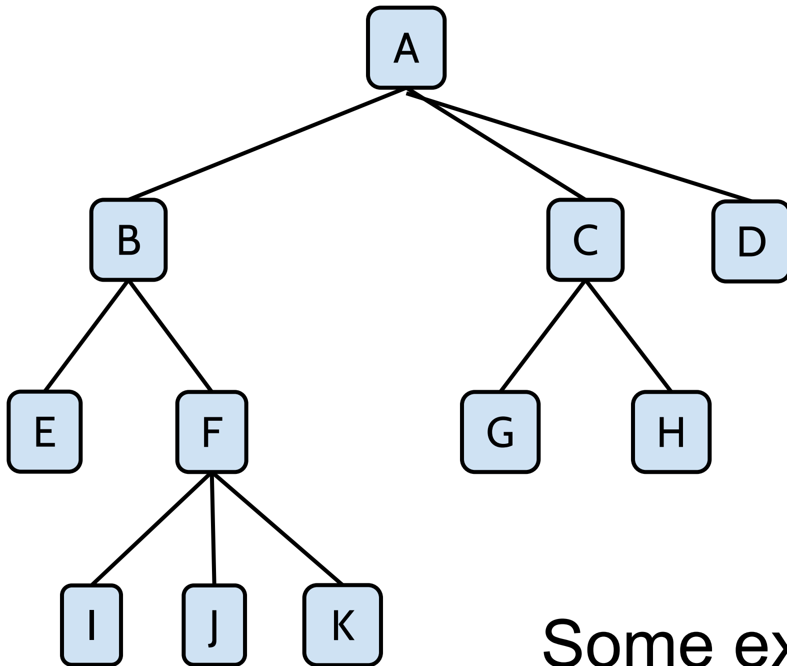
Introduction (some tree properties)



If the tree has n nodes, how many edges does it have?

n nodes => n-1 links (edges)

Introduction (some tree properties)

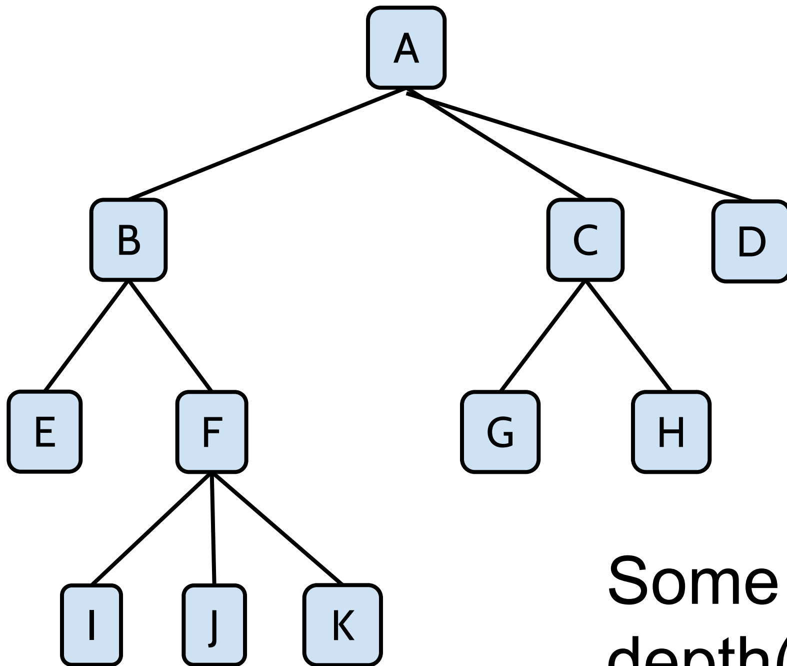


Size of x (node) is the number of nodes in its subtree.

Size of a tree is the number of its nodes

Some examples:
size(B)=6
size(C)=3
size(A)=11

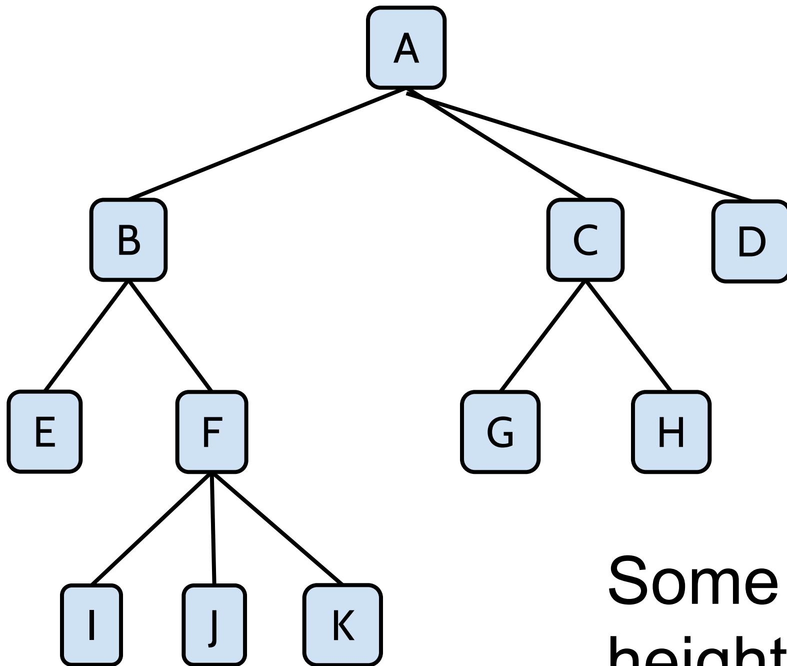
Introduction (some tree properties)



Depth of x (node) is the length of the path from the root to node x.

Some examples:
depth(A)=0
depth(B)=1
depth(E)=2
depth(J)=3

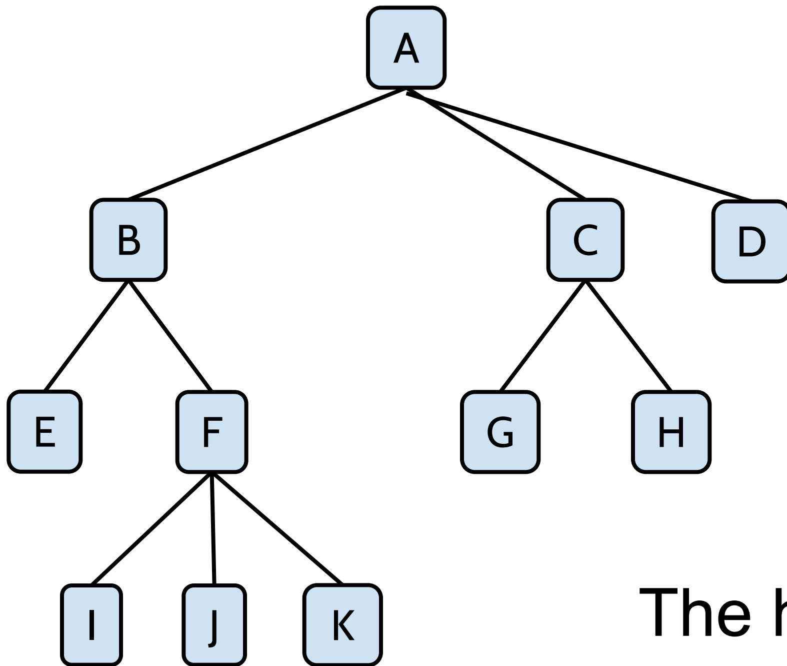
Introduction (some tree properties)



Height of x (node) is the length of the longest path from the node x to any leaf.

Some examples:
height(K)=0
height(F)=1
height(B)=2
height(A)=3

Introduction (some tree properties)



Height of a tree = height of its root

The height of this tree is: height(A)=3

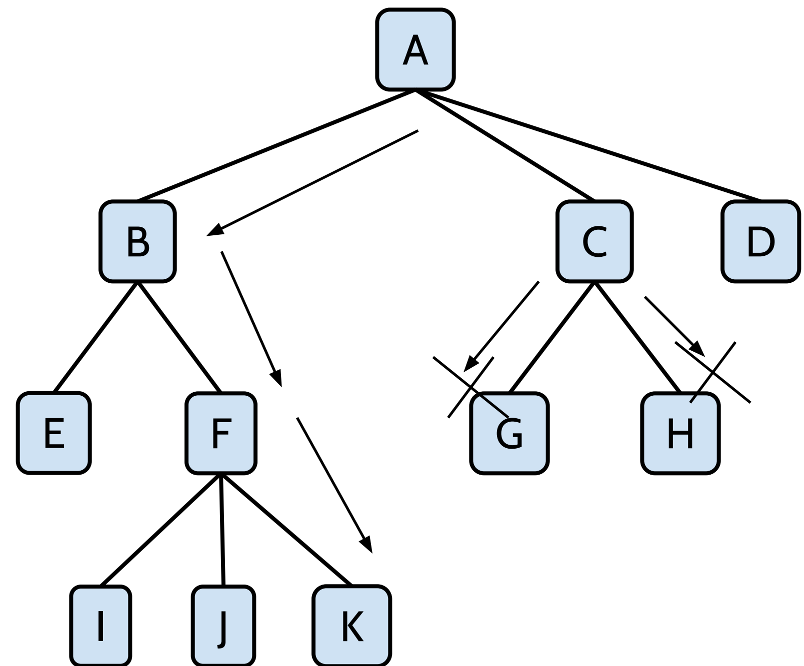
Note: The height of an empty tree is -1.

Introduction (some tree properties)

- **path:** there is a **path between nodes X and Y** if there is a sequence of nodes allowing to reach Y from X (going only through descendants).

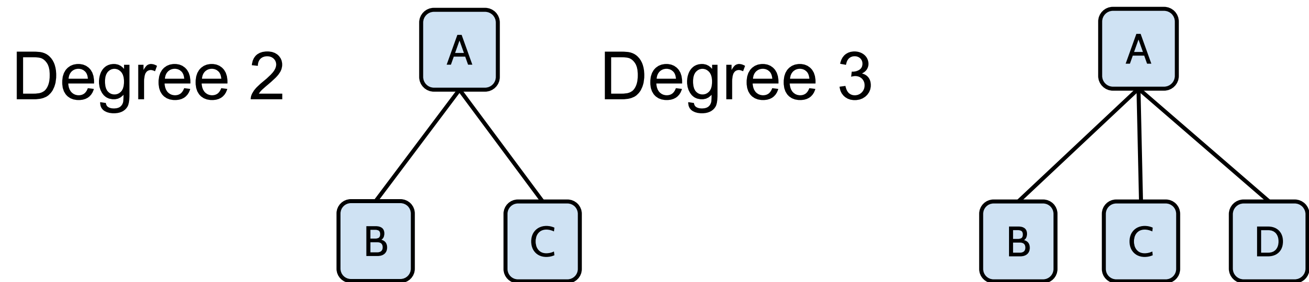
path(A,K)={A,B,F,K}

path(C,K)={}



Introduction (some tree properties)

- **Degree of a node**: number of its children



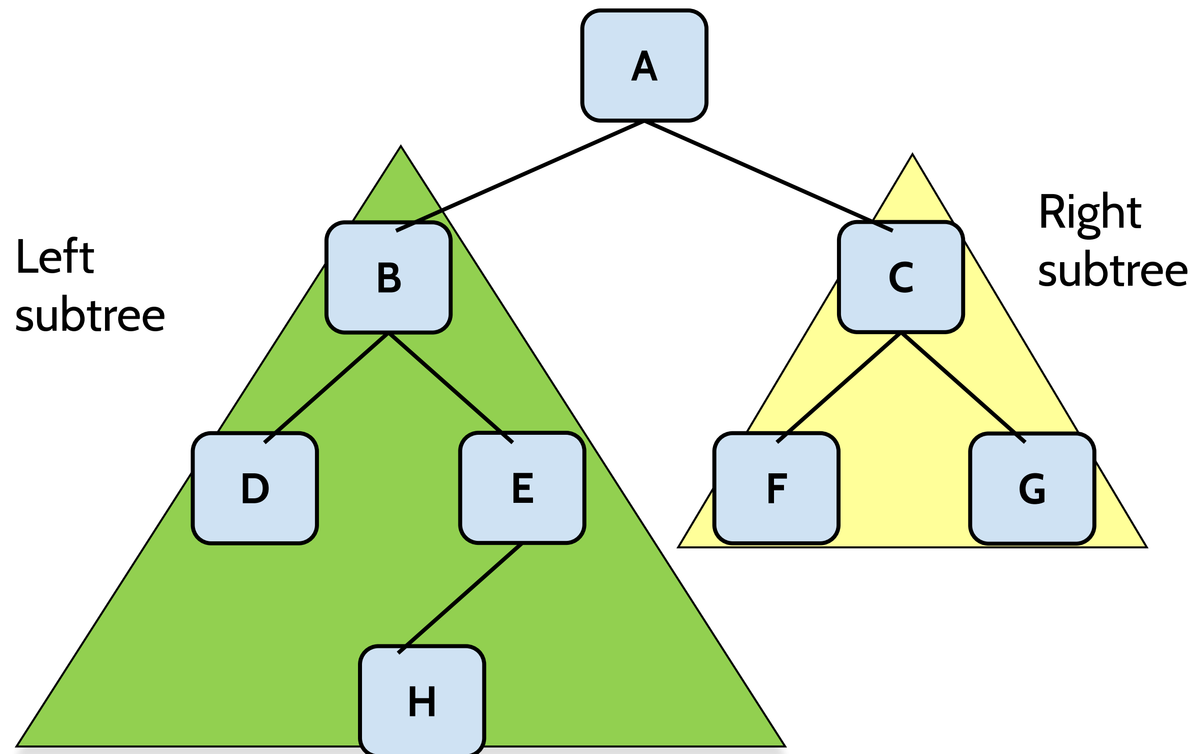
- **Degree of a tree**: the greatest degree for all its nodes

Index

- Introduction (basic concepts)
- **Binary Tree ADT**
- Binary Search Tree ADT
- Balanced trees

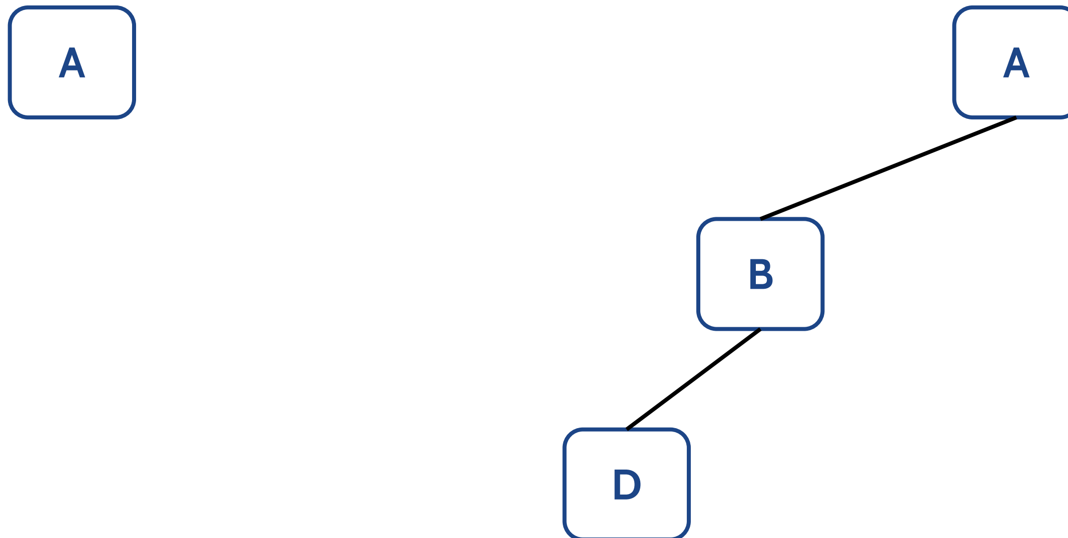
Binary tree ADT

- **Binary tree**: tree with degree 2.
  - each node has at **most** two children



Binary tree ADT

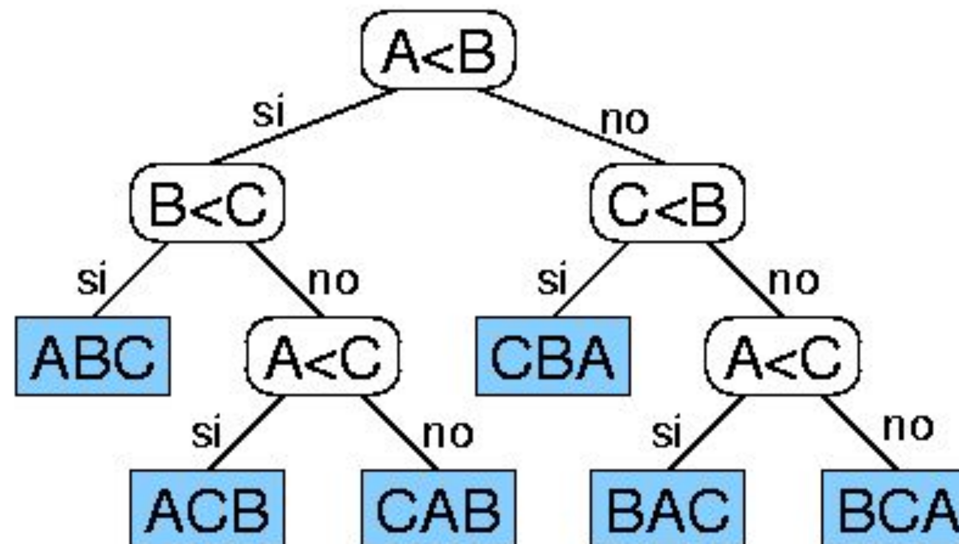
They are also binary trees



Binary tree ADT (applications)

Example I: Decision trees

- Intern node: questions with yes/no answers
- Leaf nodes: decisions



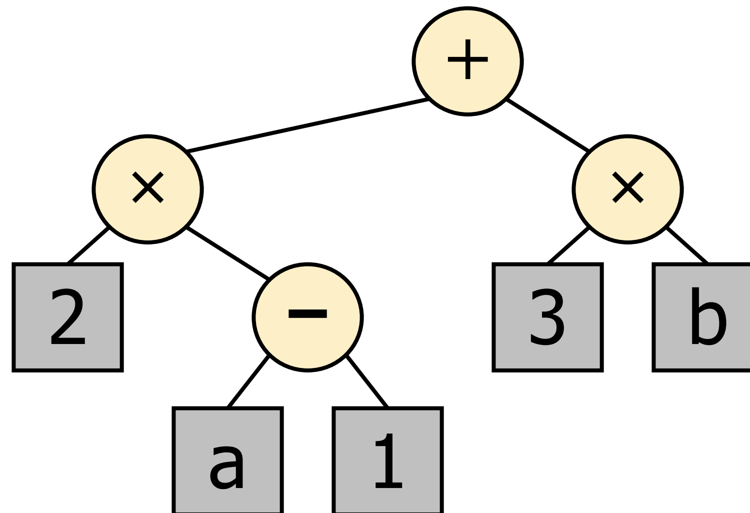
Example of a decision tree to order three elements A, B, C

Binary tree ADT (applications)

Example II: representing arithmetic expressions

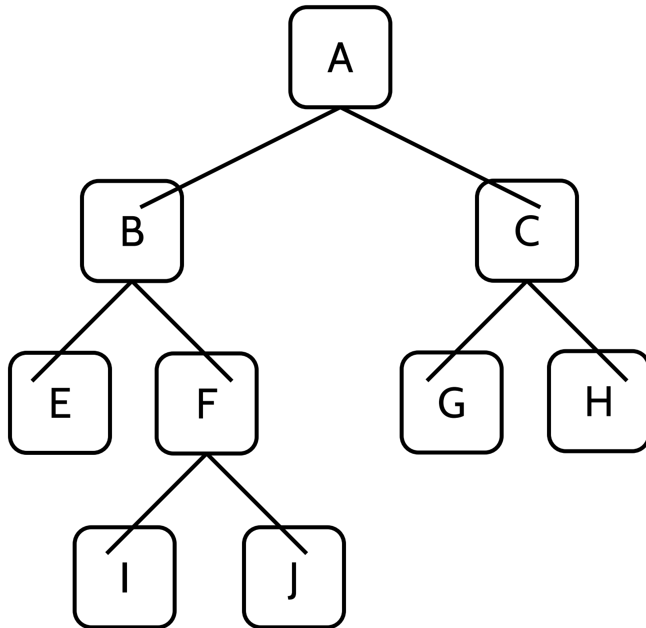
- Internal nodes: operators
- Leaf nodes: operands

2x(a-1)+3xb

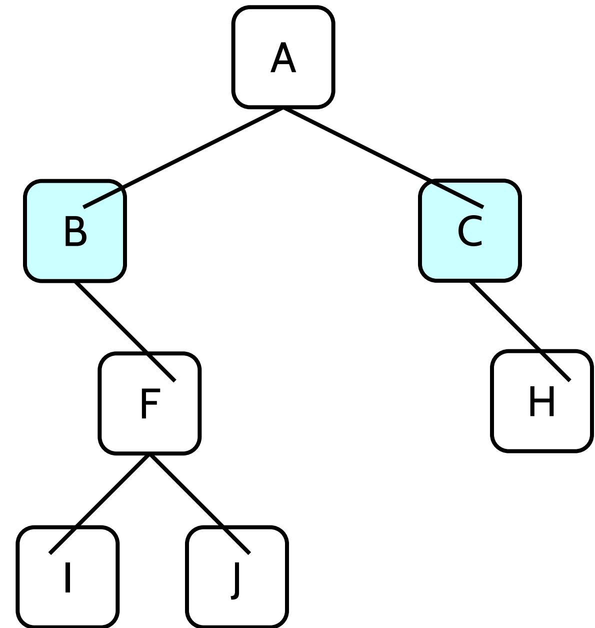


Binary tree ADT

- A binary tree is a **strict (proper) binary** tree if every node has 0 or two children.



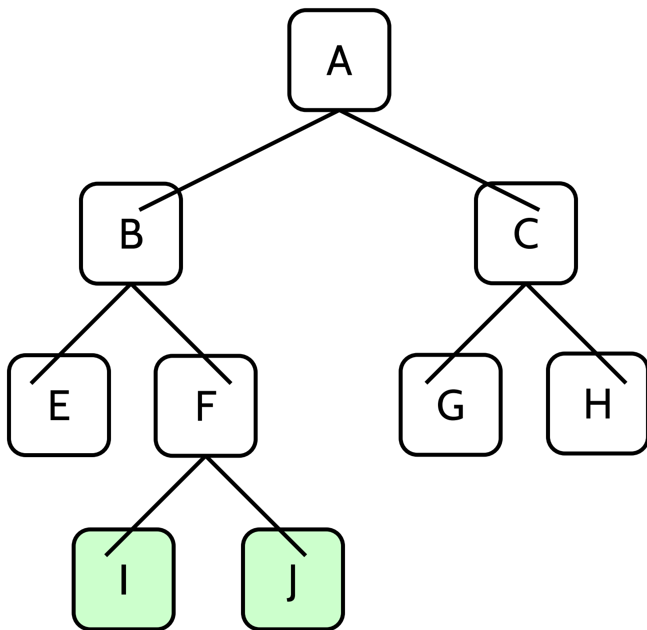
It is a strict binary tree



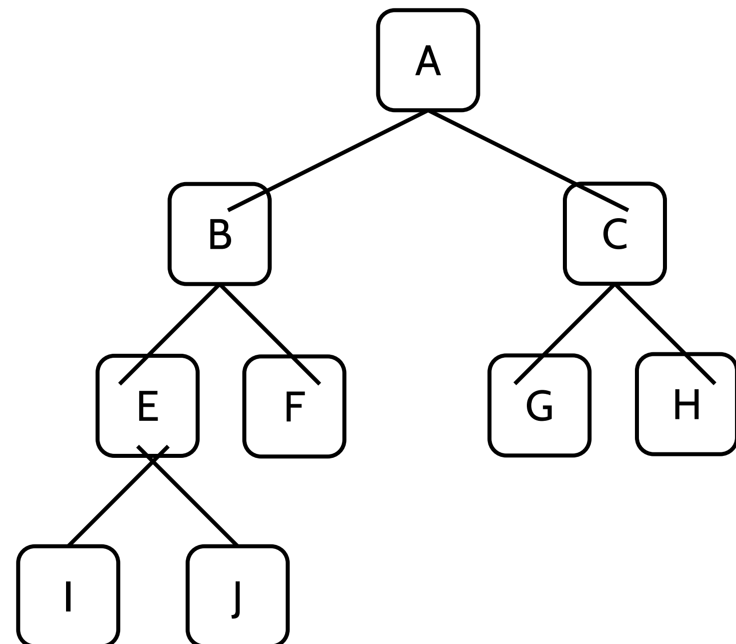
It is not a strict binary tree

Binary tree ADT

A **complete binary tree** is a **binary tree** in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible.



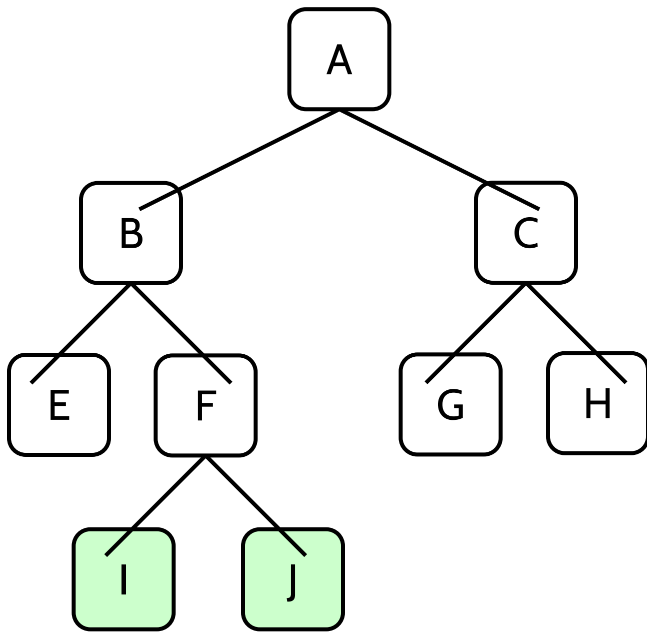
It is not a complete binary tree



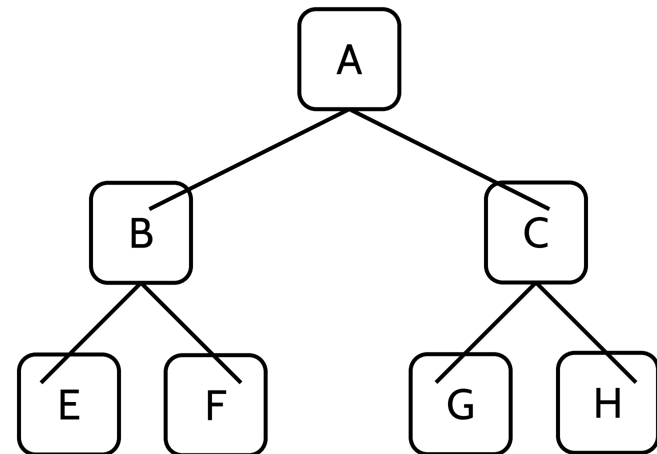
It is a complete binary tree

Binary tree ADT

In a **perfect binary tree,** all the levels are filled. This means that all the leaves are at the same level



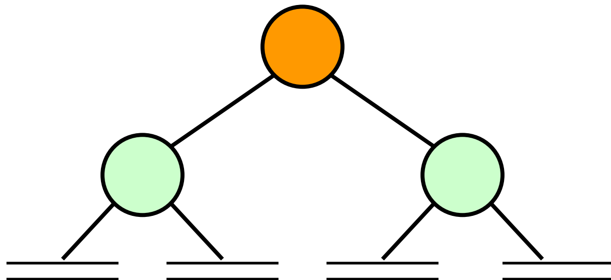
It is not a perfect binary tree



It is a perfect binary tree

Binary tree ADT (properties)

- Notation
  - n: number of nodes
  - e: number of leafs
  - i: number of internal nodes
  - h: height of a tree



- Properties
  - $n = 2e-1$
  - $h \le i$
  - $h \le (n-1)/2$
  - $e \le 2^h$
  - $h \ge \log_2 e$
  - $h \ge \log_2(n+1)-1$

- If it is a perfect binary tree:
  - $e = i+1$
  - $e \ge h+1$

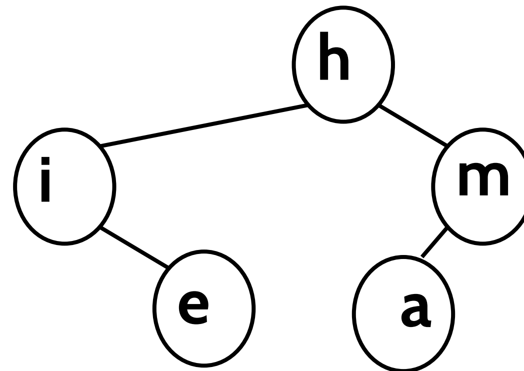
Index

- Introduction (basic concepts)
- **Binary Tree ADT**
  - **Binary Tree Traversals**
- Binary Search Tree ADT
- Balanced trees

Binary tree ADT: Preorder traversal

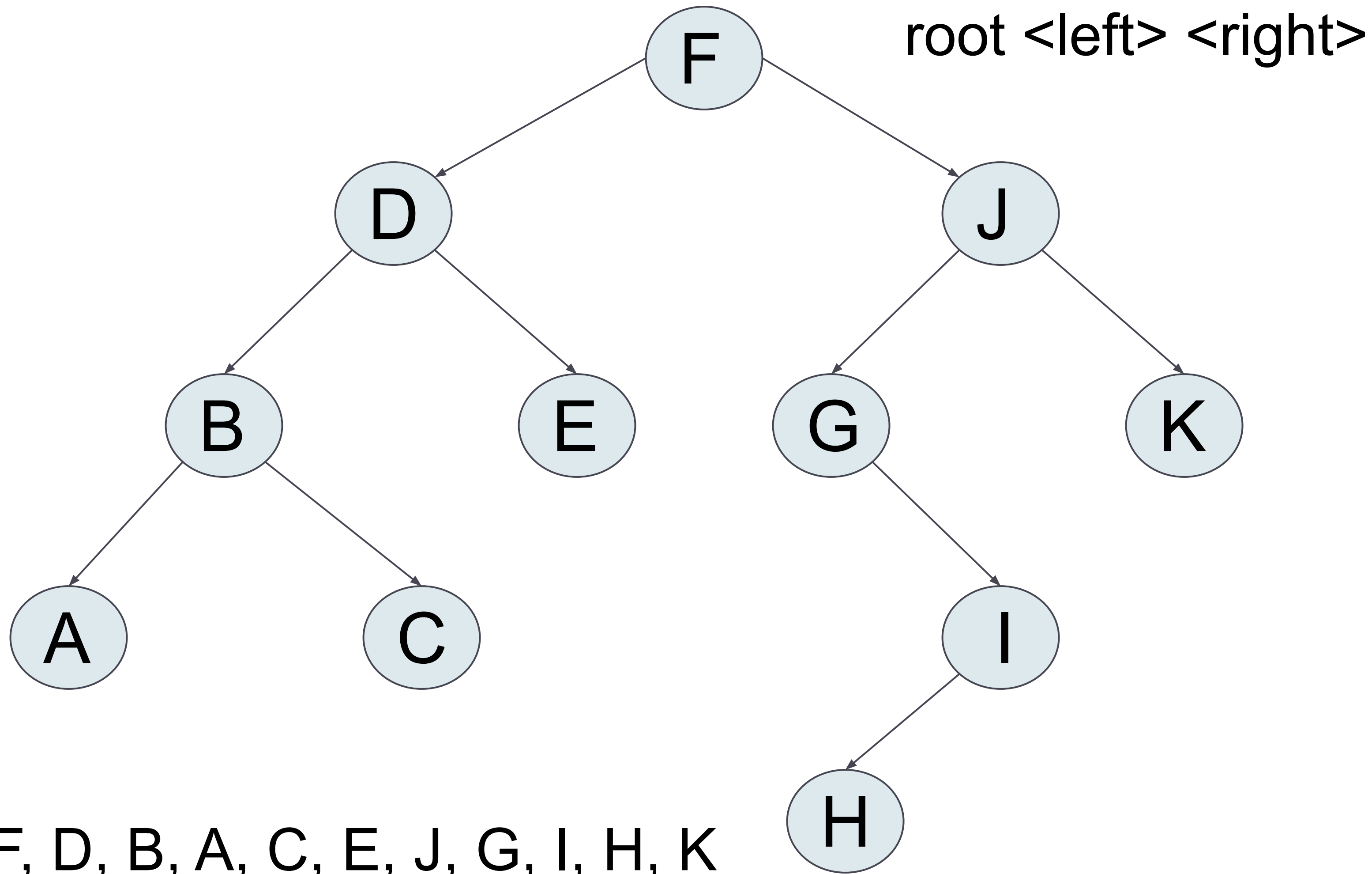
Preorder traversal

- First, root is visited, then the left subtree is visited and, finally, the right subtree (root, left, right)
- Example:



pre-order: (h, i, e, m, a)

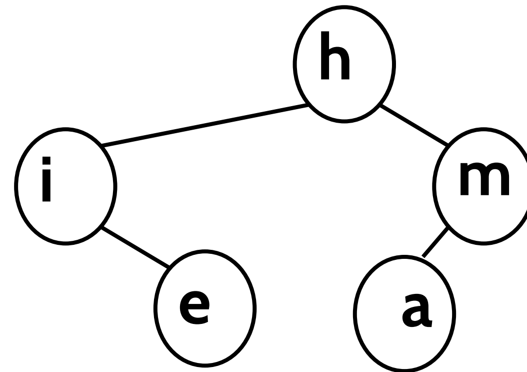
Binary tree ADT: Preorder traversal



Binary tree ADT: Postorder traversal

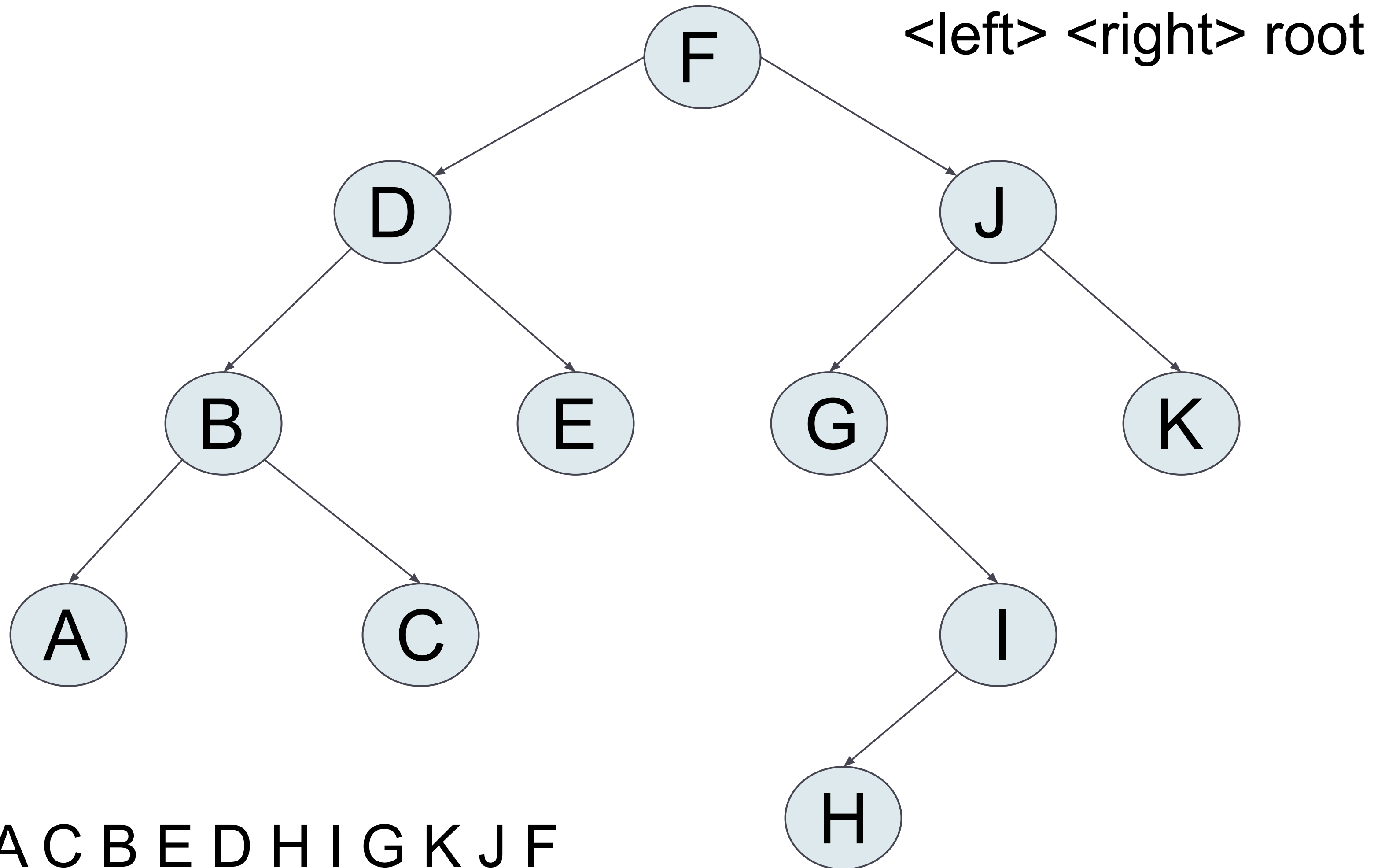
Postorder traversal

- First, we visit the left subtree, then the right subtree, and finally, the root. (left, right, root)
- Example:



post-orden: (e, i, a, m, h)

Binary tree ADT: Postorder traversal



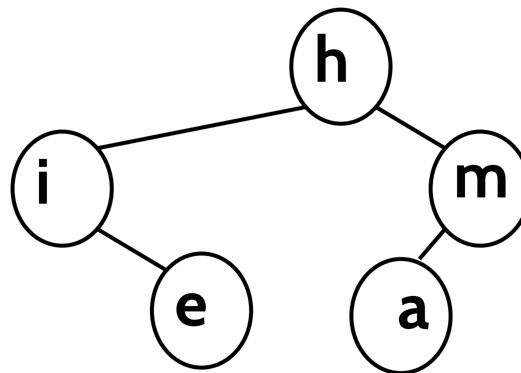
<left> <right> root

A C B E D H I G K J F

Binary tree ADT: In-order traversal

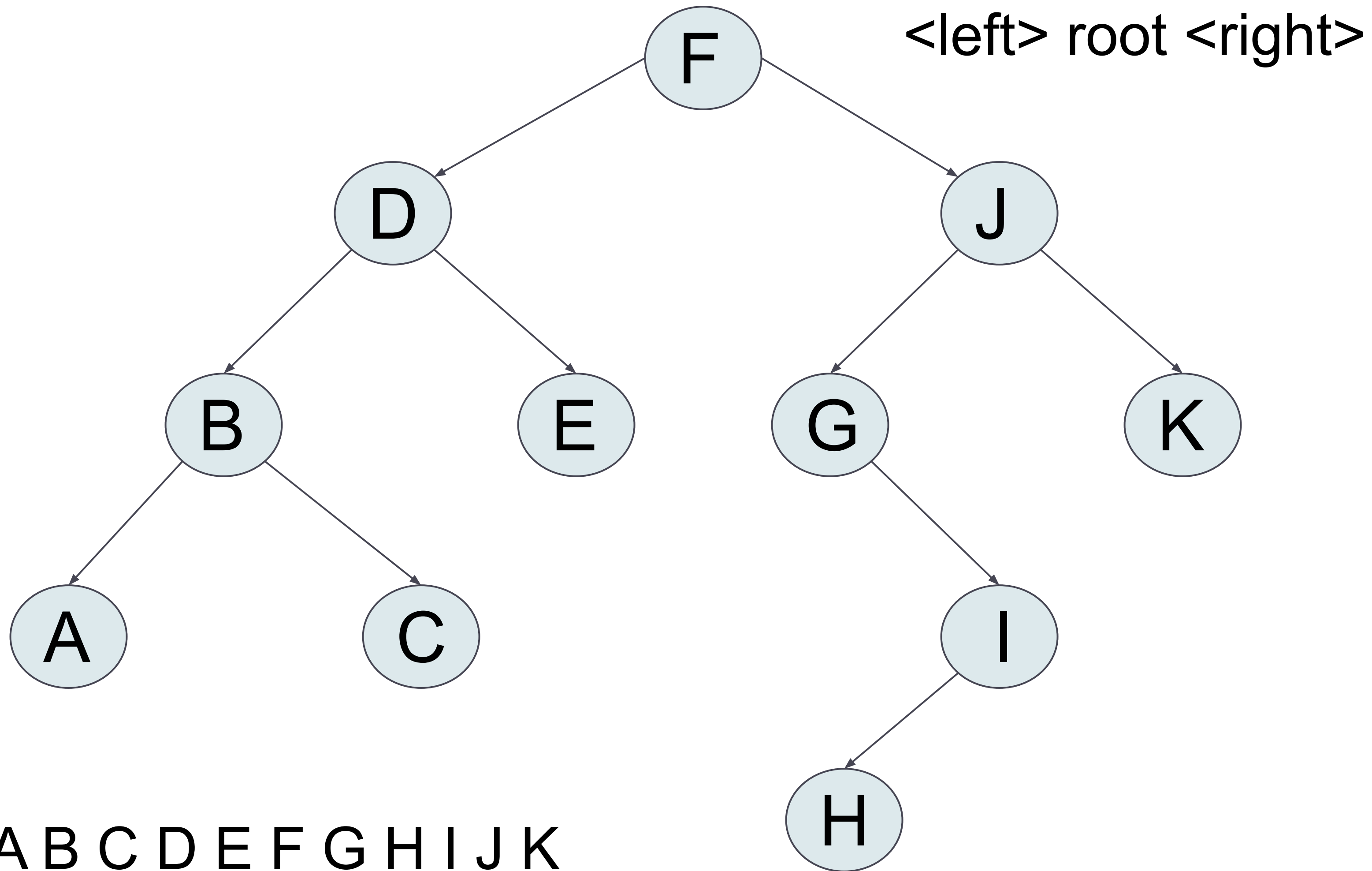
In-order traversal we visit the left subtree, the root and the right subtree (left, root, right)

- Example:



in-order: (i, e, h, a, m)

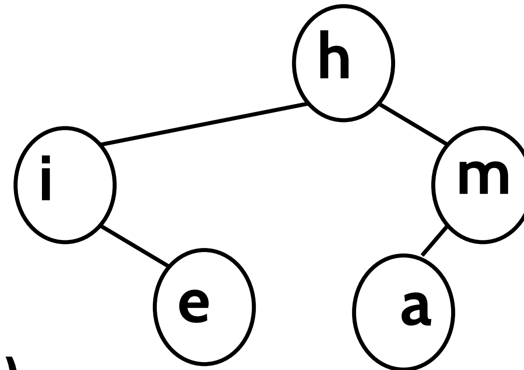
Binary tree ADT: In-order traversal



Binary tree ADT: Level-order traversal

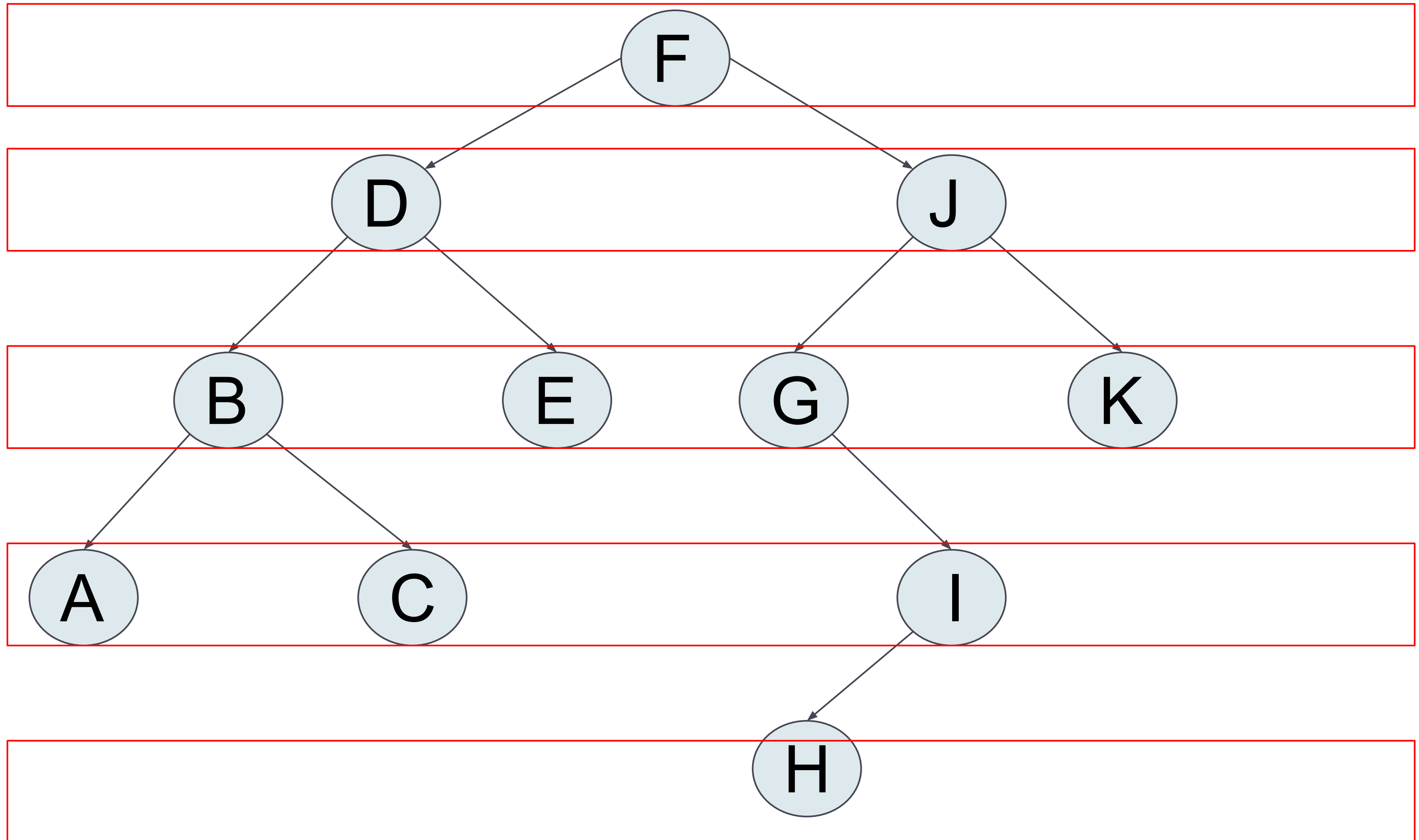
Level-order traversal

- Node are visited by level according to depth in the tree. So, nodes ate the same level are visited, in descending order and from left to right.
- Example:



Level-order: (h,i,m,e,a)

Binary tree ADT: Level-order traversal



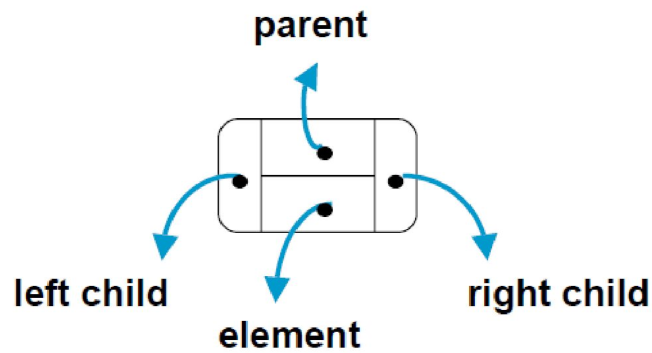
Level-order:F D J B E G K A C I H

Index

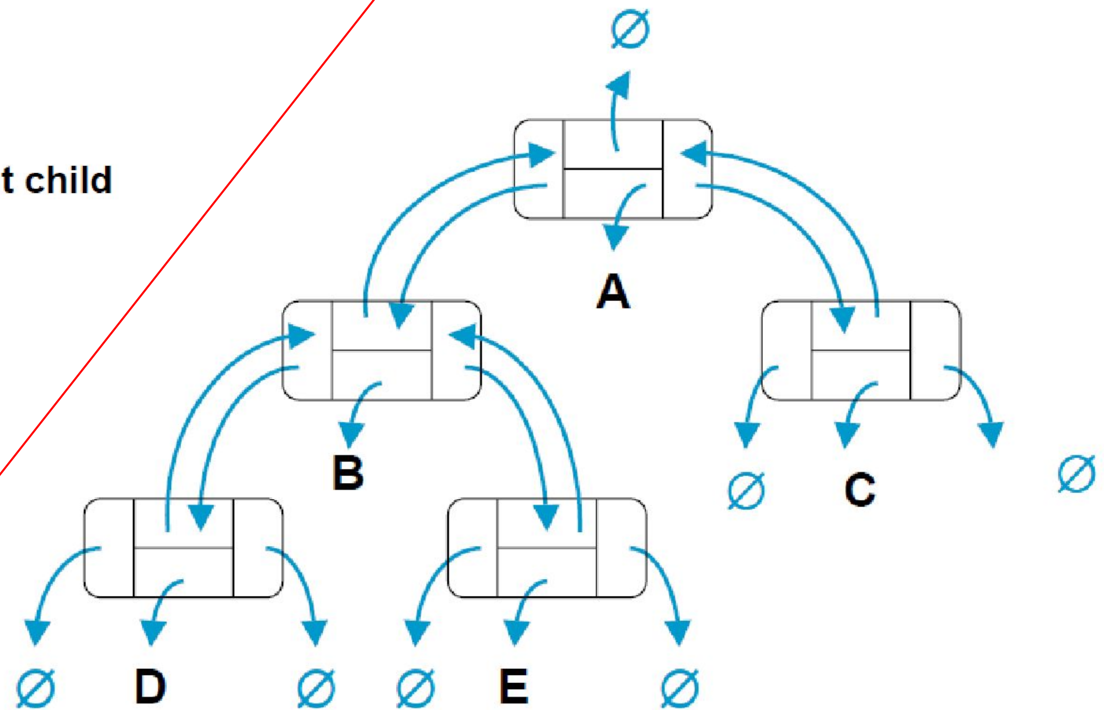
- Introduction (basic concepts)
- **Binary Tree ADT**
  - Binary Tree Traversals
  - **Implementation**
- Binary Search Tree ADT
- Balanced trees

Binary Tree ADT: implementation

Node

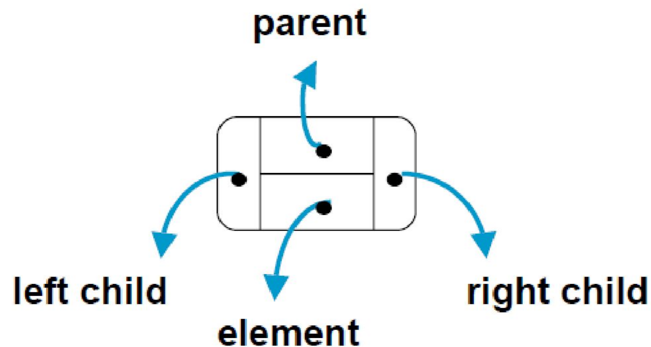


Tree



Binary Tree ADT: implementation

Node




```
Algorithm Node(node,elem):
    node.elem=elem
    node.leftChild=None
    node.rightChild=None
    node.parent=None
```

Binary Tree ADT: implementation

By default, the tree constructor must create an empty tree (root=None):

```
Algorithm Tree(T):
  T.root=None
```

Binary Tree ADT: implementation (size method)

- The size of a tree is the number of nodes. It can be defined as the size of the root node. The size of a node is its descendants including itself.

```
Algorithm size(T):
   return sizeNode(T.root)

Algorithm sizeNode(node):
   ....
```

Binary Tree ADT: implementation (size method)

```
Algorithm size(T):
   return sizeNode(T.root)


Algorithm sizeNode(node):
   If node is None:
      return 0

   return 1 + sizeNode(node.leftChild) +
sizeNonde(node.rightChild)
```

Binary Tree ADT: implementation (height method)

- The height of a tree is the height of its root node. The height of a node is the number of edges on the longest path from the node to a leaf.

```
Algorithm height(T):
   return heightNode(T.root)

Algorithm heightNode(node):
   ....
```

Binary Tree ADT: implementation (height method)

```
Algorithm height(T):
   return heightNode(T.root)

Algorithm heightNode(node):
   If node is None:
      return -1

   return 1 +max(heightNode(node.leftChild),
heightNode(node.rightChild))
```

Binary Tree ADT: implementation (depth method)

- The depth of a node is the length of the path from the root of the tree to the node:

```
Algorithm depth(T,node):
   If node==T.root:
      return 0

   return 1 + depth(T,node.parent)
```

Binary Tree ADT: implementation (preorder method)

```
Algorithm preorder(T):
   preorderNode(T.root)

Algorithm preorderNode(node):
      ....
```

Binary Tree ADT: implementation (preorder method)

```
Algorithm preorder(T):
   preorderNode(T.root)

Algorithm preorderNode(node):
   If node is not None:
      print(node.element)
      preorderNode(node.leftChild)
      preorderNode(node.rightChild)
```

Binary Tree ADT: implementation (postorder method)

```
Algorithm postorder(T):
   postorderNode(T.root)

Algorithm postorderNode(node):
       ....
```

Binary Tree ADT: implementation (postorder method)

```
Algorithm postorder(T):
  postorderNode(T.root)

Algorithm postorderNode(node):
  If node is not None:
    postorderNode(node.leftChild)
    postorderNode(node.rightChild)
    print(node.element)
```

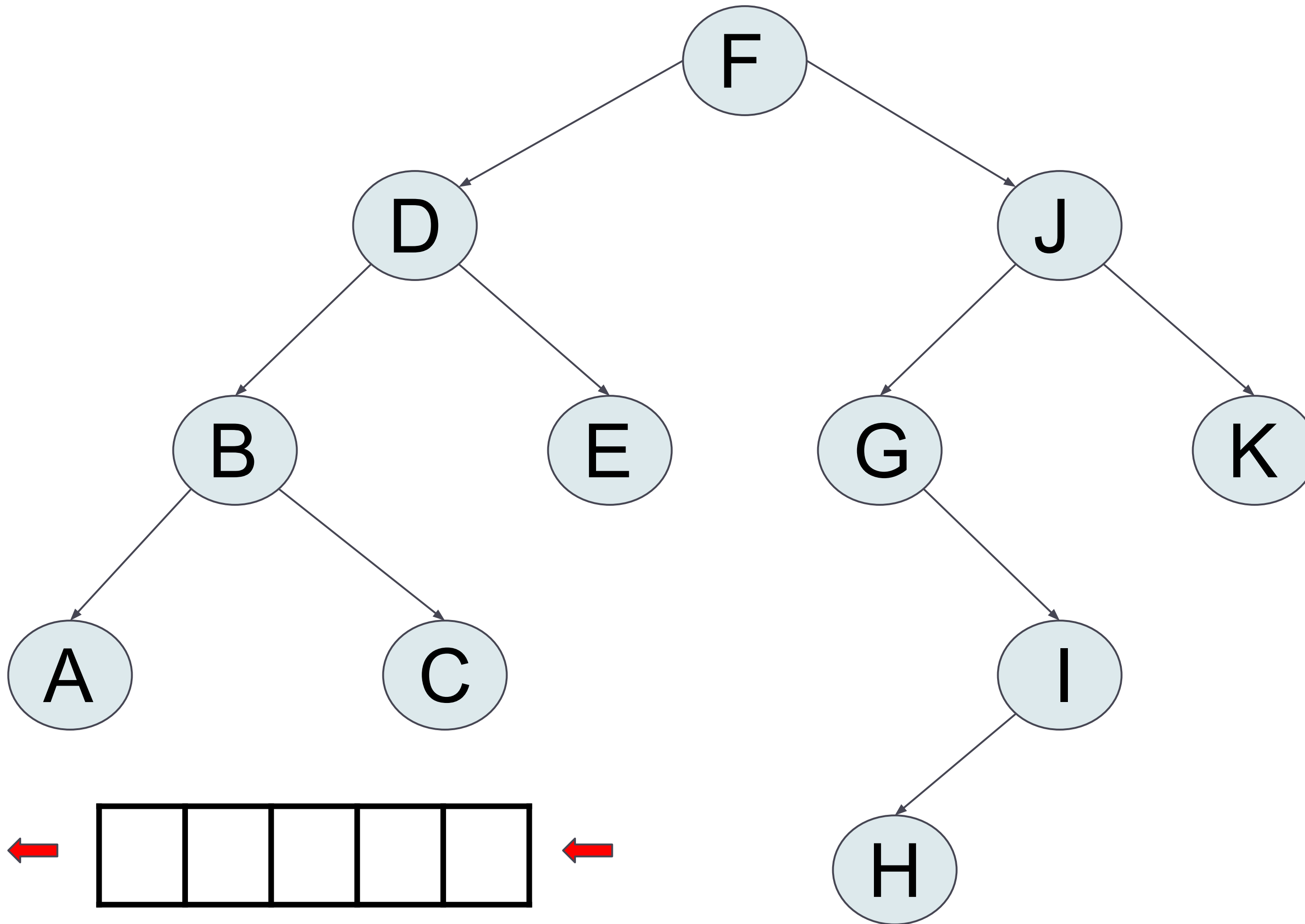
Binary Tree ADT: implementation (inorder method)

```
Algorithm inorder(T):
   inorderNode(T.root)

Algorithm inorderNode(node):
      ....
```

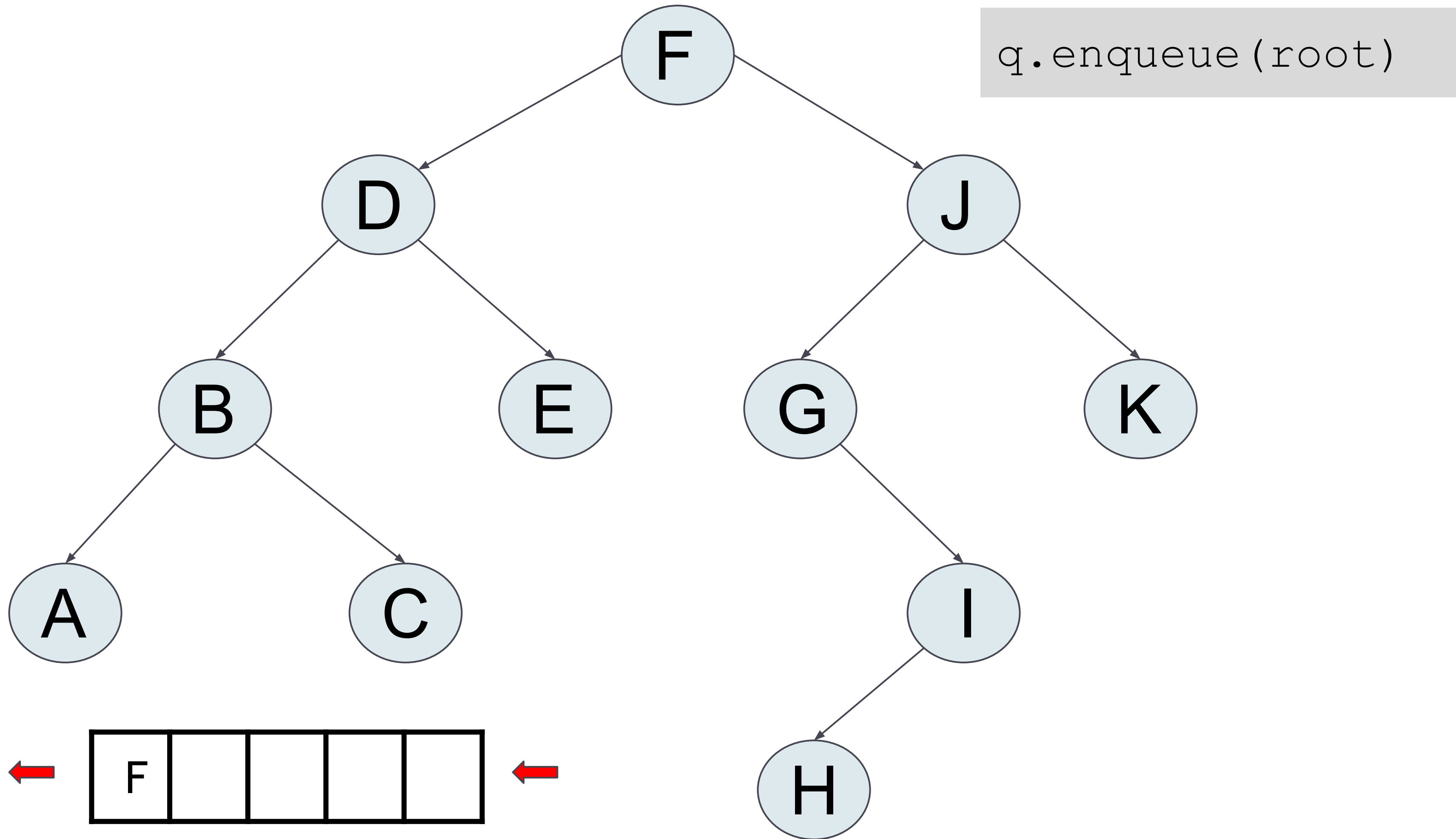
Binary Tree ADT: implementation (inorder method)

```
Algorithm inorder(T):
   inorderNode(T.root)

Algorithm inorderNode(node):
   If node is not None:
      inorderNode(node.leftChild)
      print(node.element)
      inorderNode(node.rightChild)
```

Binary Tree ADT: implementation (level-order method)

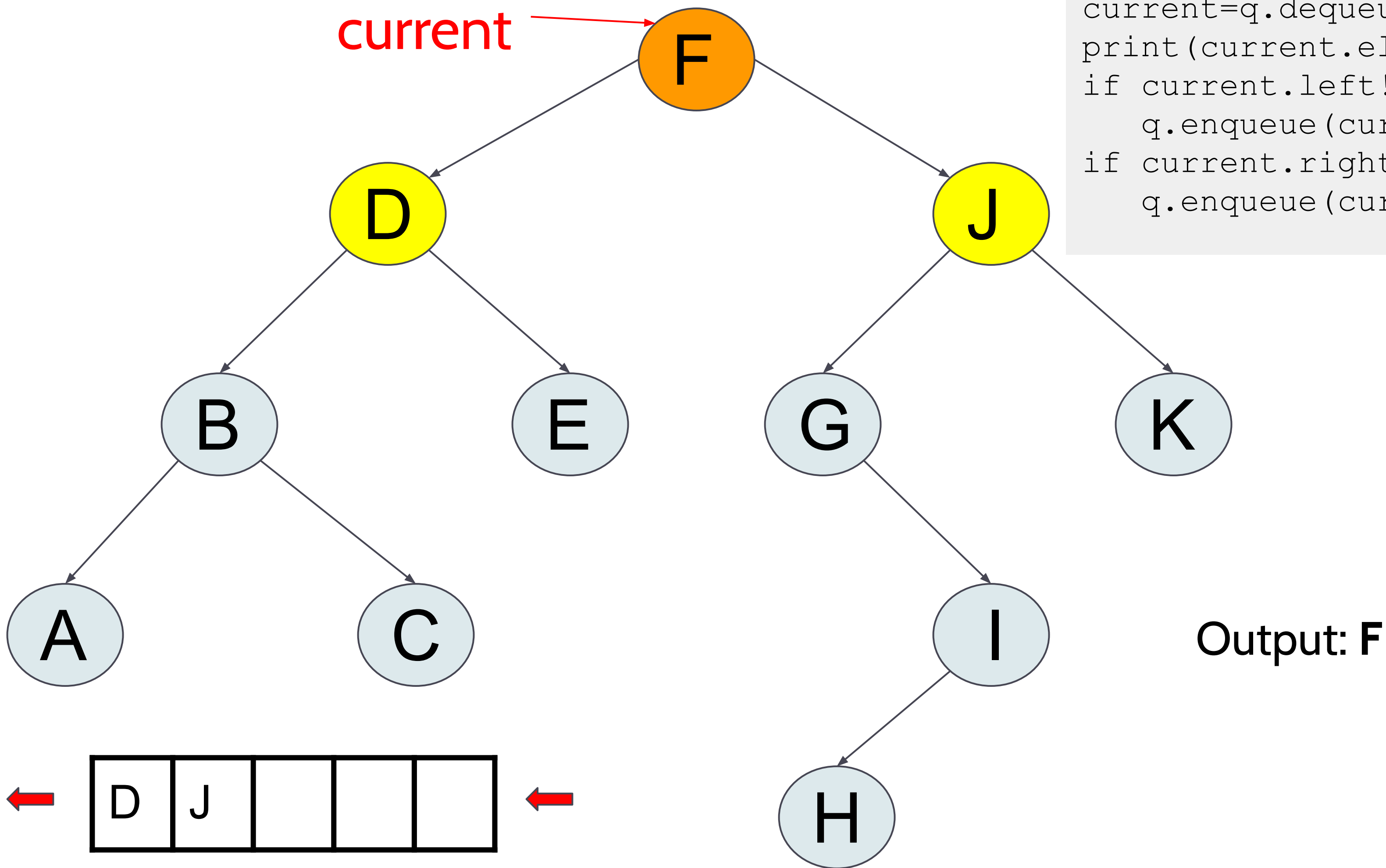


Binary Tree ADT: implementation (level-order method)



Binary Tree ADT: implementation (level-order method)

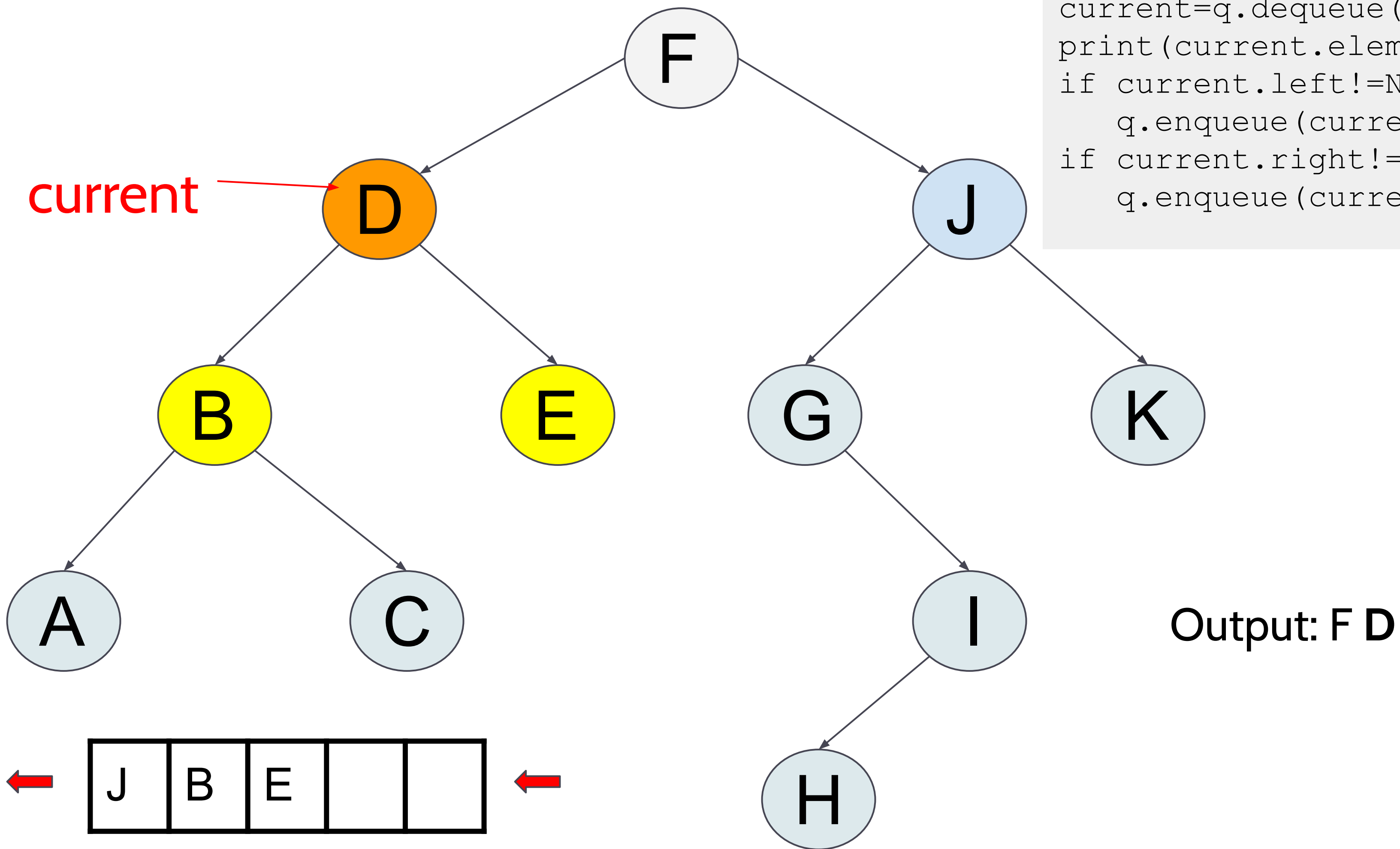
```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```



q: Queue

Binary Tree ADT: implementation (level-order method)

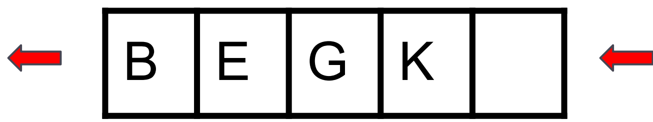
```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```



Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```

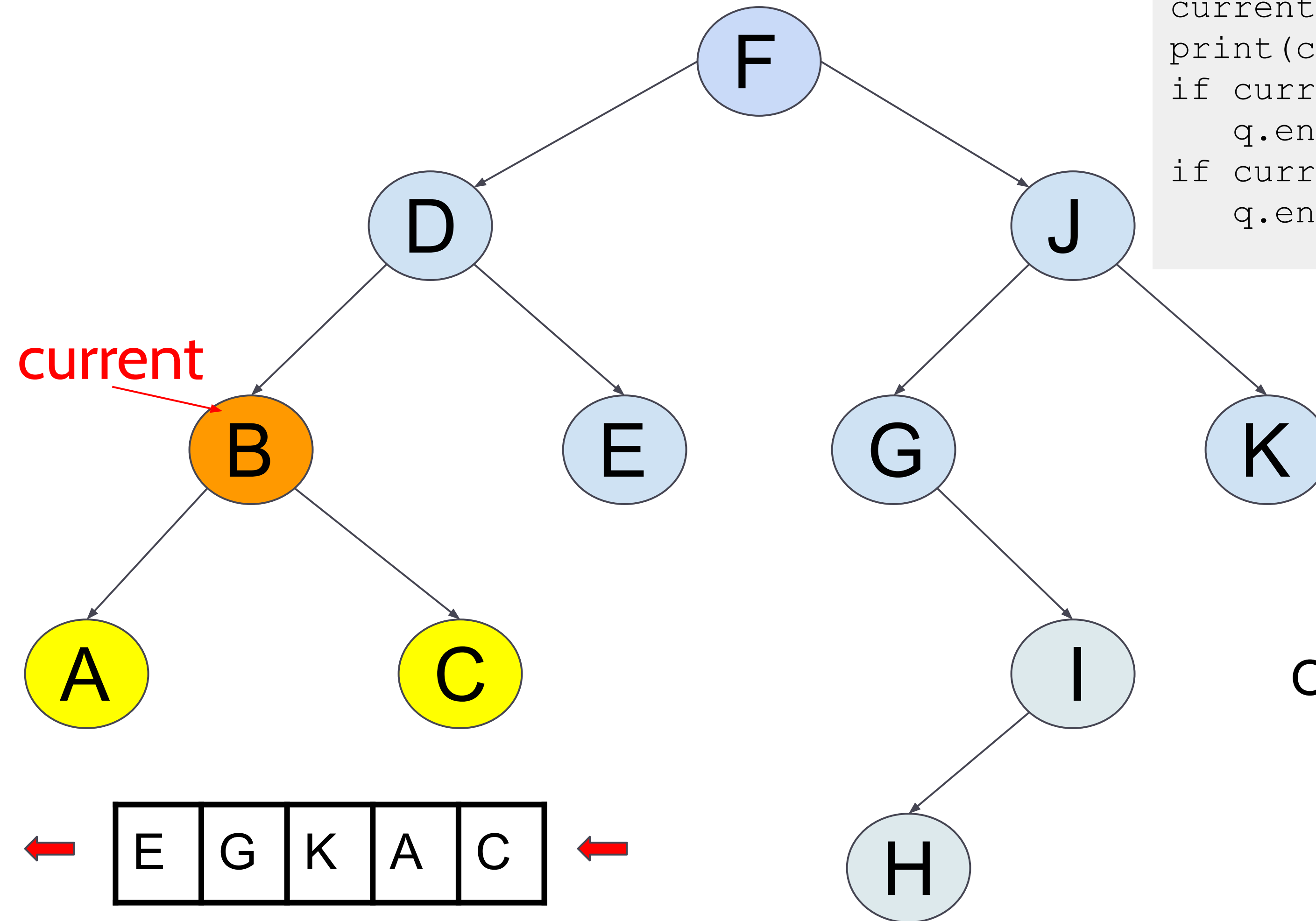

Output: F D J



q: Queue

Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```

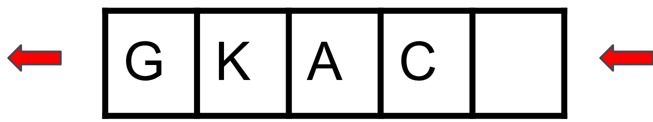


Output: F D J **B**

Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```


Output: F D J B **E**



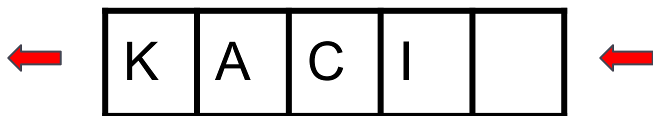
q: Queue

Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```

current

Output: F D J B E **G**



q: Queue

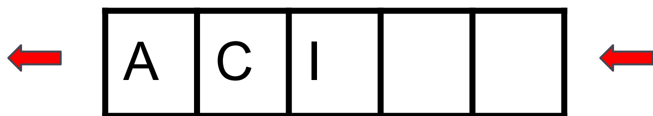
Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```

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

current

Output: F D J B E G K

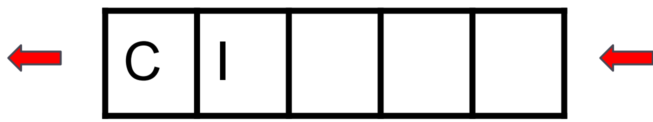


q: Queue

Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
    q.enqueue(current.left)
if current.right!=None:
    q.enqueue(current.right)
```

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



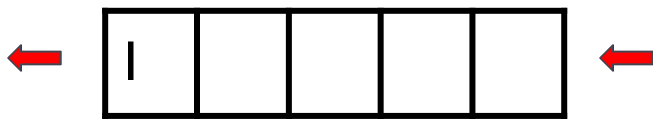
Output: F D J B E G K A

q: Queue

Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```


Output: F D J B E G K A **C**

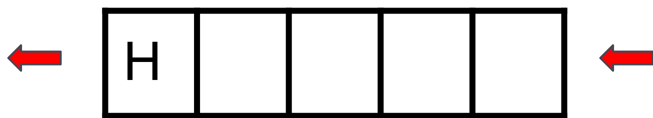


q: Queue

Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```


Output: F D J B E G K A C I

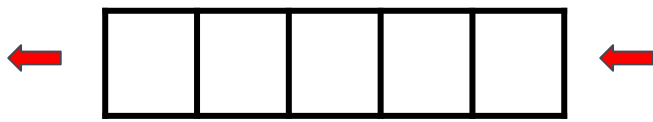


q: Queue

Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```


Output: F D J B E G K A C I **H**



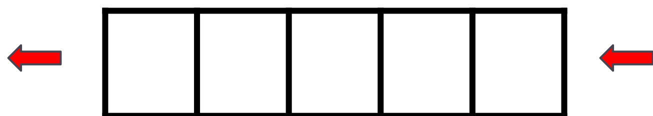
q: Queue

Binary Tree ADT: implementation (level-order method)

```
current=q.dequeue()
print(current.elem)
if current.left!=None:
   q.enqueue(current.left)
if current.right!=None:
   q.enqueue(current.right)
```

Output: F D J B E G K A C I **H**

The queue is empty!!!



q: Queue

Binary Tree ADT: implementation (level-order method)

```
Algorithm levelorder():
   If root==None:
      Show message('tree is empty')
      return

  q=Queueu() #queue of binary nodes
  q.enqueue(root)
  while !q.isEmpty():
      current=q.dequeue()
      print(current.elem)
      if current.left!=None:
         q.enqueue(current.left)
      if current.right!=None:
         q.enqueue(current.right)
```

Index

- Introduction (basic concepts)
- Binary Tree ADT
- **Binary Search Tree ADT**
- Balanced trees

Binary Search Tree (BST)

What data structure should you use to store a modifiable collection?

- *Search(x)*
- *Insert(x)*
- *Remove(x)*

Binary Search Tree (BST)

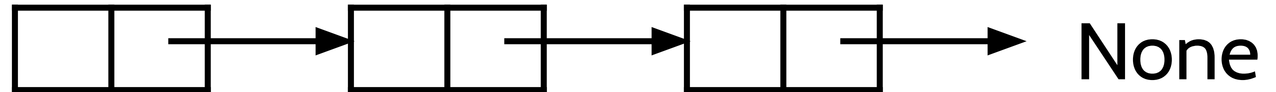
Python List

0	1	2	3	4	5	6	7
2	**-5**	**18**	**0**	**3**	**2**	**5**	**5**

operation	**Time complexity**
search(x)	O(n)
insert(x)	O(1) / O(n)
remove(x)	O(1) / O(n)

Binary Search Tree (BST)

Linked List



operation	Time complexity
search(x)	O(n)
insert(x)	O(1) / O(n)
remove(x)	O(1) / O(n)

Binary Search Tree (BST)

- Let's say that the cost of 1 comparison = 10^{-6} seconds.
- If we have to perform a search in Facebook (with more than 1 billion of users):
  - 1 billion (10^8) x 10^{-6} seconds = 100 seconds!!!

Binary Search Tree (BST)

Python List (sorted)

0	1	2	3	4	5	6	7	8
2	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**

n

$n/2$

$n/4$

$n/8$

…

$n/2^k = 1$

$n = 2^k$

$$\mathbf{k = \log_2 n}$$

operation	**Time complexity**
search(x)	$O(\log_2 n)$

Binary Search Tree (BST)

- Let's say that the cost of 1 comparison = 10^{-6} seconds.
- If we have to perform a search in Facebook (with more than 1 billion of users):

  n=1 billion (10^8) => $\log^2(10^8)$ = 18.42 comparisons
  18.42 x 10^{-6} seconds = 0.00001 seconds << 100 seconds

Binary Search Tree (BST)

Python List (sorted)

0	1	2	3	4	5	6	7	8
2	**5**	**8**	**10**	**13**	**20**	**23**	**50**	**90**

operation	**Time complexity**
search(x)	$O(\log_2 n)$
insert(x)	$O(n)$
remove(x)	$O(n)$

Binary Search Tree (BST)

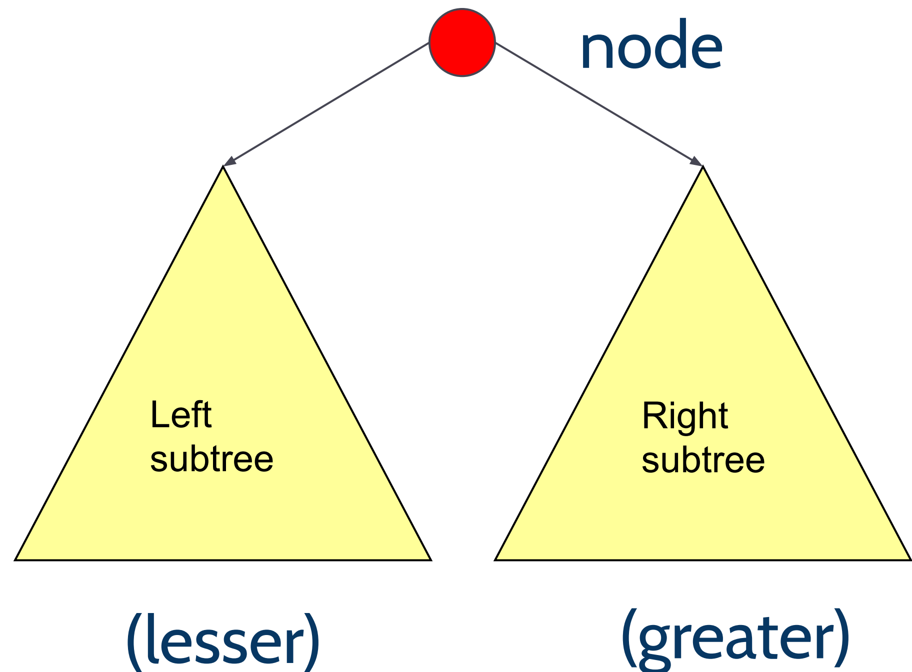
Binary Search Trees

operation	Time complexity
search(x)	$O(\log_2 n)$
insert(x)	$O(\log_2 n)$
remove(x)	$O(\log_2 n)$

Binary Search Tree (BST)

- A binary tree in which for each node, value of all the nodes in its left subtree is lesser and value of all the nodes in its right subtree is greater

Left < node < Right



Binary Search Tree (BST)

Is it a binary search tree?

Yes, it is!!!

Binary Search Tree (BST)

Is it a binary search tree?

10

5 20

3 7 15 30

1 6 21

16

Binary Search Tree (BST)

Is it a binary search tree?

10

5 20

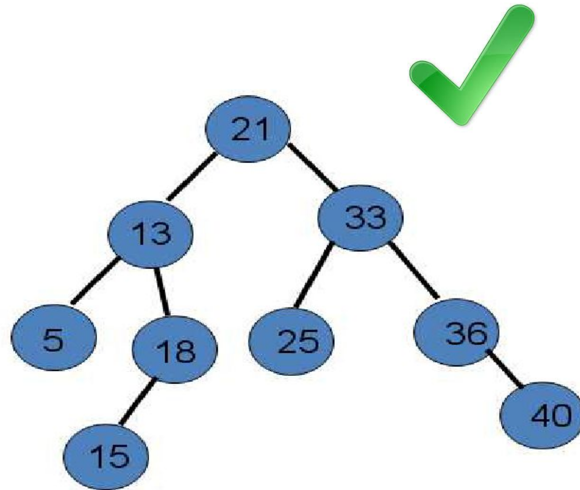
3 7 15 30

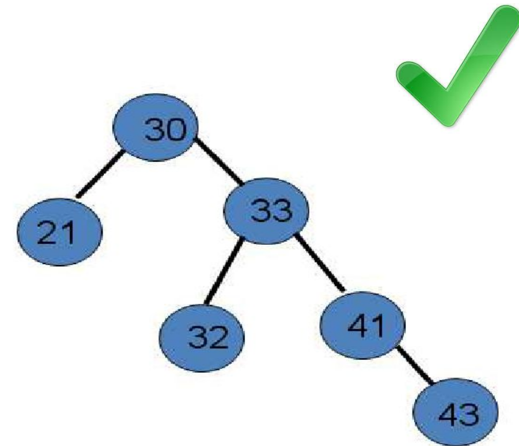
1 6 21

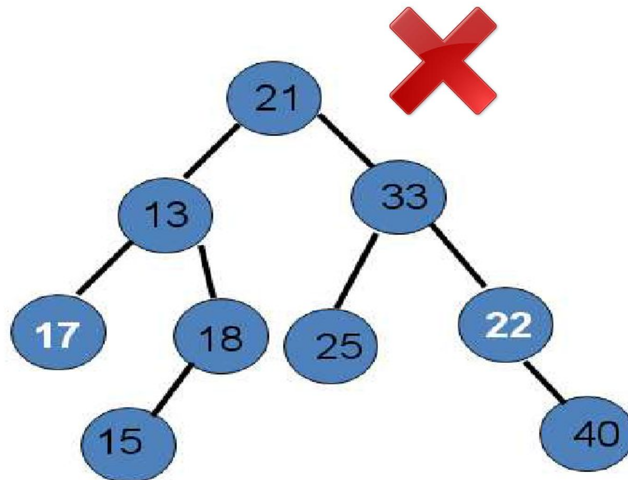
16

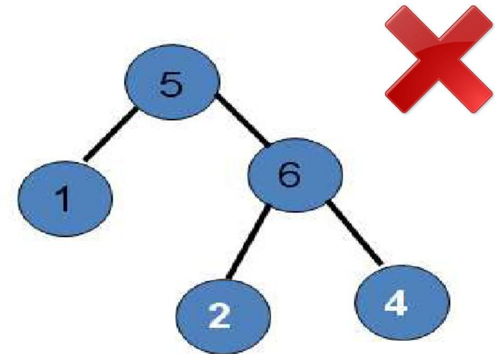
It is not a BST

Binary Search Tree (BST)









Index

- Introduction (basic concepts)
- Binary Tree ADT
- **Binary Search Tree ADT**
  - **search**
  - insert
  - remove
- Balanced trees

Binary Search Tree (BST) - search

search(16)

We must move through the tree until
to reach a node with this value.
We must return True.

Binary Search Tree (BST) - search

search(15)

10

5 20

3 7 15 30

1 4 17

16

If the node does not exist, we stop our search when we reach None. The, we must return False

None

Binary Search Tree (BST) - search

```
Algorithm search(T,x):
    searchNode(T.root,x)

Algorithm searchNode(node,x):
    If node is None:
        return False

    If node.elem==x:
        return True

    …….
```

Base Cases for the recursive function

… and the recursive cases???

Binary Search Tree (BST) - search

```
Algorithm search(T,x):
   searchNode(T.root,x)

Algorithm searchNode(node,x):
   If node is None:
      return False

   If node.elem==x:
      return True

   If x<node.elem:
      return searchNode(node.left,x)

   If x>node.elem:
      return searchNode(node.right,x)
```

Recursive cases

Binary Search Tree (BST) - search

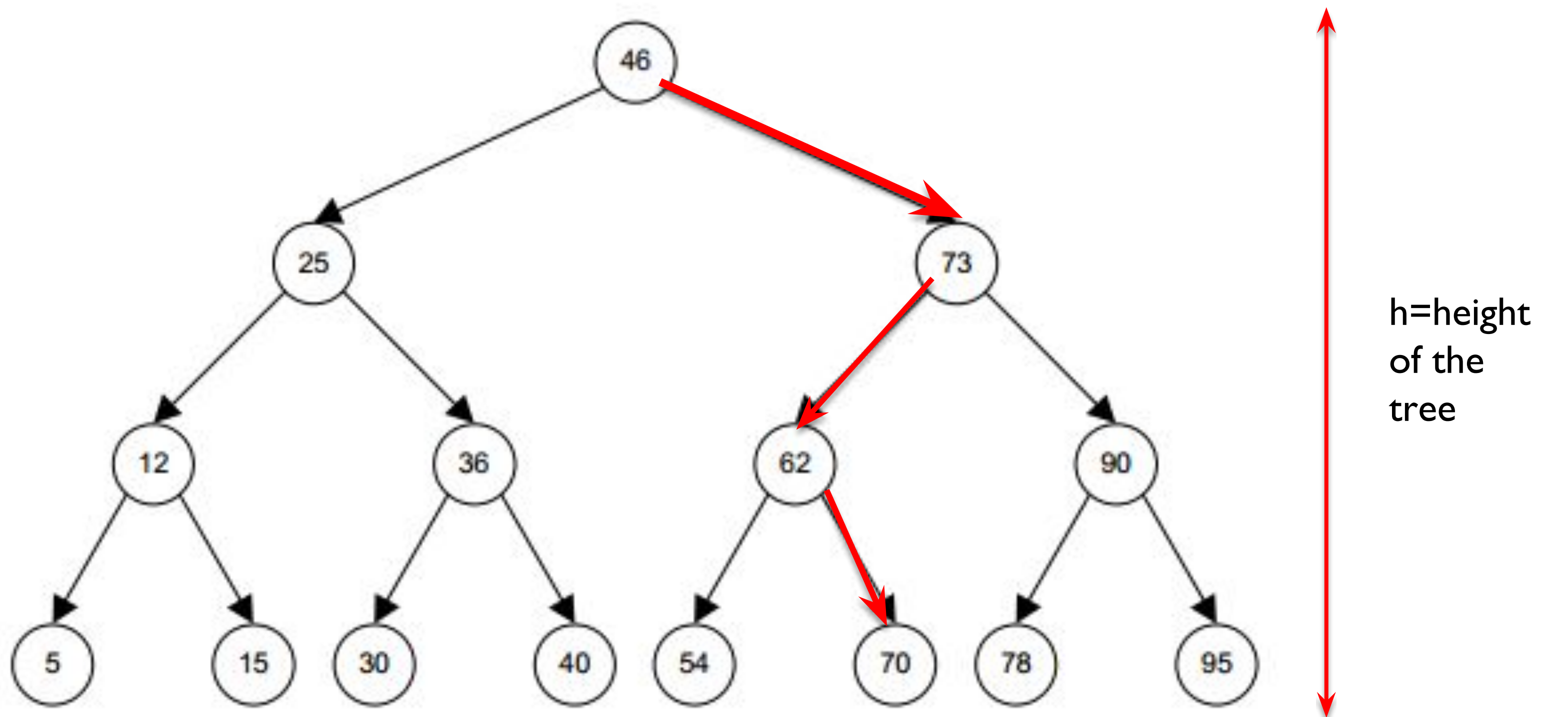
Exercise: Implement a non-recursive method to search an element into a BST

Binary Search Tree (BST) - search

```
Algorithm searchIte(T,x):
   current=T.root
   while current:
      if current.elem==x:
         return True
      if x<current.elem:
         current=current.left
      else:
         current=current.right

   return False
```

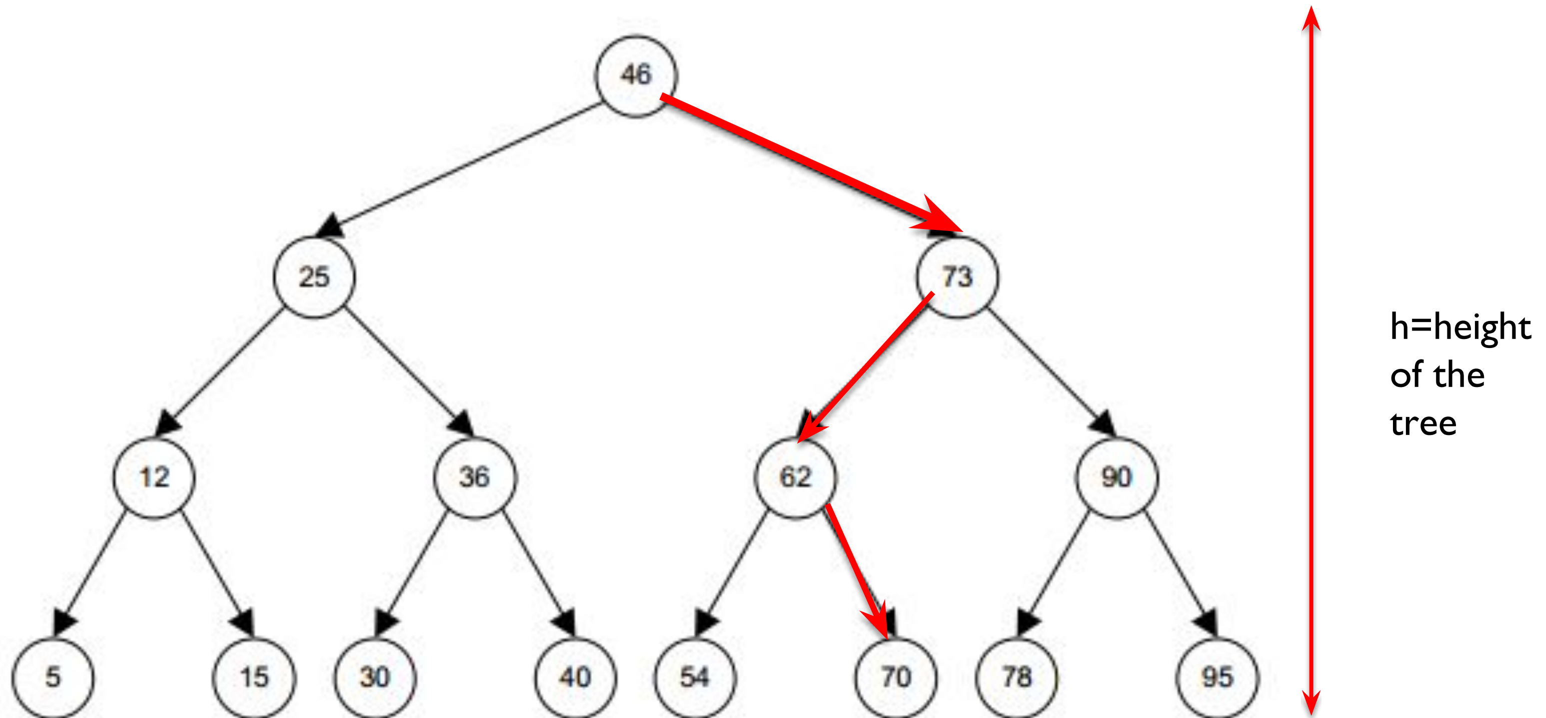
Binary Search Tree (BST) - Big-O for search

In the worst case, the search needs h comparisons, where h is the height of the tree (for example, search(70))



Binary Search Tree (BST) - Big-O for search

Therefore, **time complexity for search** will be O(h) where h is the height of the tree.



Binary Search Tree (BST) - Big-O for search

search(70)

46, 25, 73, 12, 36, 62, 90, 5, 15, 30, 40, 54, 70, 78, 95

n

n/2

Binary Search Tree (BST) - Big-O search

search(70)

46

25 73

12 36 62 90

5 15 30 40 54 70 78 95

n

$n/2$

$n/2^2$

Binary Search Tree (BST) - Big-O search

search(70)

46

25 73

12 36 62 90

5 15 30 40 54 70 78 95

n

$n/2$

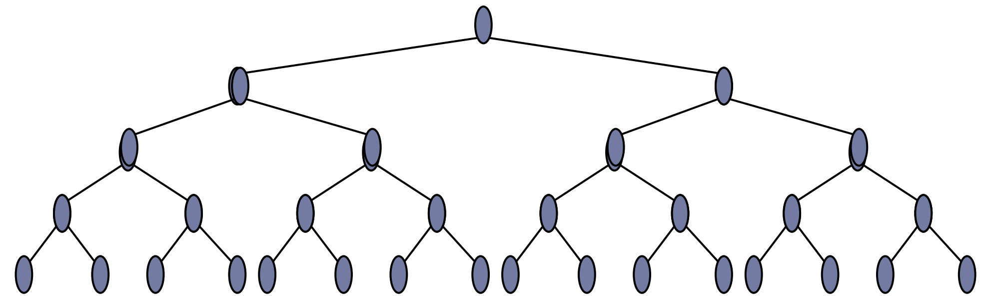
$n/2^2$

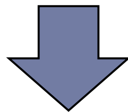
$n/2^3$

Binary Search Tree (BST) - Big-O search

The search space is always divided by 2

- $Step_1 = n / 2^1$
- $Step_2 = n / 2^2$
- $Step_3 = n / 2^3$
- .
- .
- $Step_h = n / 2^h = 1$, where h is the height of the tree





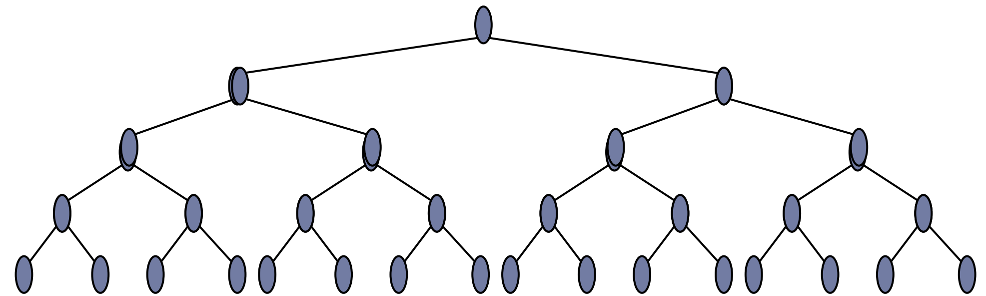
$n = 2^k$
$k = \log_2(n)$

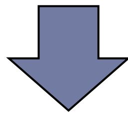
In the worst case, the number of steps to find a node is the height of the tree = h

Binary Search Tree (BST) - Big-O search

The search space is always divided by 2

- $Step_1 = n / 2^1$
- $Step_2 = n / 2^2$
- $Step_3 = n / 2^3$
- .
- .
- $Step_h = n / 2^h = 1$, where h is the height of the tree

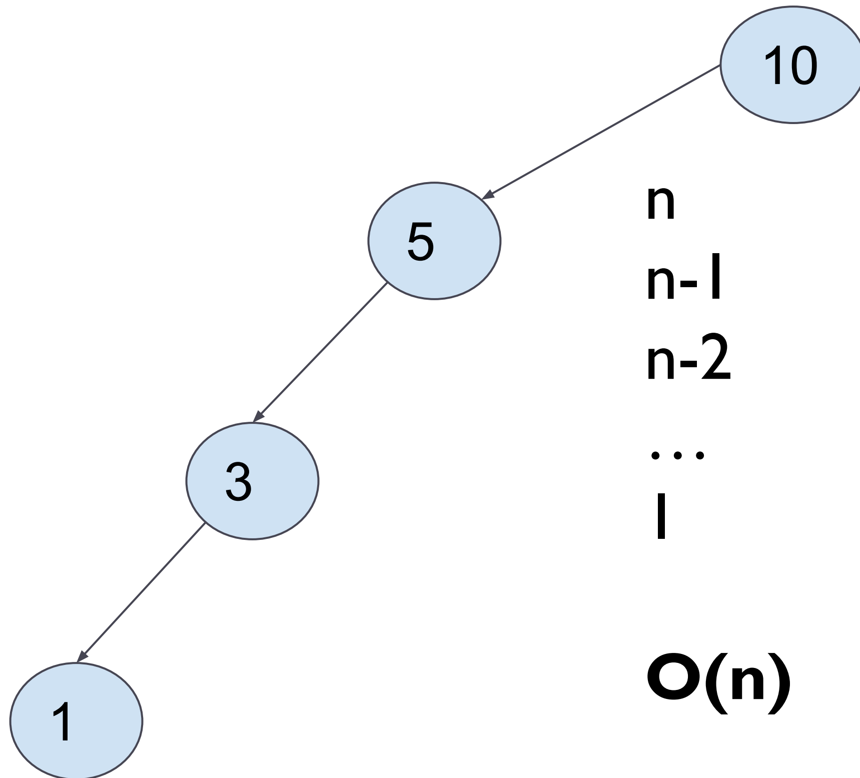




$n = 2^h$
$h = \log_2(n)$

$O(h)=O(\log_2 n)$

Binary Search Tree (BST) - Big-O search



In this BST, the search space will be reduced by only one at each step!!!!

Therefore, we should keep the BST balanced!!! (next lesson).

Index

- Introduction (basic concepts)
- Binary Tree ADT
- **Binary Search Tree ADT**
  - search
  - **insert**
  - remove
- Balanced trees

Binary Search Tree (BST) - insert

insert(8)

10

5 20

3 7 15 30

None

1 4 17

16

We must move through the tree until to reach the right position for the new node (or until to find a node with the same value)

Binary Search Tree (BST) - insert

insert(8)

10

5 20

3 7 15 30

1 4 8 17

16

Finally, we must create the new node and link it to its parent node.

Binary Search Tree (BST) - insert

insert(20)

10

5 20

3 7 15 30

1 4 17

16

If the node already exists, we can show a message such as 'That key already exists!!!'

Binary Search Tree (BST) - insert

```
Algorithm insert(T,x):
    If T.root is None:
        T.root=BinaryNode(x)
    Else:
        insertNode(T.root,x)
```

If the tree is empty (root=None), we create a new node, which will be the root.

```
Algorithm insertNode(node,x):
    If node.elem==x:
        Show message "x already exists!!!"
        Return
        ...
```

We do not allow duplicate elements!!!

Binary Search Tree (BST) - insert

```
Algorithm insert(T,x):
    If T.root is None:
        T.root=BinaryNode(x)
    Else:
        insertNode(T.root,x)

Algorithm insertNode(node,x):
    If node.elem==x:
        Show message "x already exists!!!"
        Return
        ...
```

... and the recursive cases???

Implementing a BST- insert

```
Algorithm insert(T,x):
    If T.root is None:
        T.root=BinaryNode(x)
    Else:
        insertNode(T.root,x)

Algorithm insertNode(node,x):
    If node.elem==x:
        Show message "x already exists!!!"
        Return
    If x<node.elem:
        If node.left is None:
            newNode=BinaryNode(x)
            node.left=newNode
            newNode.parent=node
        Else:
            insertNode(node.left,x)
```

If the node doesn't have left child, we have already found the position for the new node.

We must continue searching the position for the new node

Implementing a BST- insert (continue)

```
…
Else: #if x>node.elem:
    If node.right is None:
        newNode=BinaryNode(x)
        node.right=newNode
        newNode.parent=node
    Else:
        insertNode(node.right,x)
```

If the element is greater than the node's element, we must continue searching on its right child.

If there is no right child, we have just found the position for the new node.
If there is right child, we must continue the search by calling the recursive method on the right child.

Index

- Introduction (basic concepts)
- Binary Tree ADT
- **Binary Search Tree ADT**
  - search
  - insert
  - **remove**
- Balanced trees

Binary Search Tree (BST) - remove

remove(16)

10
5
20
3
7
15
30
1
4
17
16

Binary Search Tree (BST) - remove

remove(16)

10

5

20

3

7

15

30

1

4

17

16

Binary Search Tree (BST) - remove

Firstly, we must **search the node to be removed**.

Binary Search Tree (BST) - remove

Firstly, we must search the node to be removed.

```
Algorithm find(T,x):
   return findNode(T.root,x)

Algorithm findNode(node,x):
   If node is None:
      return None
   If node.elem==x:
      return node

   If x<node.elem:
      return findNode(node.left,x)
   If x>node.elem:
      return findNode(node.right,x)
```

Base Cases

Recursive Cases

Binary Search Tree (BST) - remove

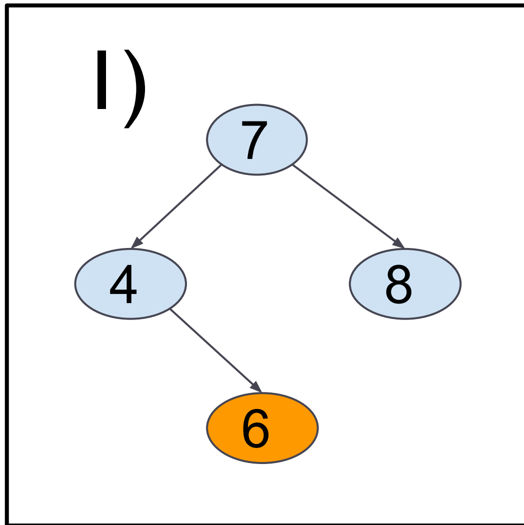
Then, we must implement the removeNode method.

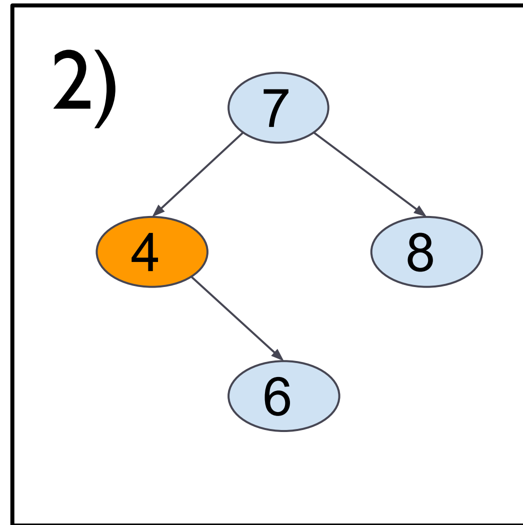
```
Algorithm remove(T,x):
   node=find(T,x)
   If node is None:
      Show message "node to be removed not found"
      return

   removeNode(T,node)

Algorithm removeNode(T,node):
   # we must check if the node:
      #1) doesn't have children,
      #2) has only one child
      #3) has two children.
```

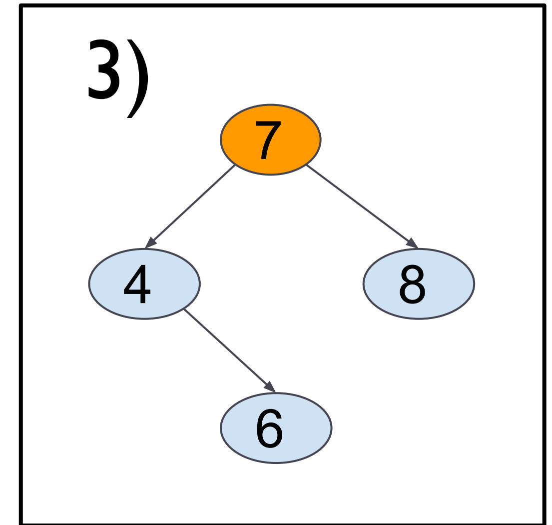
Binary Search Tree (BST) - remove

There are three possible cases:

1) The node to be deleted is a **leaf**.
2) The node to be deleted only **has only one child**.
3) The node to be removed **has two children**.

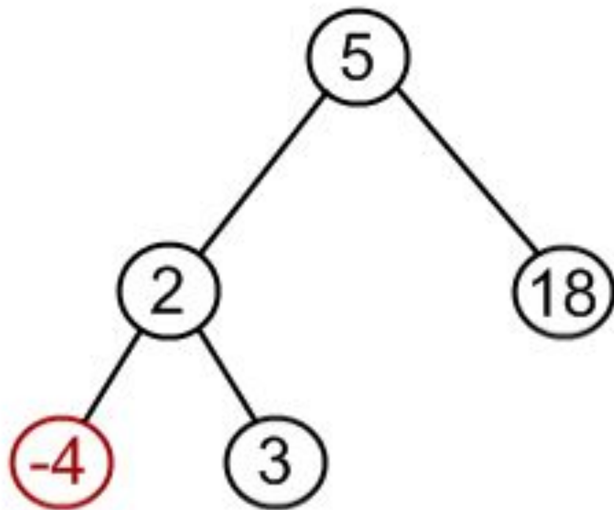


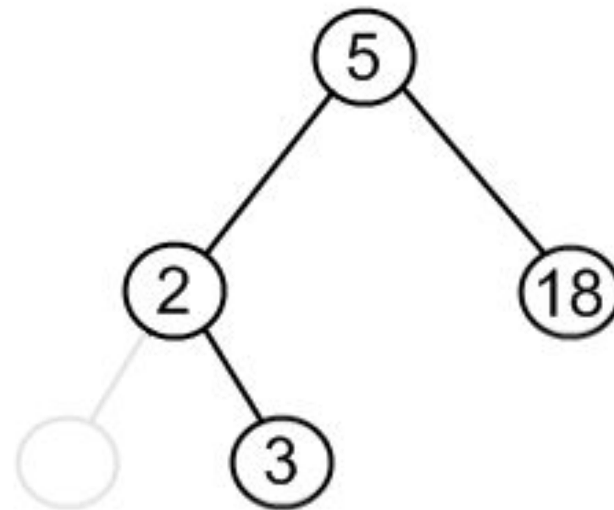




Binary Search Tree (BST) - remove

FIRST CASE: The node to be removed is a leaf
=> The reference from the parent to the node must be broken.



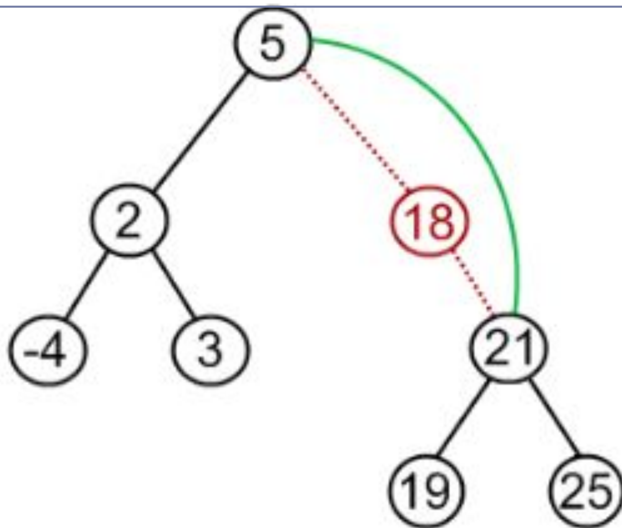


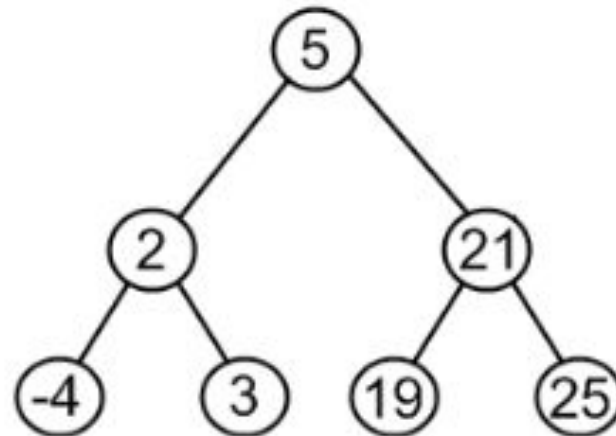
Binary Search Tree (BST) - remove

```
Algorithm removeNode(T,node):

    # First case: node doesn't have any children

    if node.left is None and node.right is None:
        If node.parent is not None:
            If node.parent.left is node:
                node.parent.left=None
            Elif node.parent.right is node:
                node.parent.right=None
            node.parent = None
        Else:
            T.root=None #we are removing the root
```

Binary Search Tree (BST) - remove

SECOND CASE: The node to be removed has only one child => The child must be connected with the parent of the node to be removed





Binary Search Tree (BST) - remove

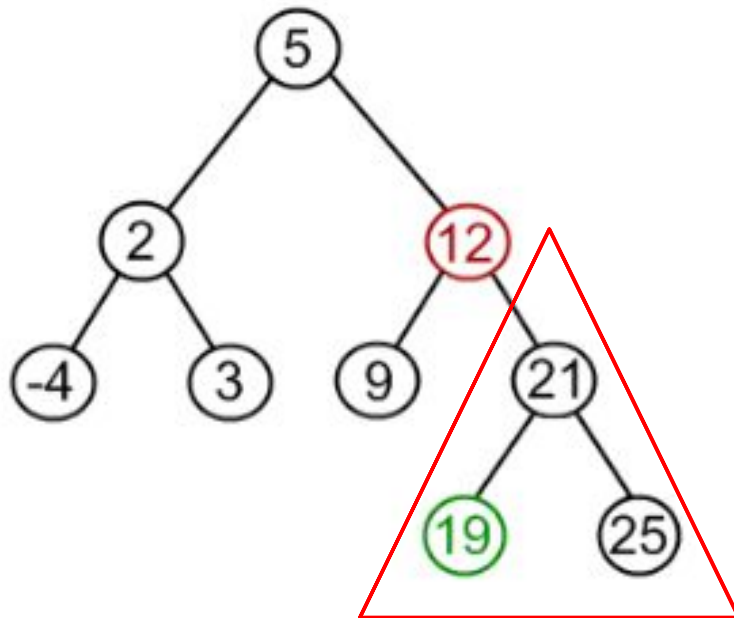
```
Algorithm removeNode(T,node):
              …
  # Second case: only one child (the left child)
  if node.left is not None and node.right is None:
    If node.parent is not None:
         If node.parent.left is node:
            node.parent.left=node.left
         Elif node.parent.right is node:
            node.parent.right=node.left

         node.left.parent = node.parent
    Else:
         T.root=node.left #we are removing the root
```

Binary Search Tree (BST) - remove

```
Algorithm removeNode(T,node):

            …
  # Second case: only one child (the right child)
  if node.left is not None and node.right is not None:
      If node.parent is not None:
            If node.parent.left is node:
               node.parent.left=node.right
            Elif node.parent.right is node:
               node.parent.right=node.right

            node.right.parent = node.parent

      Else:
            T.root=node.right #we are removing the root
```

Binary Search Tree (BST) - remove

THIRD CASE: The node to be removed has two children

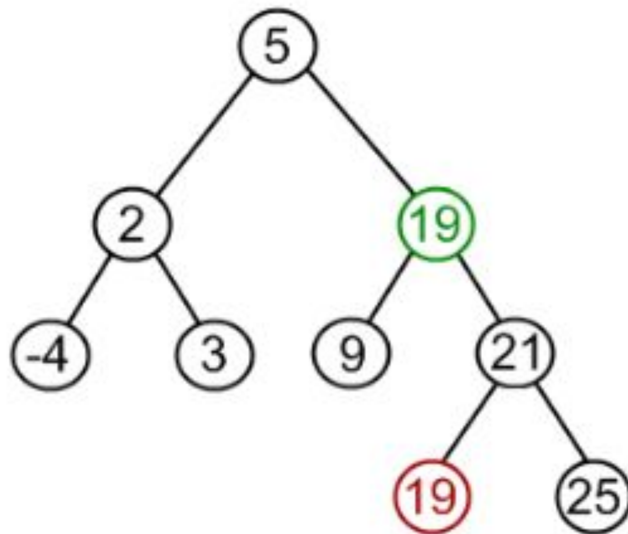


remove(12)

1) We search the **successor** for the node to be removed. The successor is the node in the child subtree with the smallest element.

Binary Search Tree (BST) - remove

THIRD CASE: The node to be removed has two children

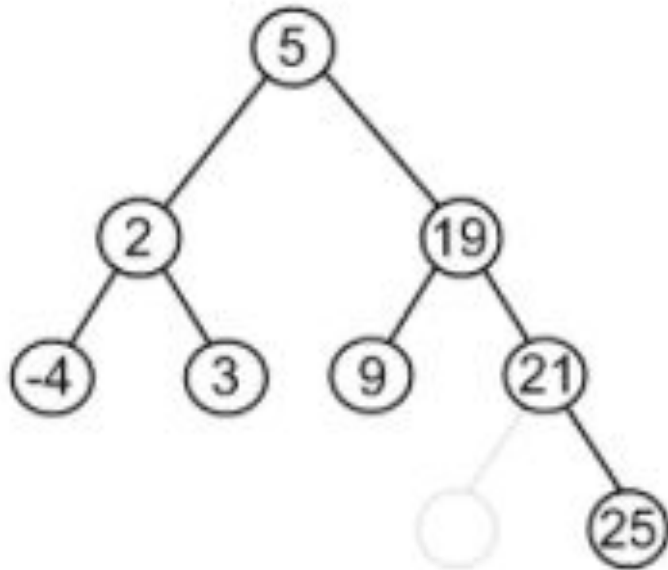


remove(12)

1) We search the successor for the node to be removed.
2) **The node's element must be replaced with the successor's element**

Binary Search Tree (BST) - remove

THIRD CASE: The node to be removed has two children



remove(12)

1) We search the successor for the node to be removed.
2) The node's element must be replaced with the successor's element.
3) **Finally, we must remove the successor from the right child.**

Binary Search Tree (BST) - remove

```
Algorithm removeNode(T,node):
              …
   # Third case: two children
   if node.left is not None and node.right is not None:
      #we must find the successor of the node
      successor=node.right
      while successor.left is not None:
         successor=successor.left
```

Binary Search Tree (BST) - remove

```
Algorithm removeNode(T,node):
              …
   # Third case: two children
   if node.left is not None and node.right is not None:
      successor=node.right
      while successor.left is not None:
         successor=successor.left
      #replace node’s element by successor’s element
      node.element=successor.element
```

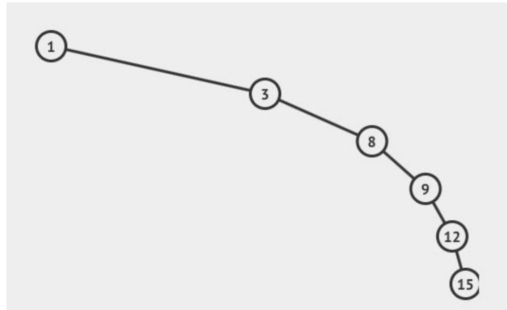
Binary Search Tree (BST) - remove

```
Algorithm removeNode(T,node):
              ...
   # Third case: two children
   if node.left is not None and node.right is not None:
      successor=node.right
      while successor.left is not None:
         successor=successor.left
      node.element=successor.element
      #finally, we remove successor
      removeNode(T,successor)
```

Note that the successor will always be a leaf node or a node with only one child.

Binary Search Tree (BST)

- BSTs improve the time complexity for search, insertion and removal operations.
- Problem: A BST can degenerate in a list



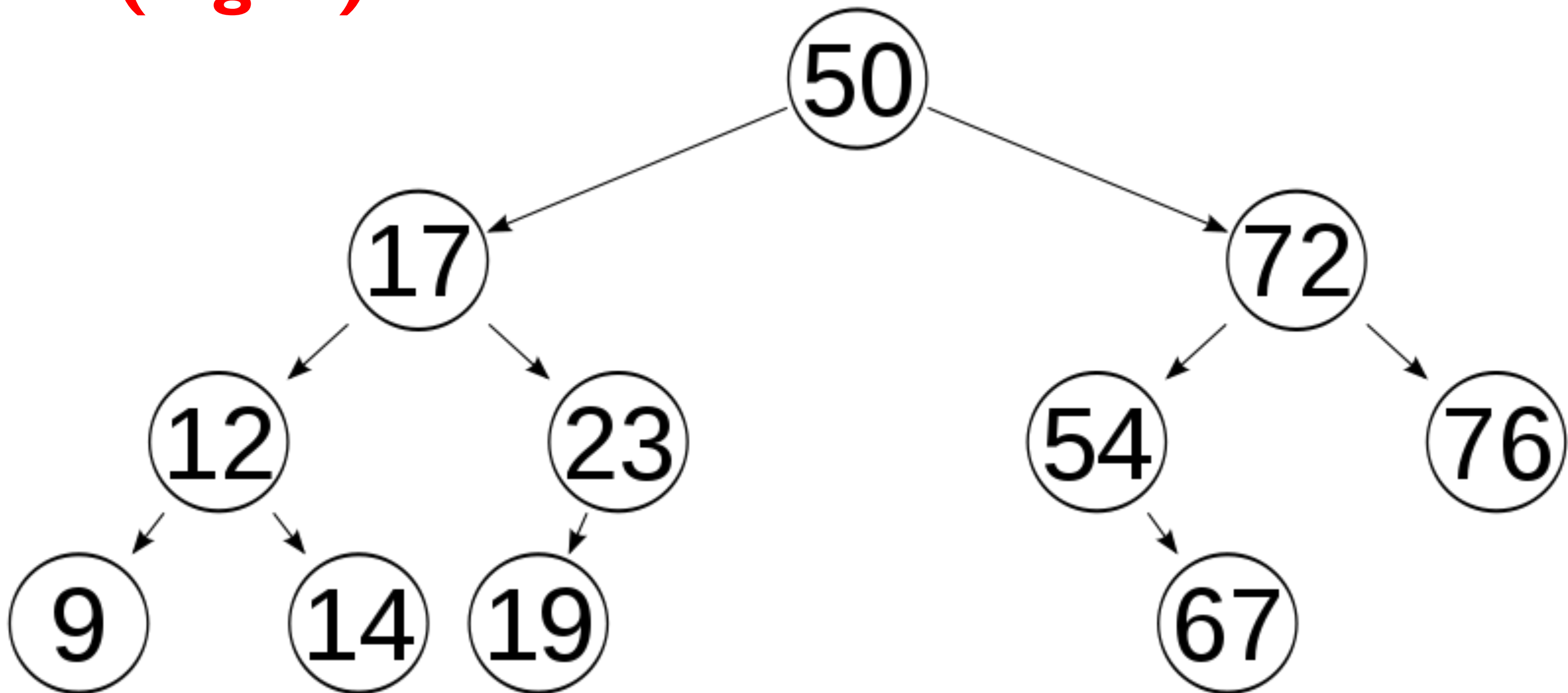
Solution: **Balanced BST (AVL) (next lesson)**

Index

- Introduction (basic concepts)
- Binary Tree ADT
- Binary Search Tree ADT
- **Balanced trees**

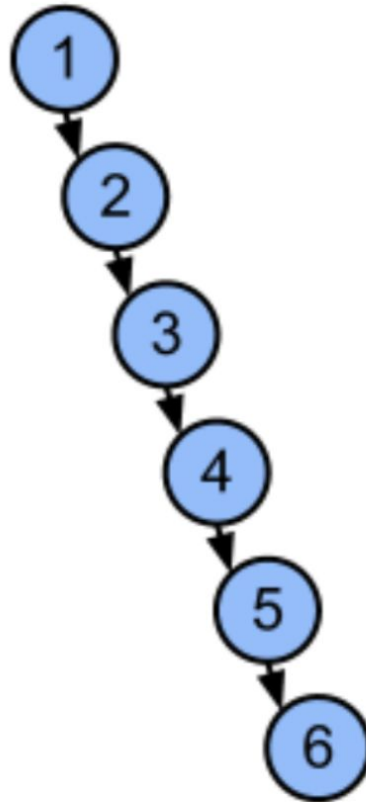
Balanced Trees

- Advantage of BSTs: Insert, remove and search ~ **O(log2n).**



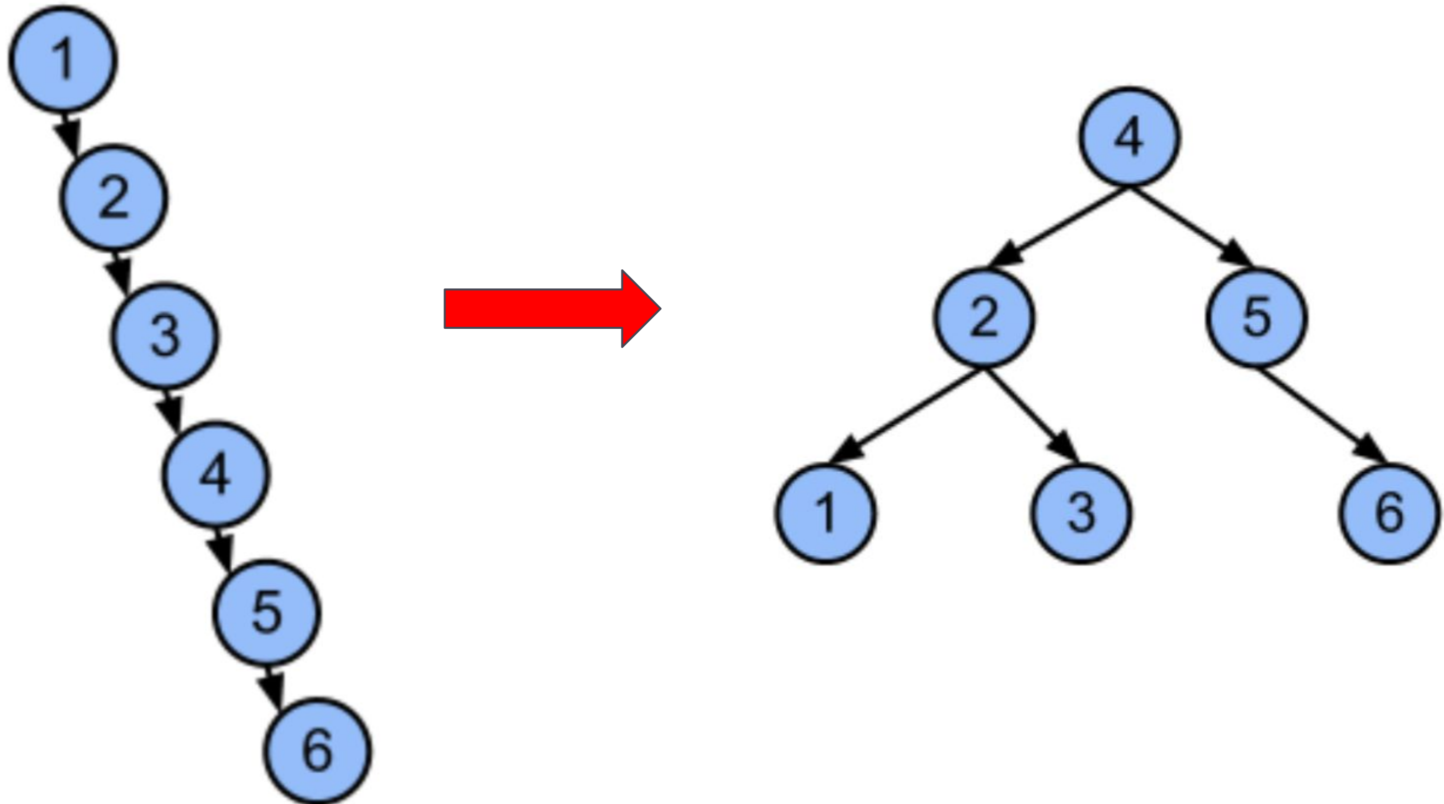
Balanced Trees

- If h (height) ≃ n => **O(h)=O(n).**



Balanced Trees

- Solution: Balancing the BST

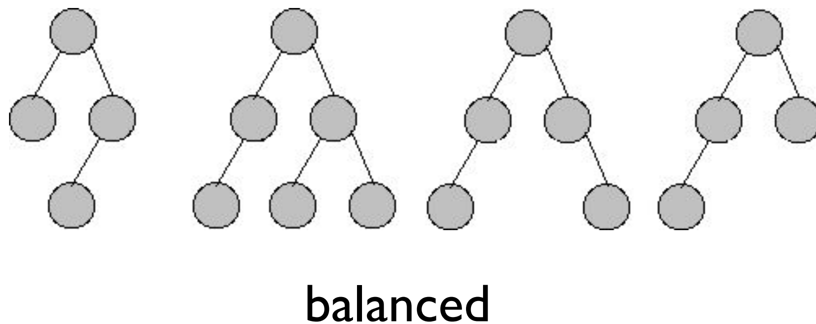


Balanced Trees

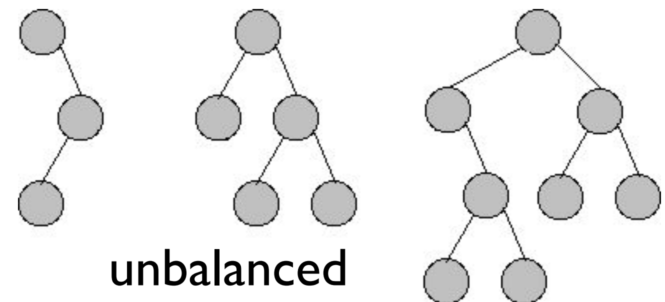
- Solution: keep the tree balanced, how?
  - **Size balance**
  - Height balance (AVL trees, by mathematicians Adelson-Velskii y Landis)

Size-balanced BSTs

- **Balance factor** of a node (fe) : difference between the size of the left subtree and the size of the right subtree (or vice versa)
- ***A BST is size-balanced if*** for EVERY node, if the difference between the number of nodes in left and right subtrees cannot be more than 1.



balanced



unbalanced

Size balanced BSTs: rebalancing algorithm

- **Goal:** Move nodes from the subtree with more nodes to the subtree with less nodes.
  - How many? balance factor / 2 nodes.
  - The rebalancing process is performed **from the root down**

Size balanced BSTs: rebalancing algorithm

- Move to the right
  - Insert the root's elem into the right subtree
  - Get the predecessor of the root (from the left subtree)
  - Replace the root's elem with the predecessor's elem.
  - Remove the predecessor from the left subtree
  - Repeat previous steps as many times as nodes to be moved

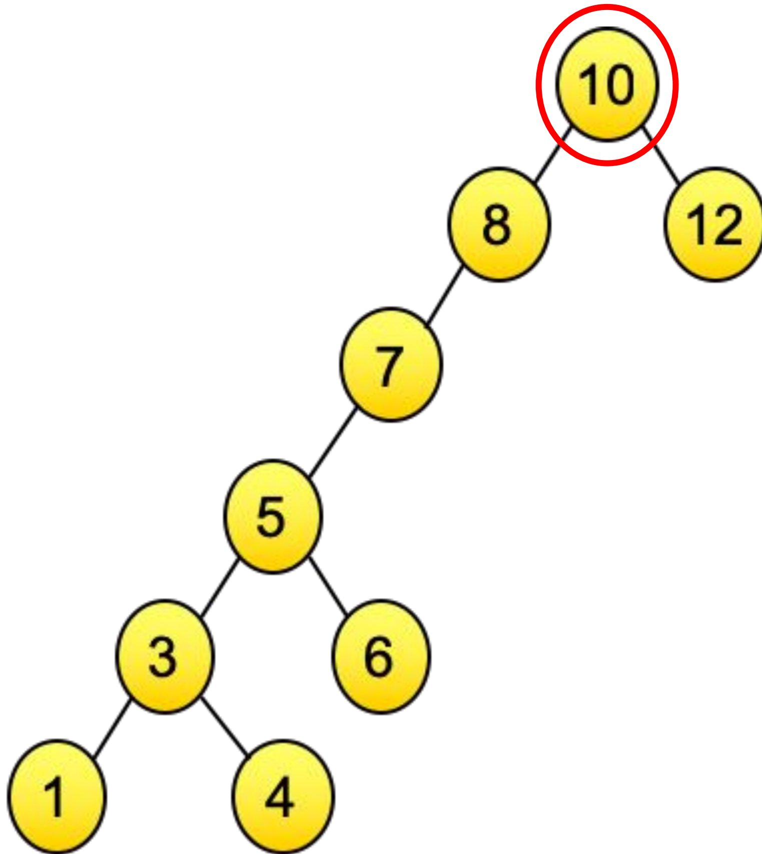
Size balanced BSTs: rebalancing algorithm

- Move to the left: (symmetric)
  - Insert the root’s elem into the left subtree
  - Get the successor of the root (from the right subtree)
  - Replace the root’s elem with the successor's elem.
  - Remove the successor from the right subtree.
  - Repeat previous steps as many times as nodes to be moved.

Size balanced BSTs: rebalancing algorithm

- Modify insert/remove algorithms
  - Rebalancing after each insert/remove operation, or
  - Rebalancing at a given moment
- The cost to maintain a BST balanced is high, O(n)

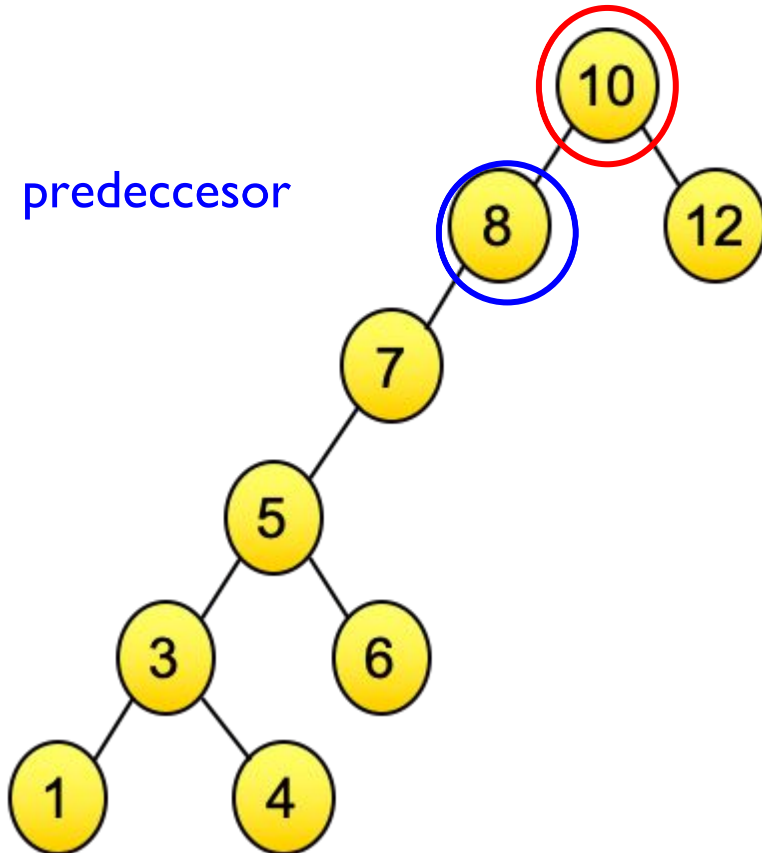
Size balanced BSTs: rebalancing algorithm (example)



- The balancing must be performed **from the root down**
- Balance factor for root = size(root.left)-size(root.right) = 7 - 1 = 6

  How many nodes should we move to right? 6/2= 3

Size balanced BSTs: rebalancing algorithm (example)



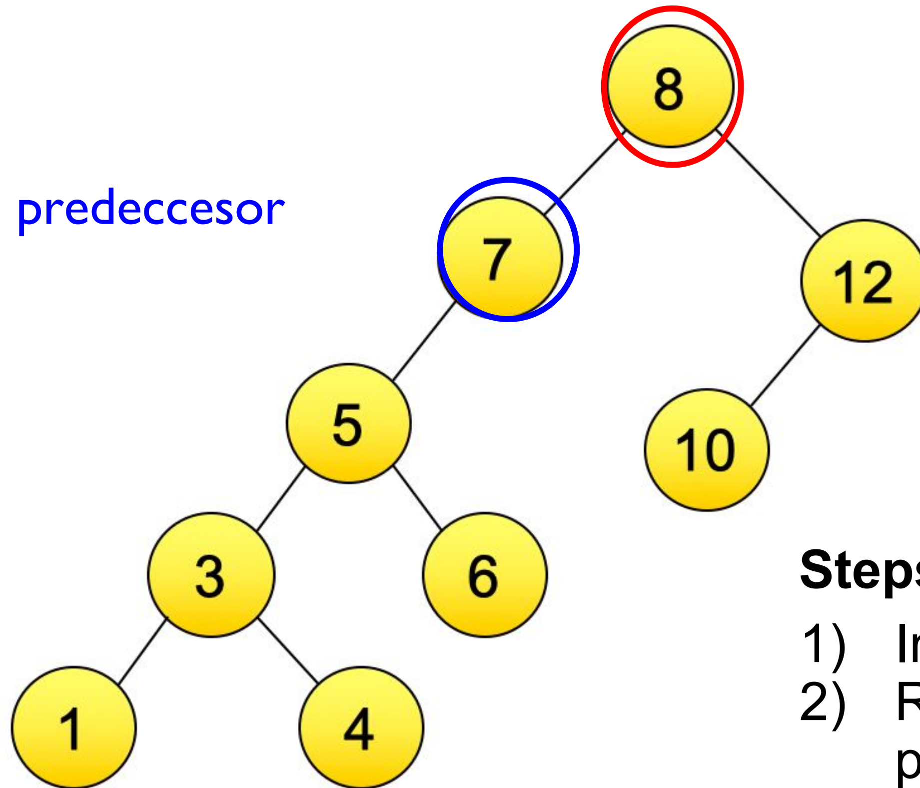
Steps:

1) Insert the root into the right subtree.
2) Replace the root'elem by the predecessor'elem from the left subtree.
3) Remove the predecessor node from the left subtree.

Size balanced BSTs: rebalancing algorithm (example)

predecessor

Size balanced BSTs: rebalancing algorithm (example)



- Balance factor for root = size(root.left)-size(root.right)=6 -2=4

  How many nodes should we move to right? 4/2= 4

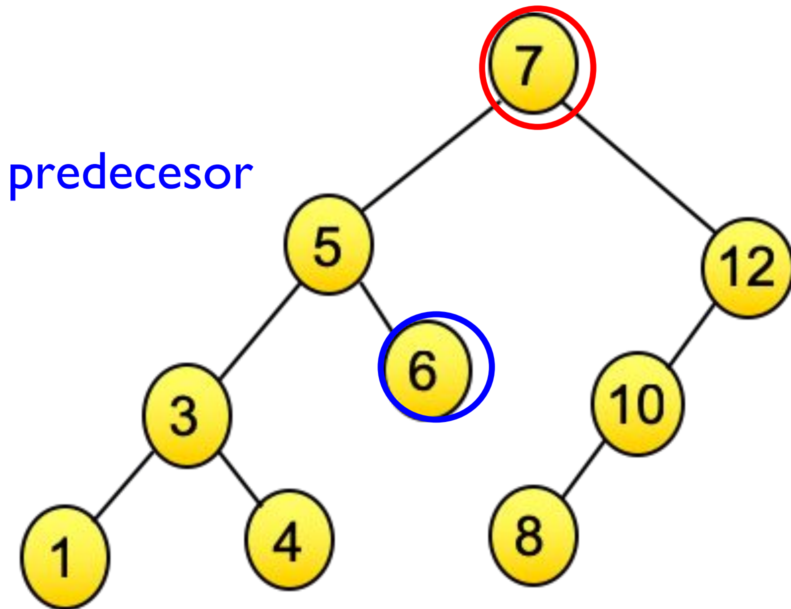
Steps:

1) Insert the root into the right subtree.
2) Replace the root'elem by the predecessor'elem from the left subtree.
3) Remove the predecessor node from the left subtree.

Size balanced BSTs: rebalancing algorithm (example)

predecessor

Size balanced BSTs: rebalancing algorithm (example)



- Balance factor for root (7) = size(root.left)-size(root.right)=5 -3=2

  How many nodes should we move to right? 2/2= 1

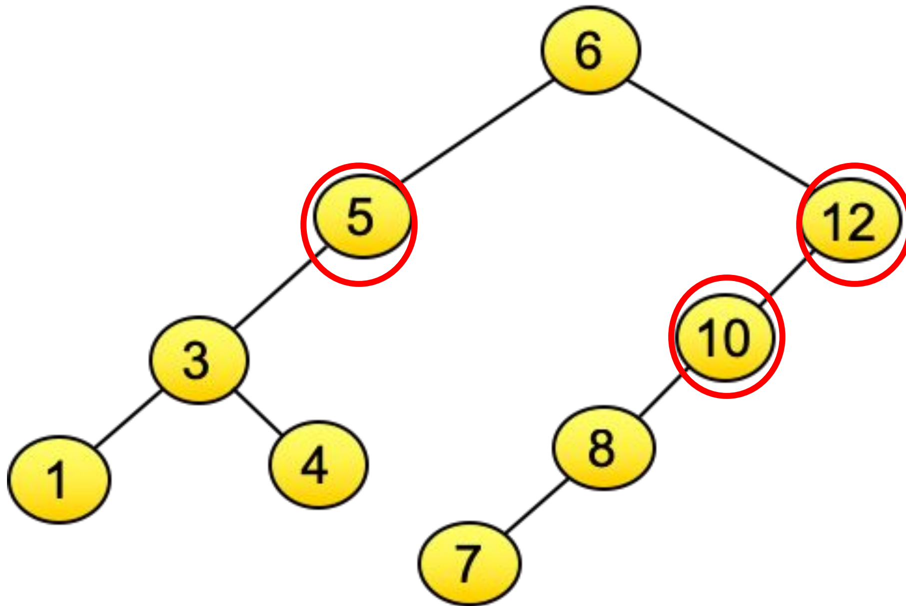
Steps:

1) Insert the root into the right subtree.
2) Replace the root'elem by the predecessor'elem from the left subtree.
3) Remove the predecessor node from the left subtree.

Size balanced BSTs: rebalancing algorithm (example)

predecesor

Size balanced BSTs: rebalancing algorithm (example)



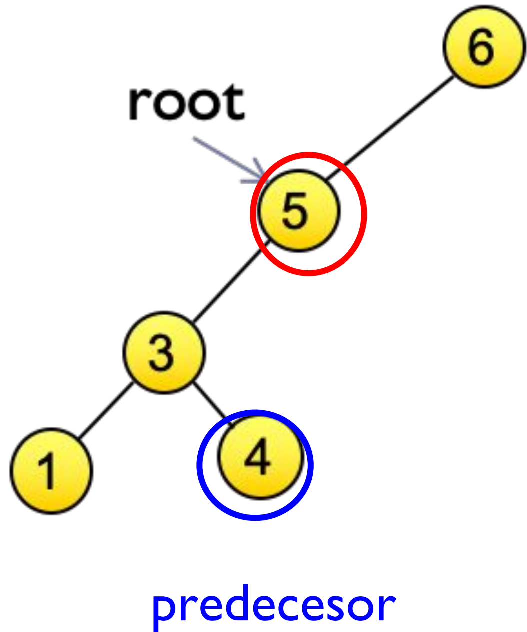
The root is balanced (factor=4-4).

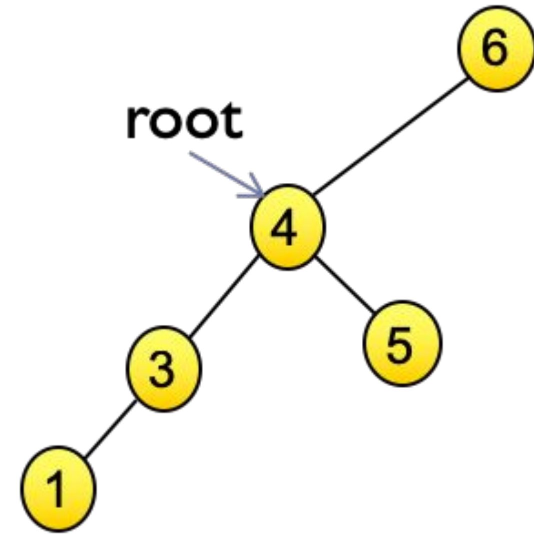
The unbalanced nodes are: 5, 12 and 10.

You should choose 5 or 12 to balance (remember always from the root down).

For example, choose 5

Size balanced BSTs: rebalancing algorithm (example)



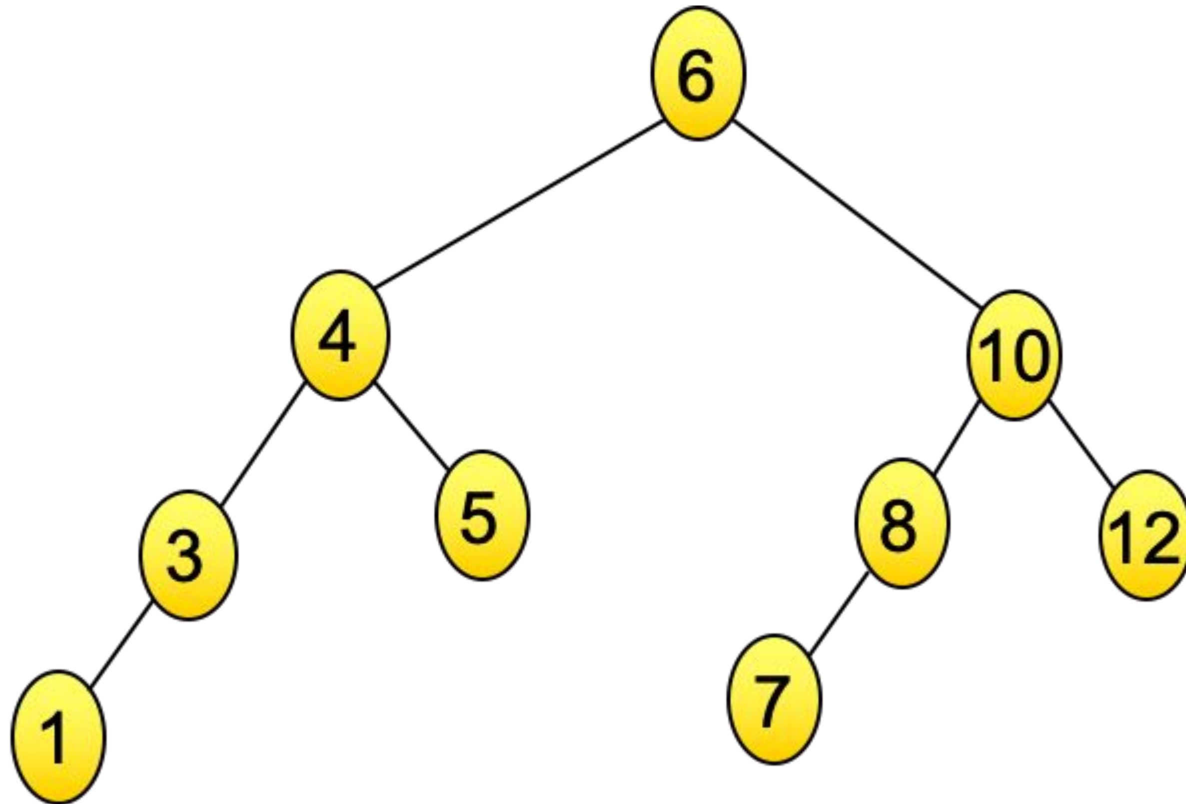


Size balanced BSTs: rebalancing algorithm (example)

6
12
root
10
8
7
predecesor

6
10
root
8
12
7

Size balanced BSTs: rebalancing algorithm (example)



Balanced tree

Size balanced BSTs: rebalancing algorithm (example)

fe=1-5=-4

10

5

20

successor

15

30

17

16

Size balanced BSTs: rebalancing algorithm (example)

fe=1-5=-4

fe=2-4=-2

successor

Size balanced BSTs: rebalancing algorithm (example)

fe=2-4=-2

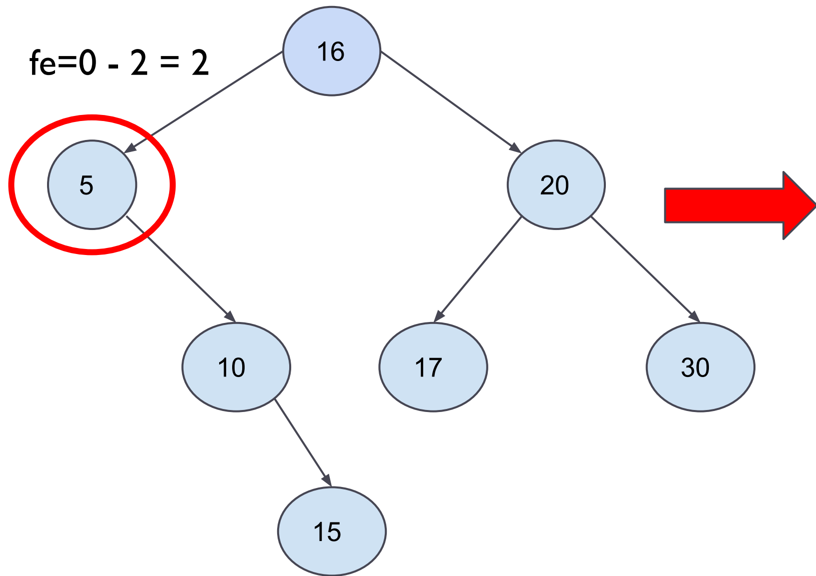
15, 5, 20, 10, 17, 30, 16

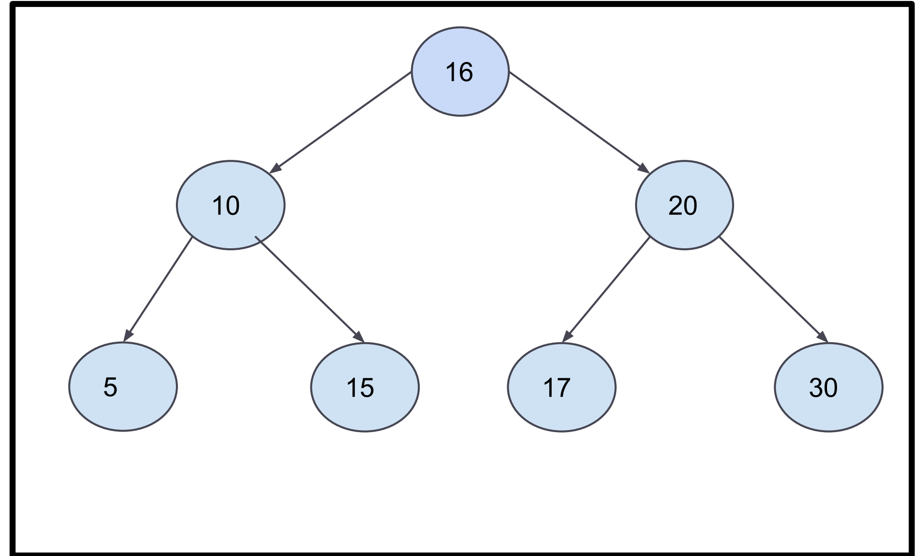
successor

fe=0

16, 5, 20, 10, 17, 30, 15

Size balanced BSTs: rebalancing algorithm (example)





Size-balanced BST

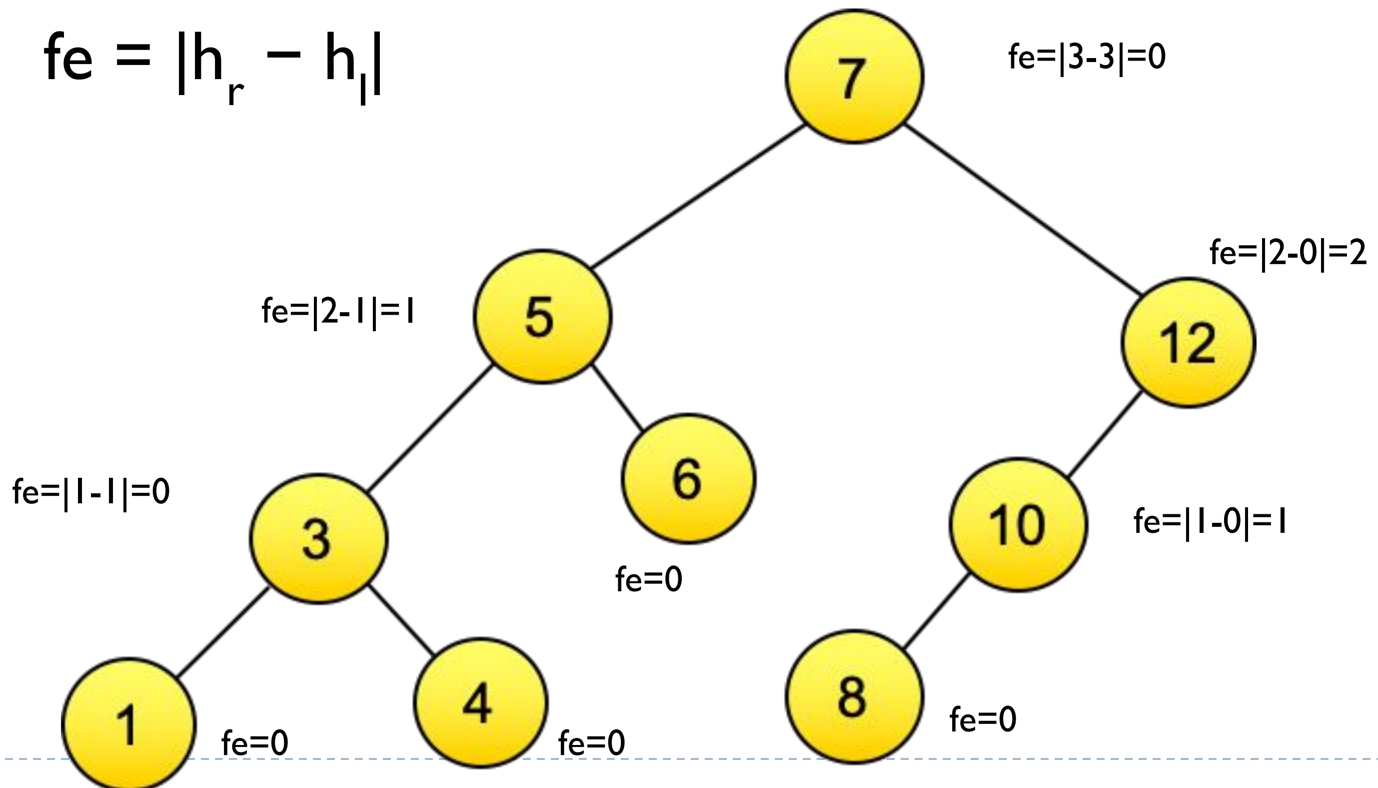
Index

- Introduction (basic concepts)
- Binary Tree ADT
- Binary Search Tree ADT
- **Balanced trees**
  - Size-balanced BST
  - **Height-balanced BST**

Height balanced BSTs v

Height balance factor of a node (fe) :
Difference between the height of the right subtree and the height of the left subtree (o viceversa)

$$fe = |h_r - h_l|$$



Height balanced BSTs v

- Height balanced BST
  - For each node, the difference between the height of the left and right subtrees is 1 maximum

Therefore, the height balance factor of every node must be: -1, 0 or +1

Height balanced BSTs v

- This BST is not height-balanced

7 fe=|3-3|=0

5 fe=|2-1|=1

12 fe=|2-0|=2

3 fe=|1-1|=0

6 fe=0

10 fe=|1-0|=1

1 fe=0

4 fe=0

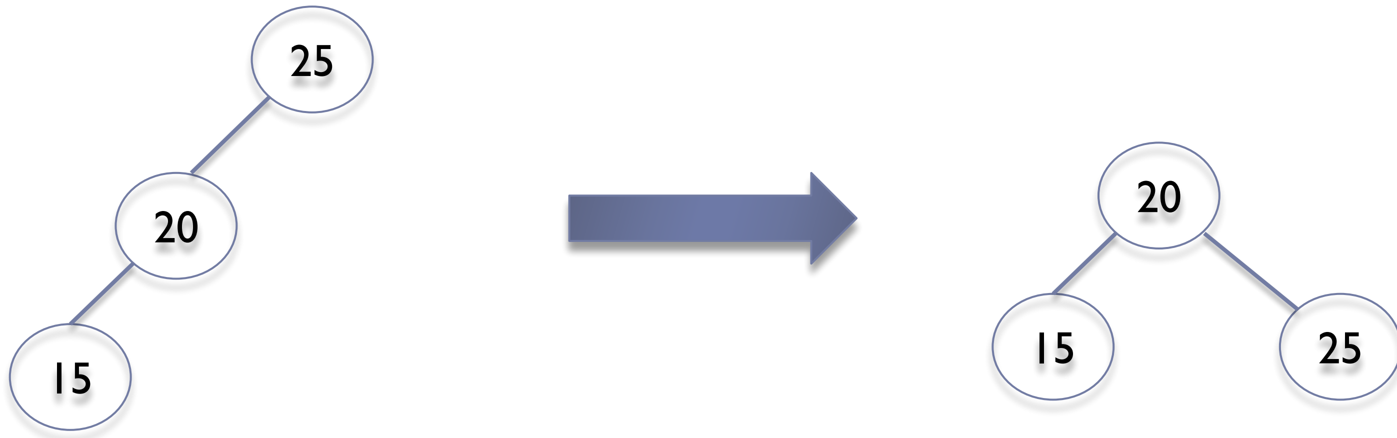
8 fe=0

Height balanced BSTs (AVL)

- **Idea**: move nodes from the longest branch to the shortest branch.
- **Important**: The balancing is done in **ascending order**, i.e., always from the bottom, only in the path from the inserted or removed node to the root node.
- The resulting tree must be a binary search tree (having the same in-order traversal)
- Rotations:
  - Simple right rotation
  - Simple left rotation
  - Double rotation (left-right)
  - Double rotation (right-left)

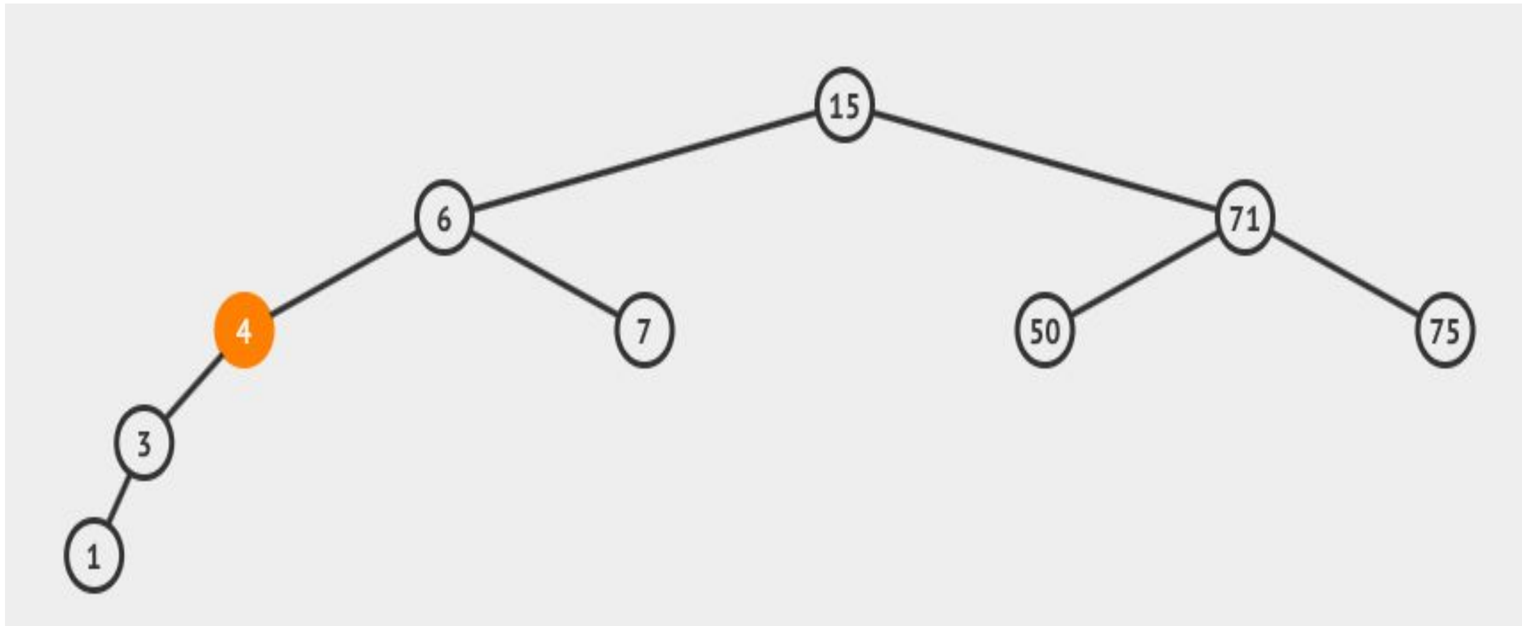
Height balanced BSTs (AVL)

- Simple right rotation example:



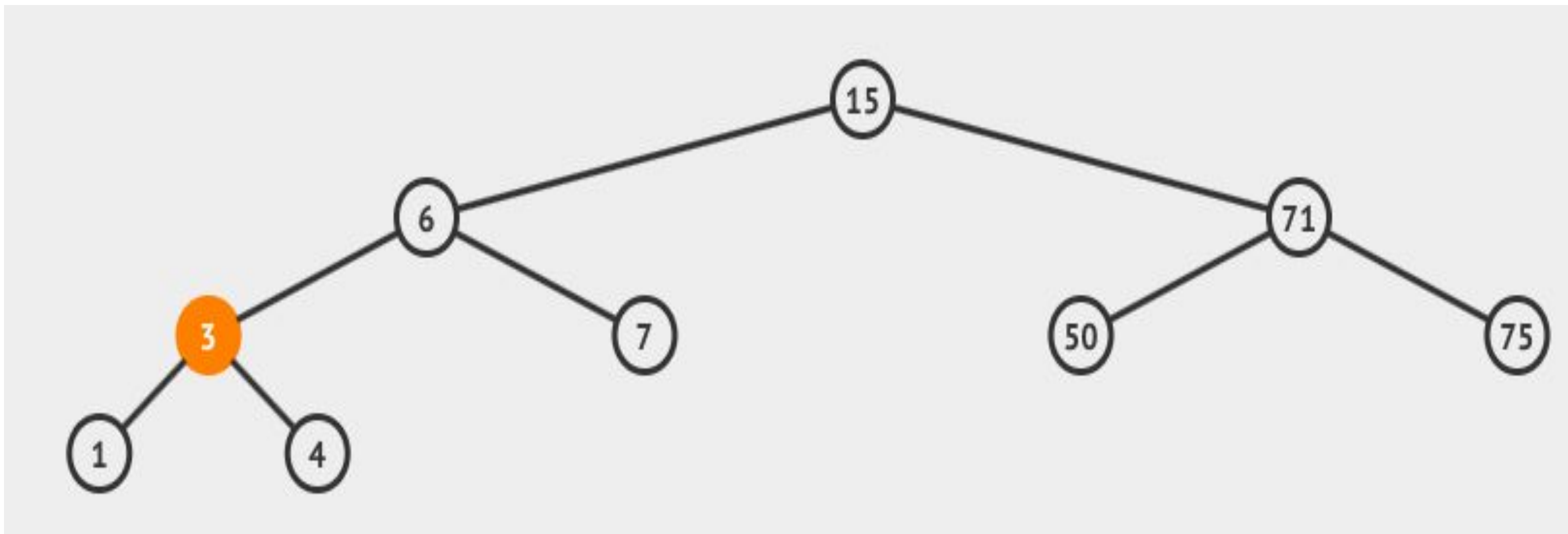
Height balanced BSTs (AVL)

- Simple right rotation example:



Height balanced BSTs (AVL)

- Simple right rotation example:

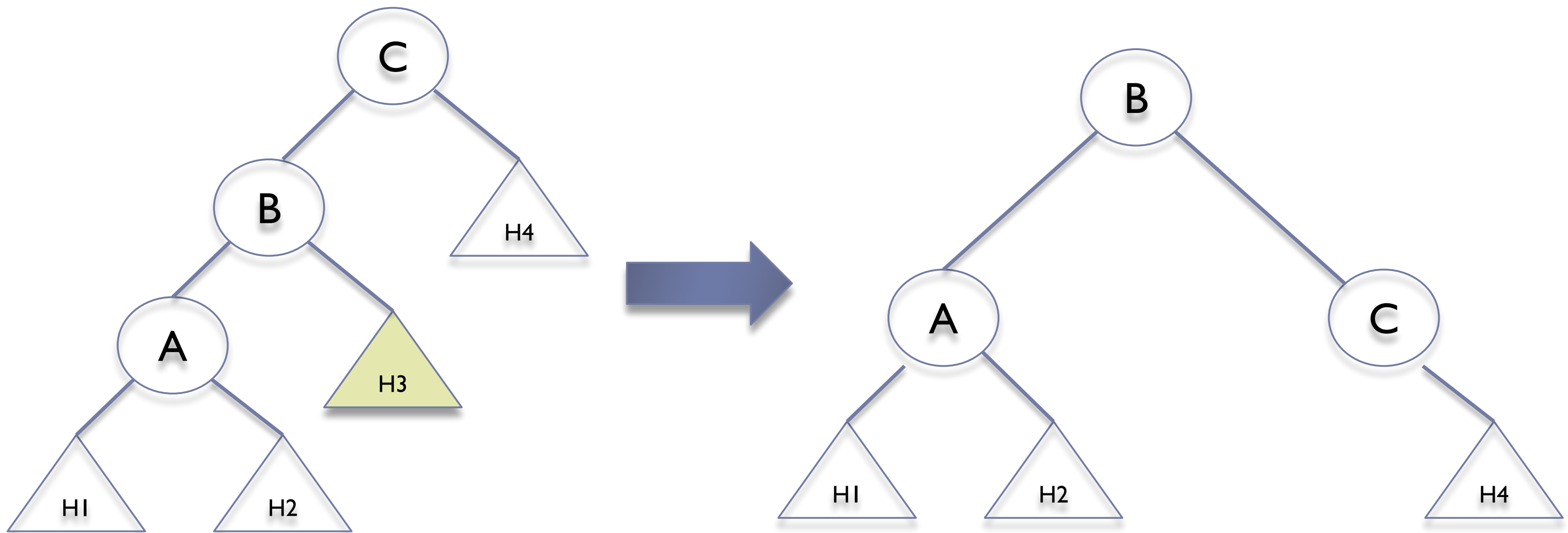


http://visualgo.net/bst.html

Height balanced BSTs (AVL)

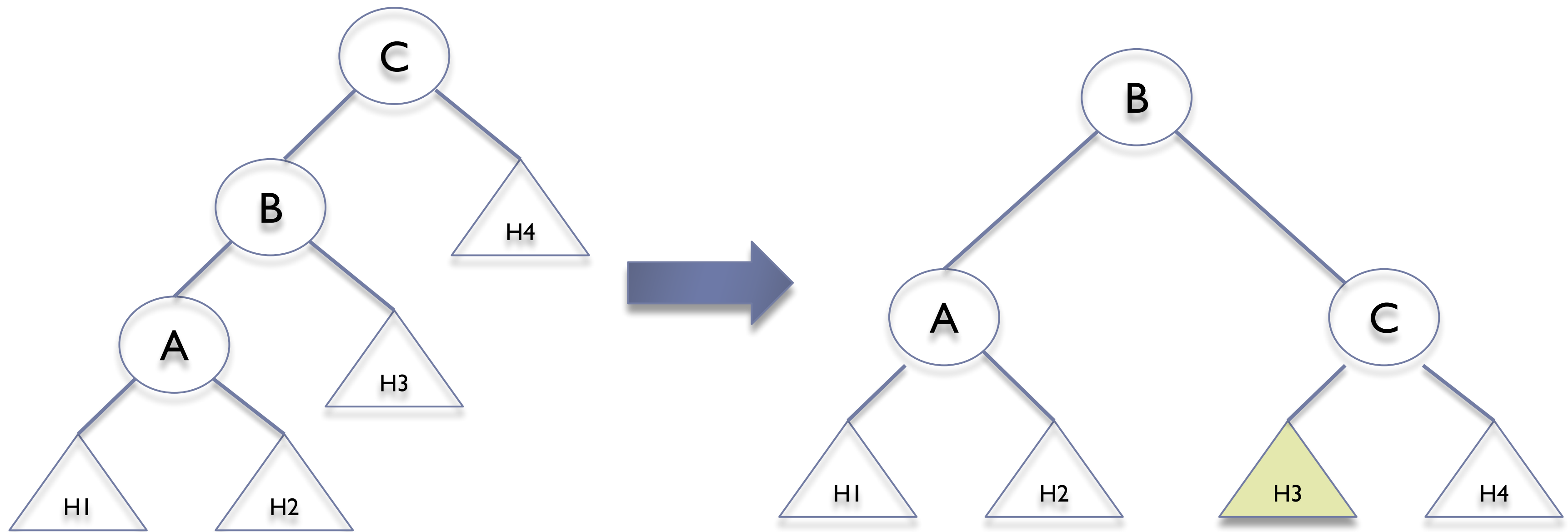
- Simple right rotation:

Where do we put the right subtree of B?



Height balanced BSTs (AVL)

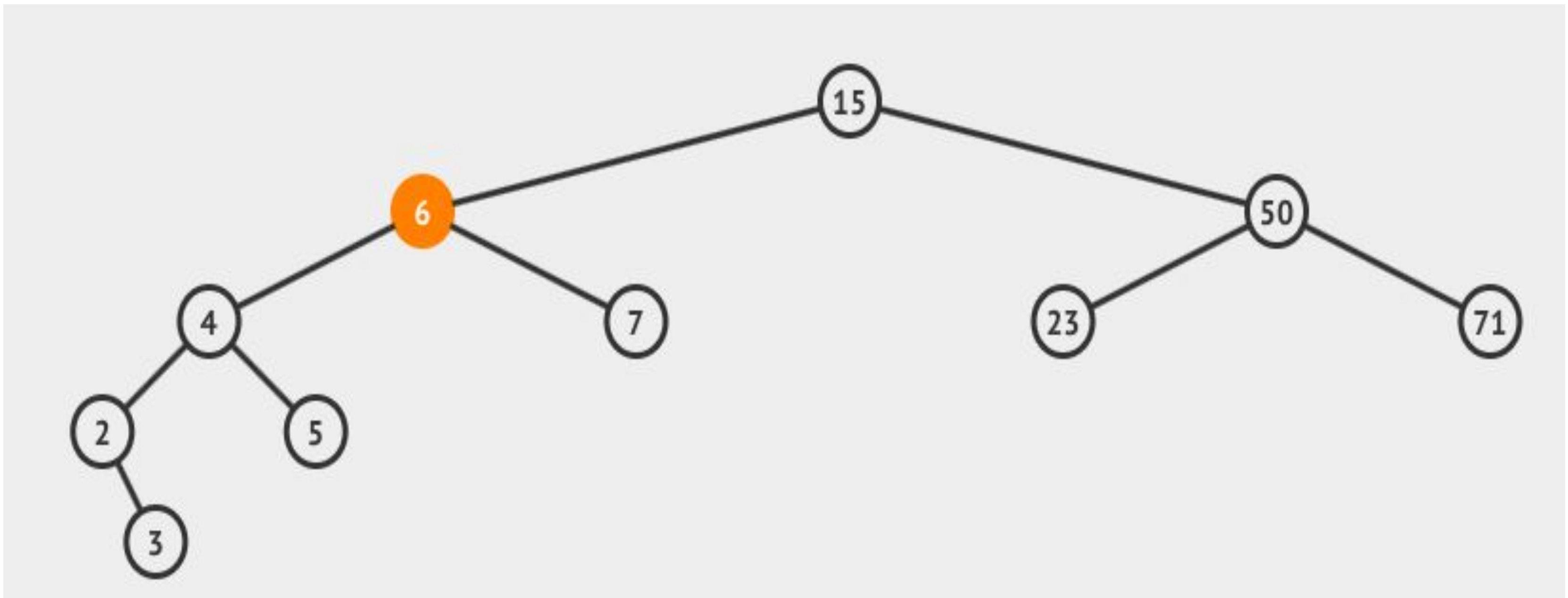
- Simple right rotation:



InOrder: $H_1AH_2BH_3CH_4$ = InOrder: $H_1AH_2BH_3CH_4$

Height balanced BSTs (AVL)

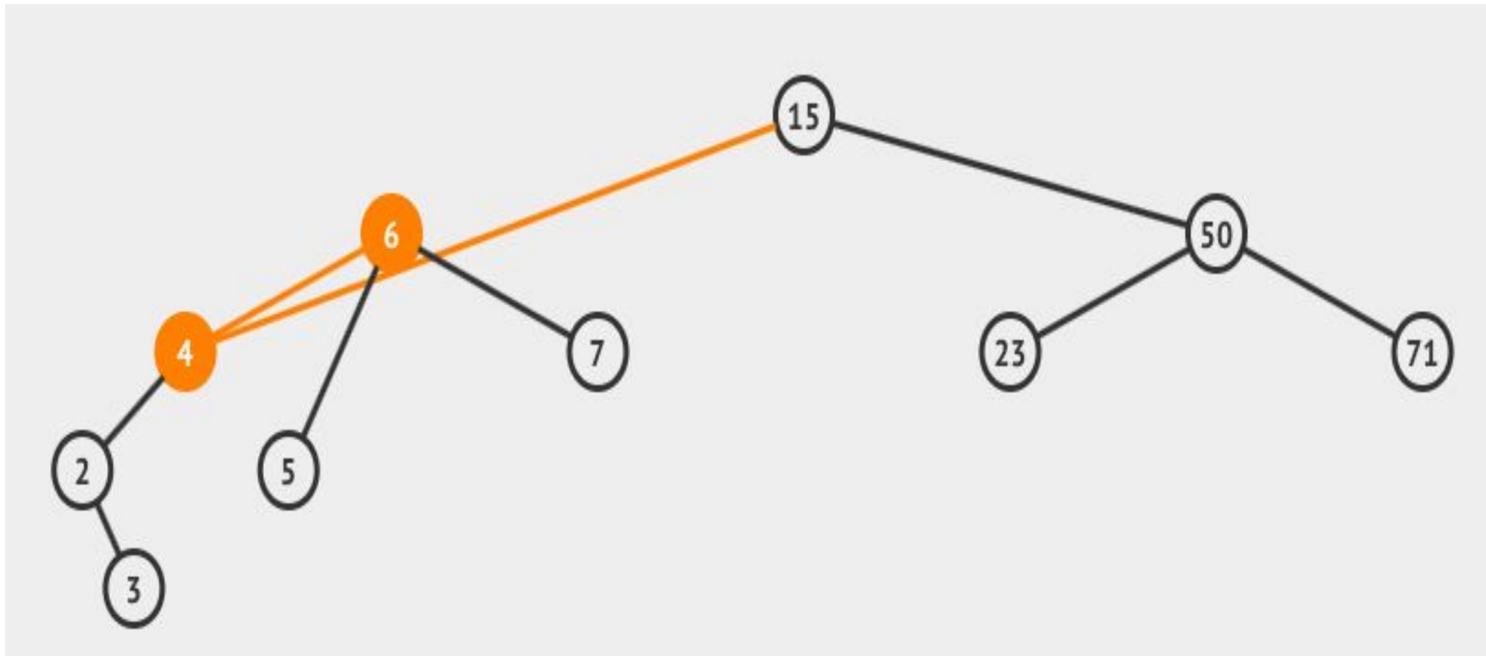
Example (with subtrees) for simple right rotation:



- The first unbalanced node is 6
- A simple right rotation can be applied: **move 4 as the root for the subtree and 6 as its right child**
- Note that node 4 has a right subtree, what do we do with node 5?

Height balanced BSTs (AVL)

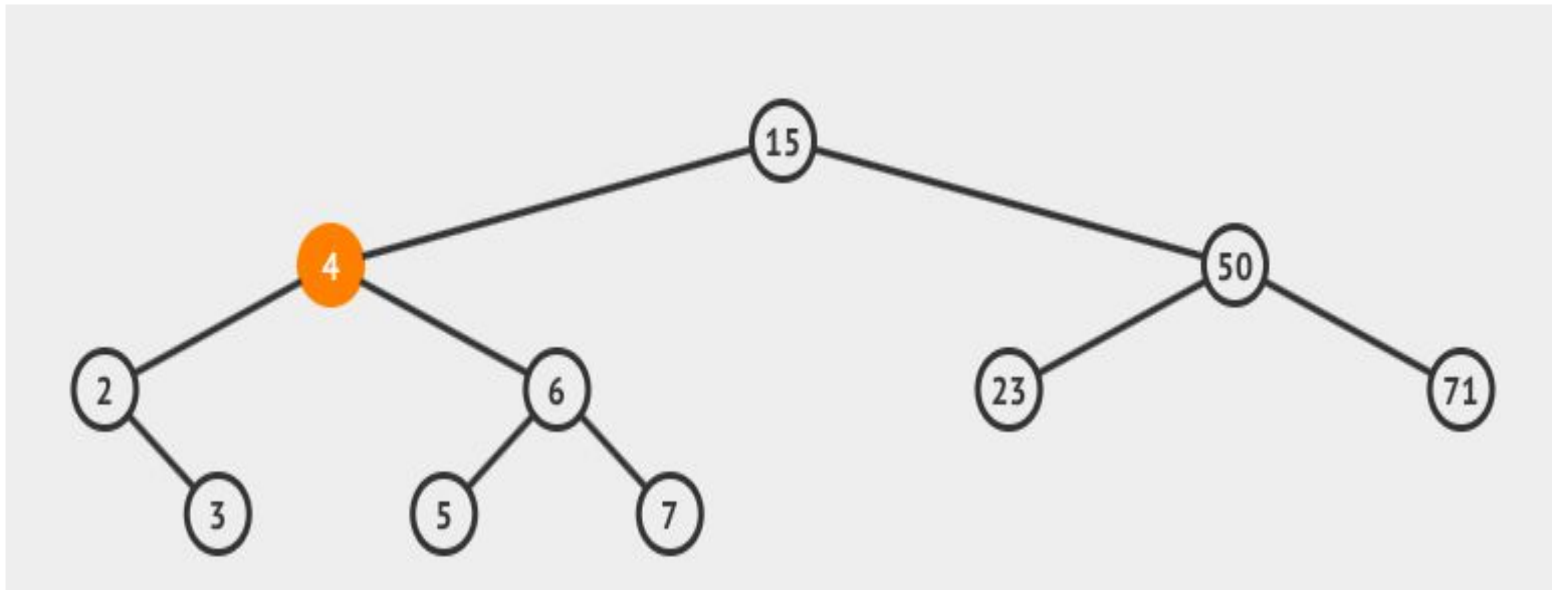
Example (with subtrees) for simple right rotation:



- Node 6 is rotated as right child of node 4, and it is the new root for the subtree
- **The former right subtree for node 4, that is node 5, must be the new left subtree of node 6**

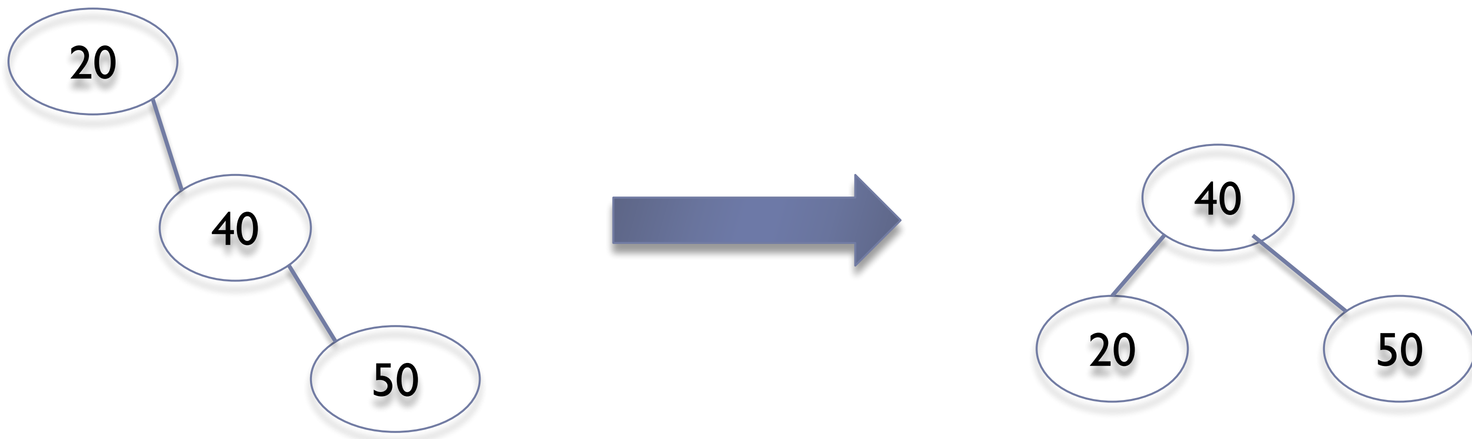
Height balanced BSTs (AVL)

Example (with subtrees) for simple right rotation:



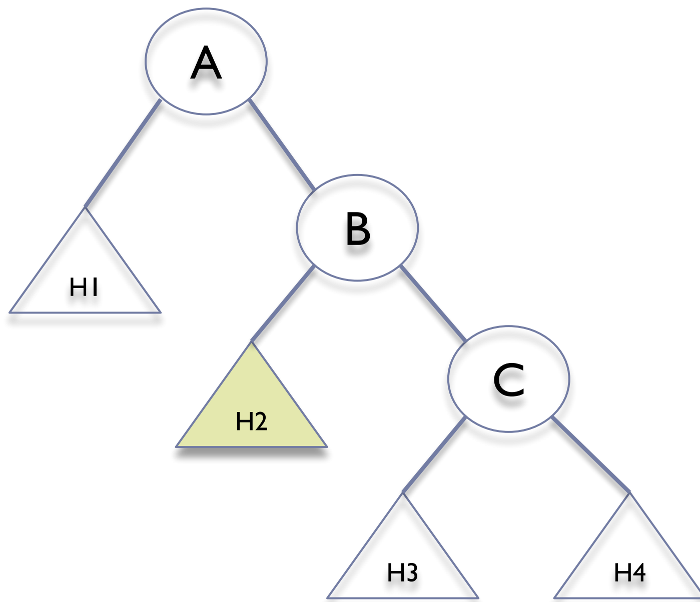
Height balanced BSTs (AVL)

Simple left rotation example:

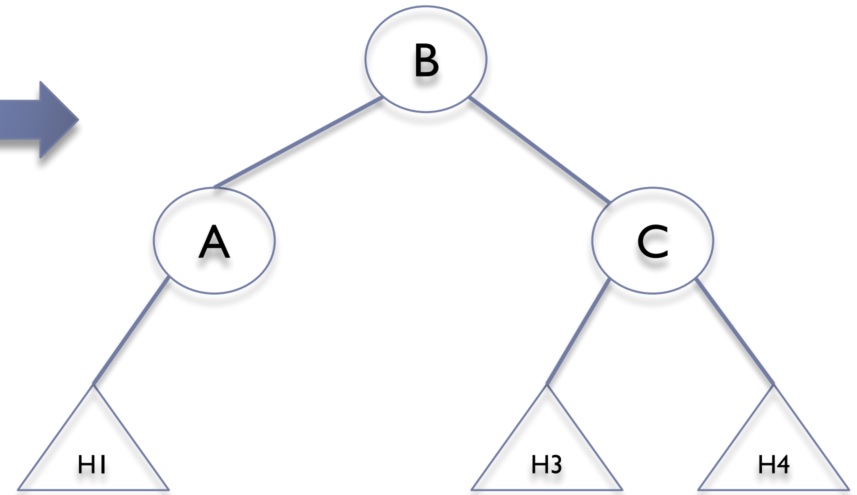


Height balanced BSTs (AVL)

Simple left rotation:

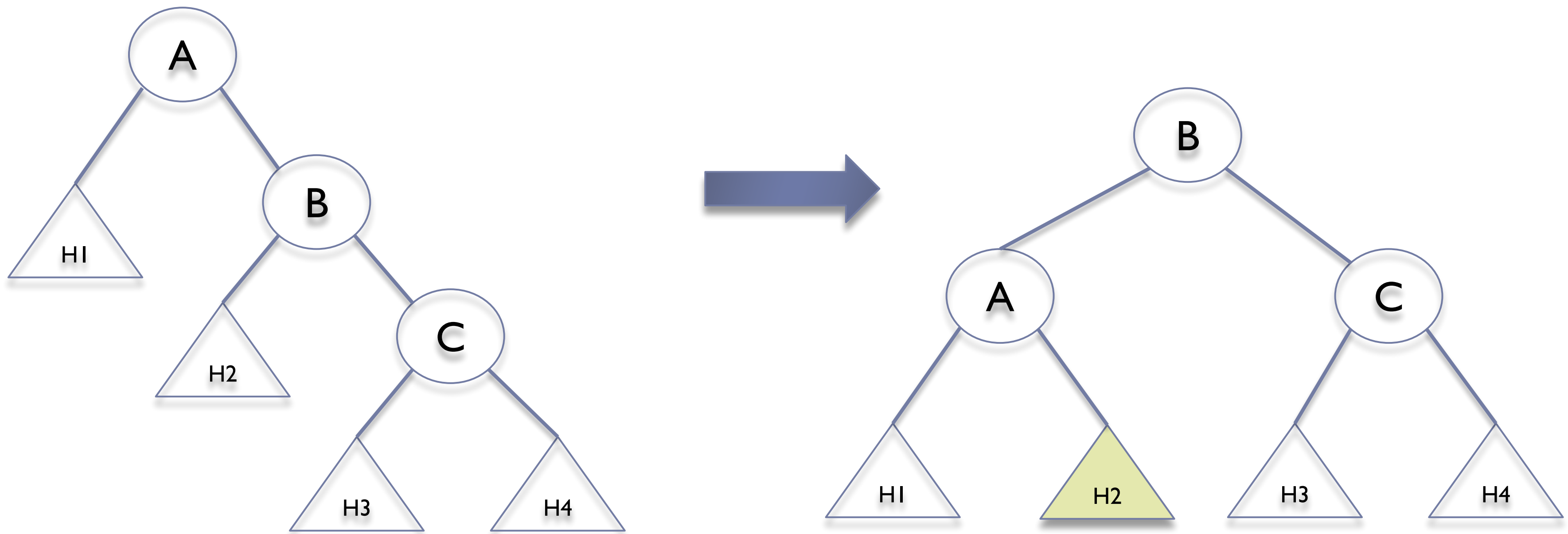


Where do we put the left subtree of B node?



Height balanced BSTs (AVL)

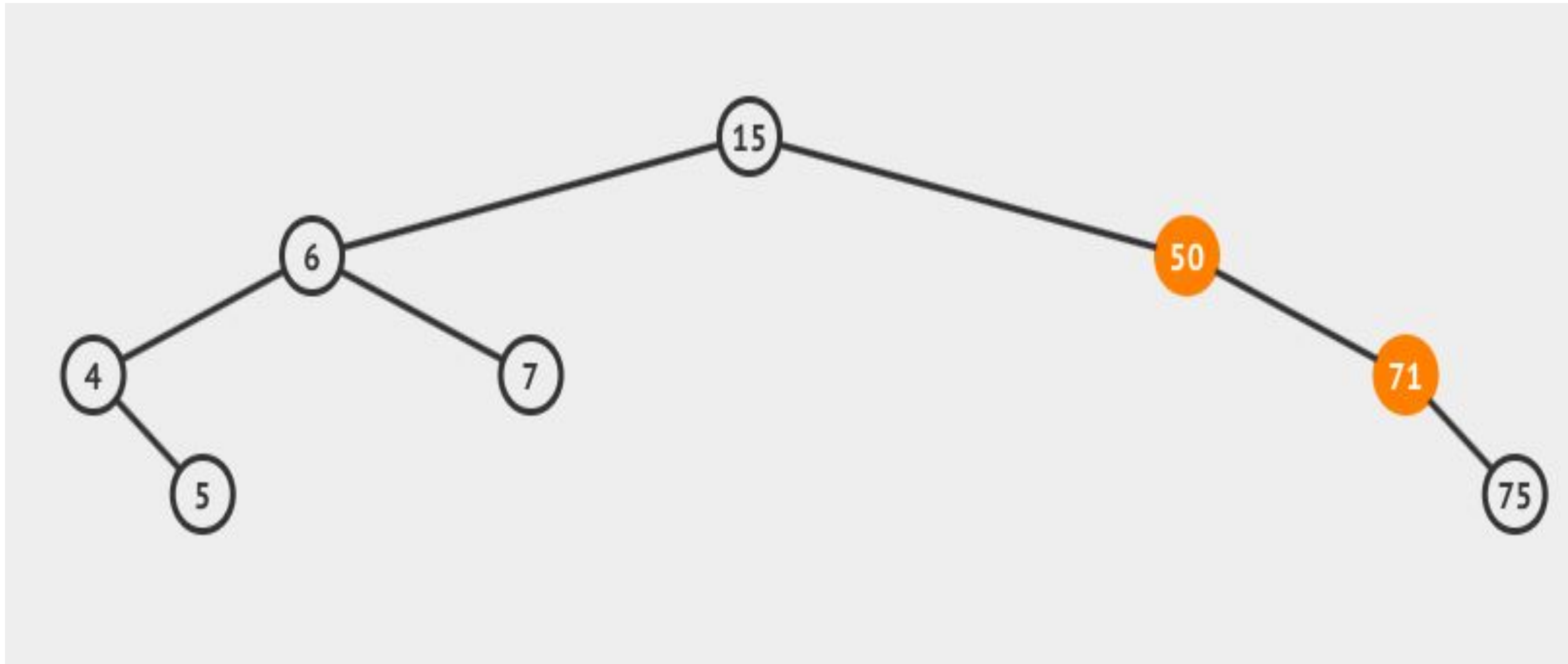
Simple left rotation:



InOrder: $H_1AH_2BH_3CH_4$ = InOrder: $H_1AH_2BH_3CH_4$

Height balanced BSTs (AVL)

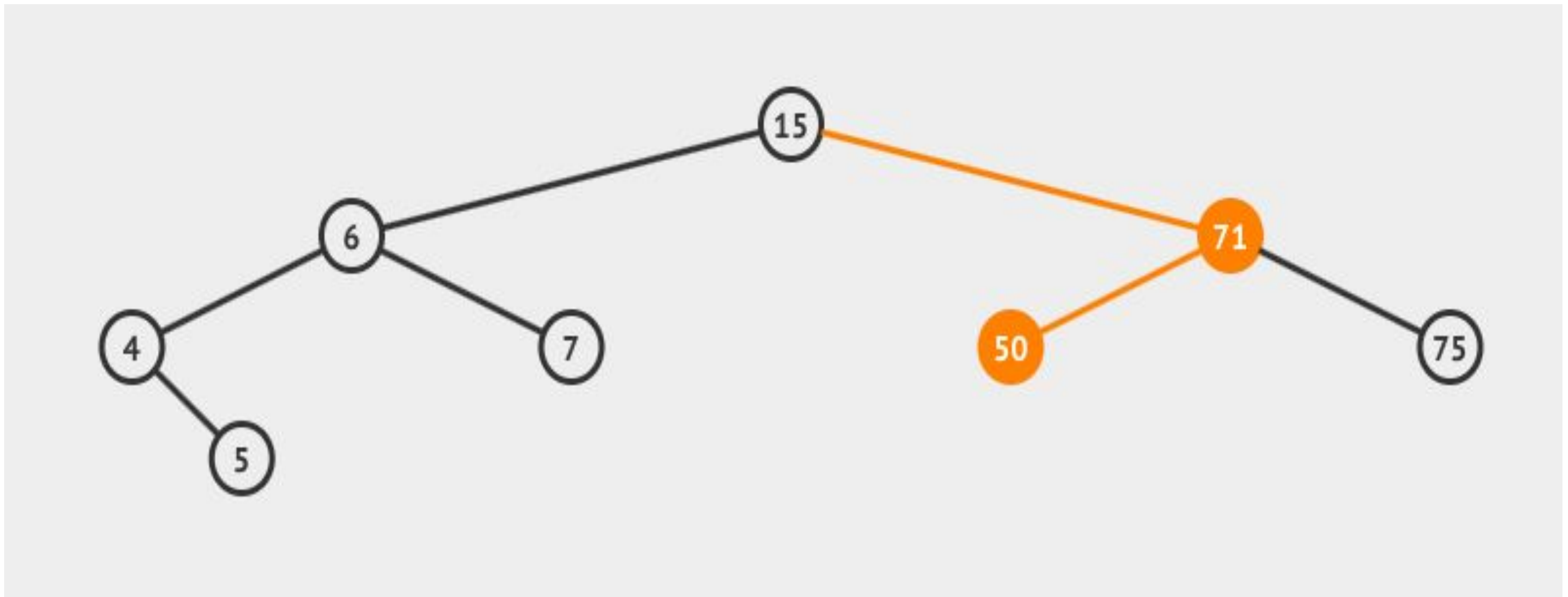
Simple left rotation example:



http://visualgo.net/bst.html

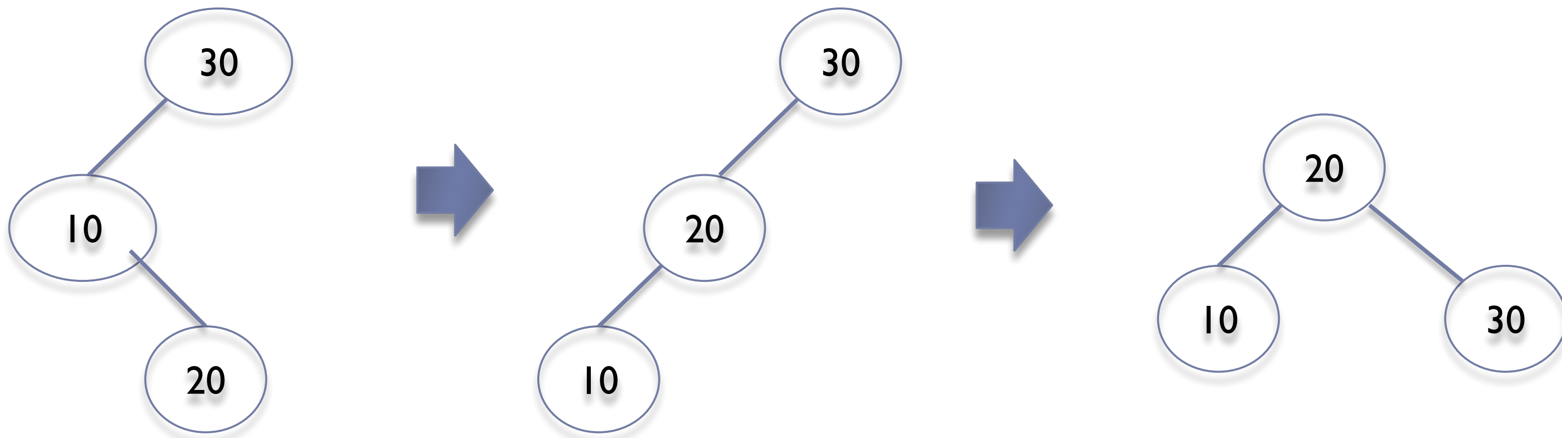
Height balanced BSTs (AVL)

Simple left rotation example:



Height balanced BSTs (AVL)

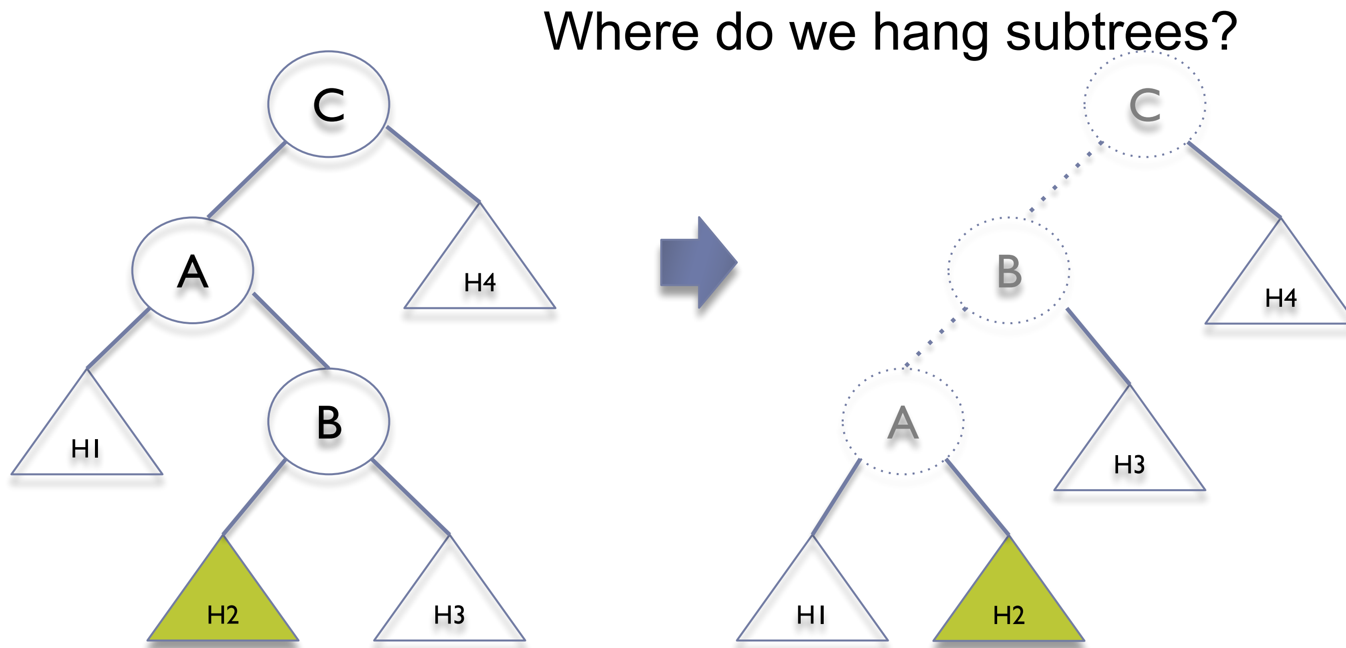
Double rotation example (left-right):



Height balanced BSTs (AVL)

Double rotation (left-right) :

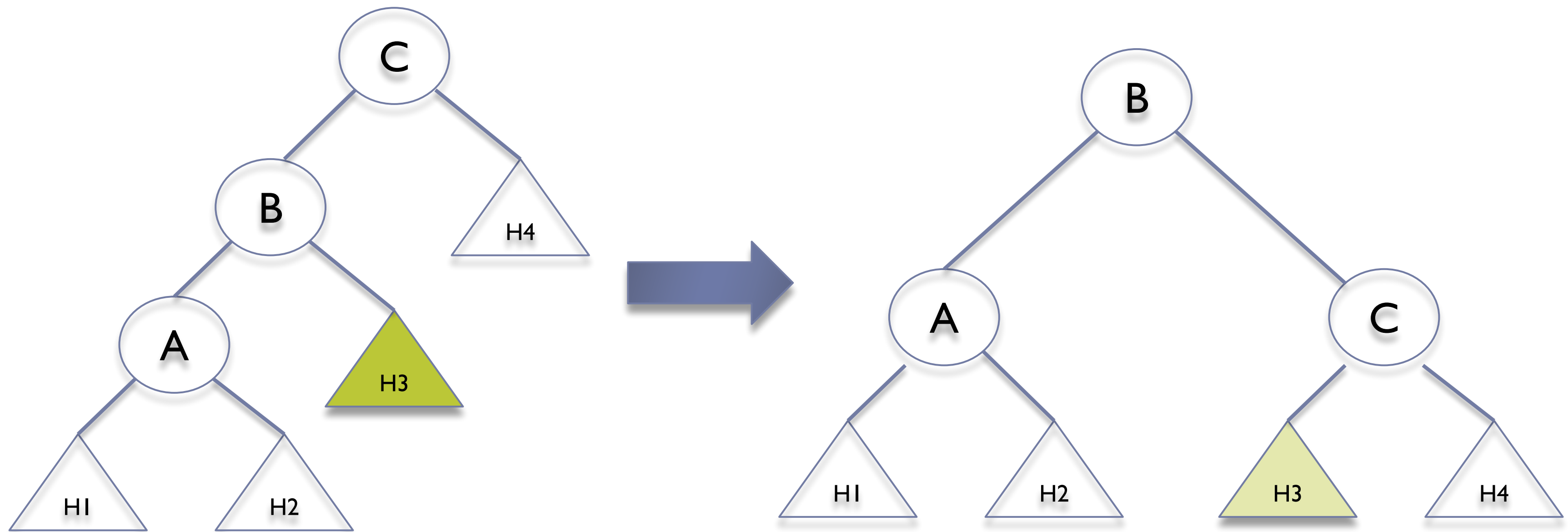
Where do we hang subtrees?



First step: left rotation of node B (as a left child of C)

Height balanced BSTs (AVL)

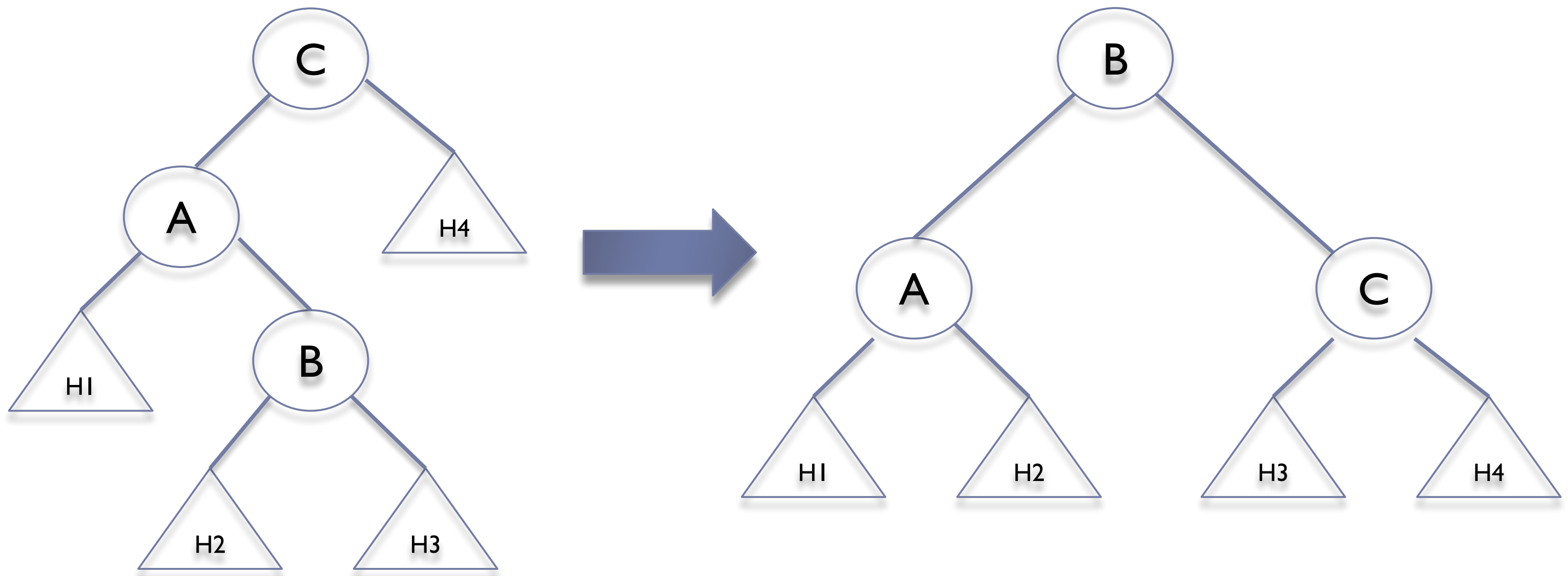
Double rotation (left-right):



Second rotation: move C node as right child of B

Height balanced BSTs (AVL)

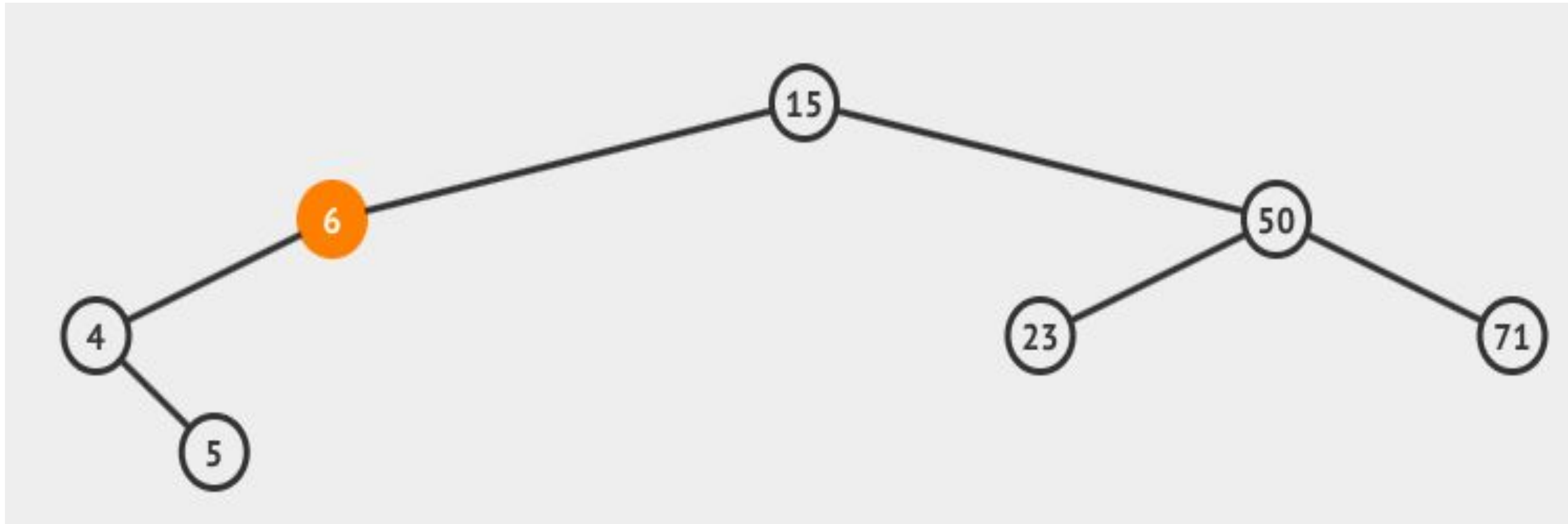
Double rotation (left-right):



InOrder: $H_1AH_2BH_3CH_4$ = InOrder: $H_1AH_2BH_3CH_4$

Height balanced BSTs (AVL)

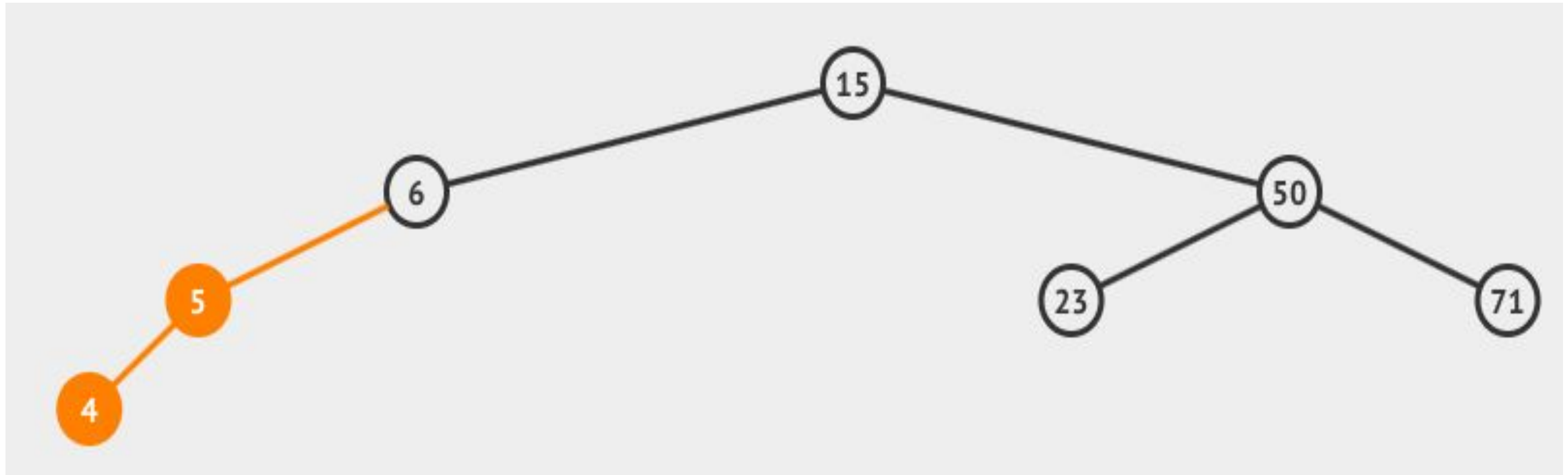
Double rotation example (left-right):



- Node 6 is unbalanced. For a left-right balancing two rotations are needed:
- First rotation, left: node 5 is the new left child of node 6 and 4 is the new left child of 5.
- Second rotation, right: rotate node 6 to right and node 5 is the new root for the subtree.

Height balanced BSTs (AVL)

Double rotation example (left-right):



- Node 6 is unbalanced. For a left-right balancing two rotations are needed:
- **First rotation, left: node 5 is the new left child of node 6 and 4 is the new left child of 5.**
- Second rotation, right: rotate node 6 to right and node 5 is the new root for the subtree.

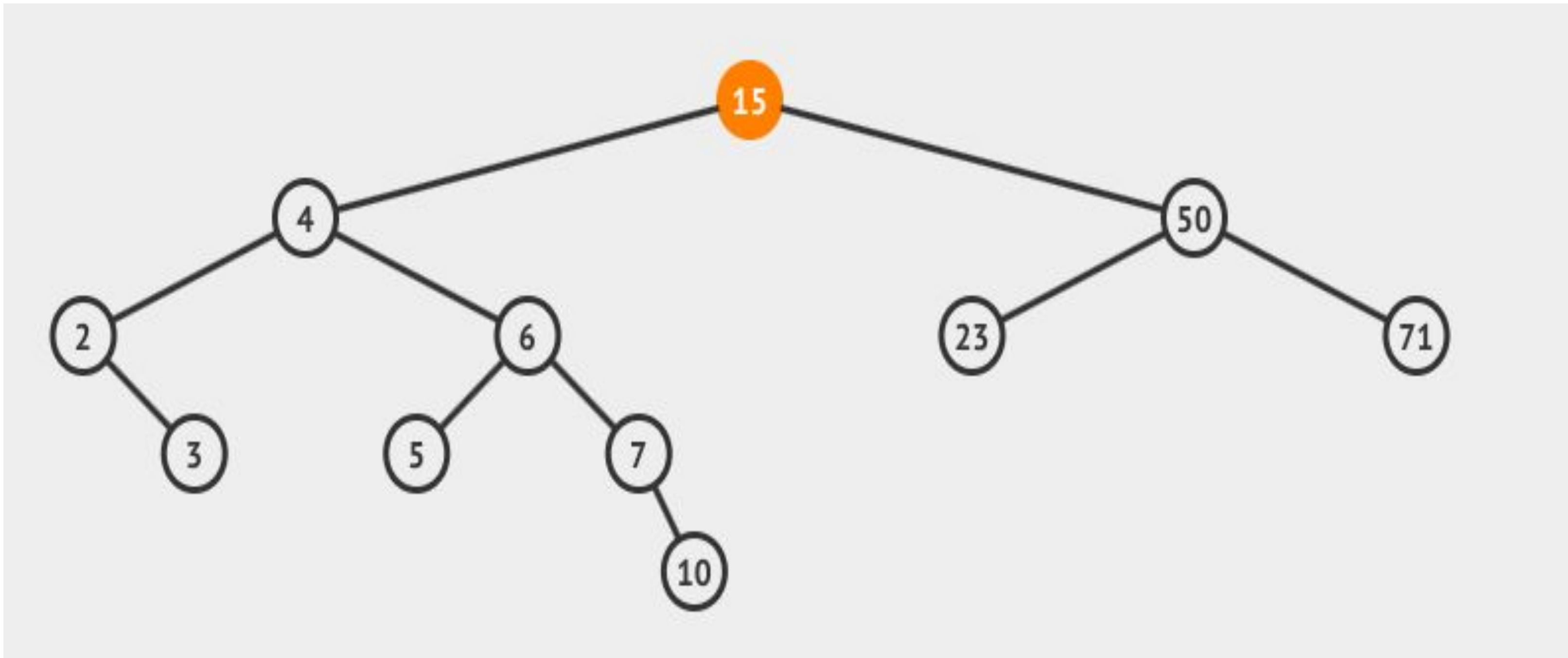
Height balanced BSTs (AVL)

Double rotation example (left-right):

- Node 6 is unbalanced. For a left-right balancing two rotations are needed:
- First rotation, left: node 5 is the new left child of node 6 and 4 is the new left child of 5.
- **Second rotation, right: rotate node 6 to right and node 5 is the new root for the subtree.**

Height balanced BSTs (AVL)

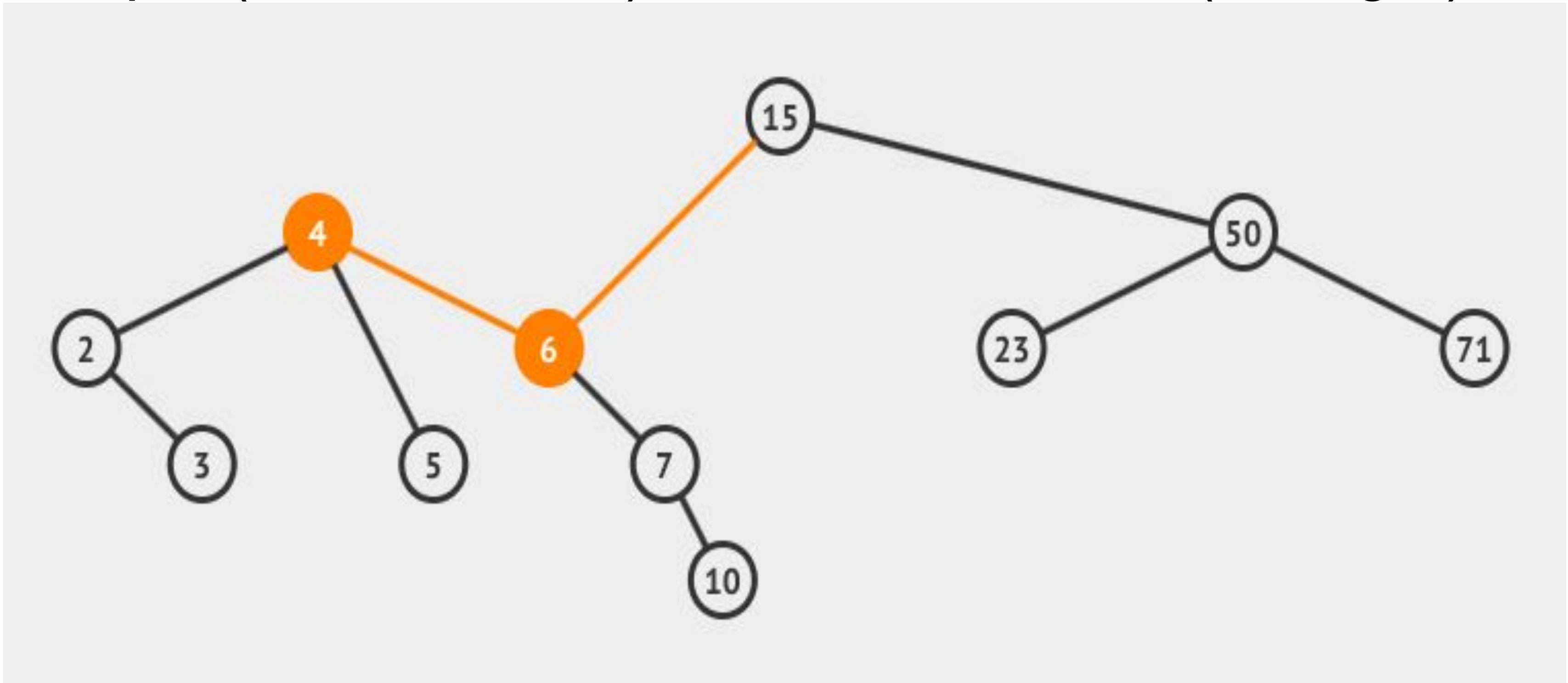
Example (with subtrees) for double rotation (left-right):



- Node 15 is unbalanced (fe=2). Left-right rotation can be applied
- First rotation moves 6 as the left child of node 15
- Second rotation will put 15 as right subtree of 6 node (and this is the new root)

Height balanced BSTs (AVL)

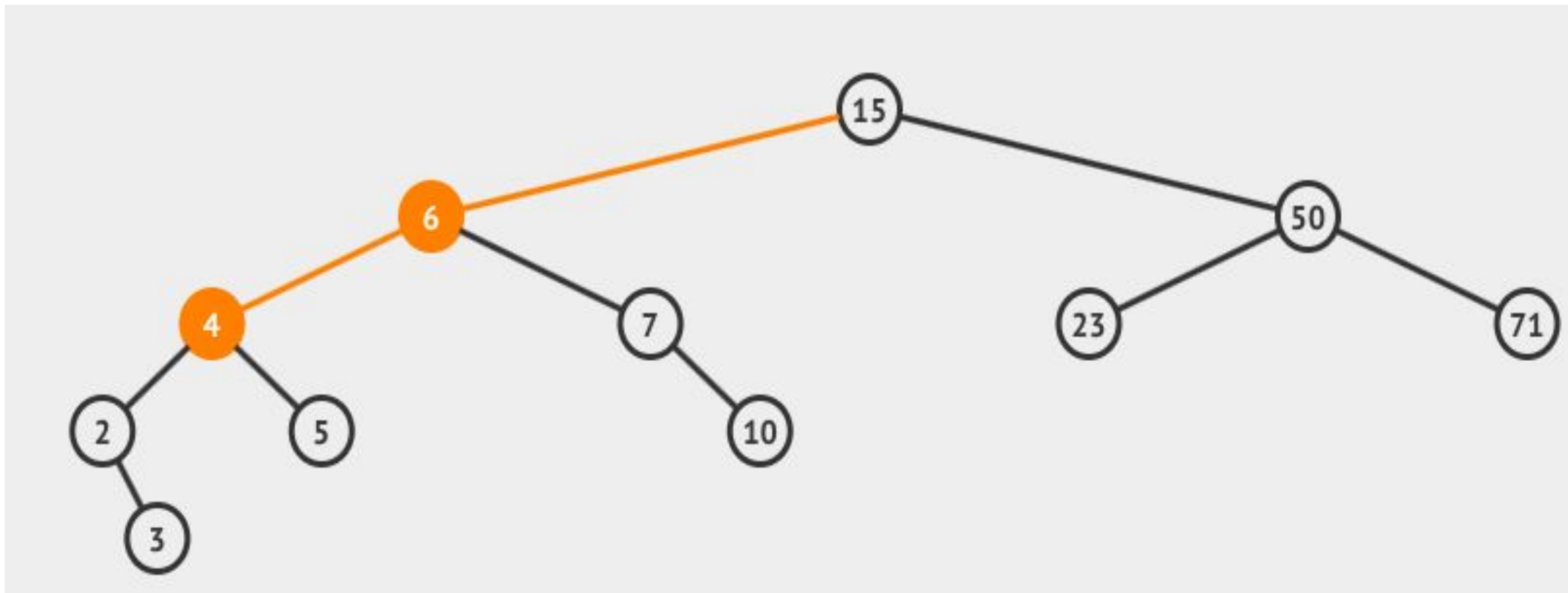
Example (with subtrees) for double rotation (left-right):



- Node 15 is unbalanced (fe=2). Left-right rotation can be applied
- **First rotation moves 6 as the left child of node 15**
- Second rotation will put 15 as right subtree of 6 node (and this is the new root)

Height balanced BSTs (AVL)

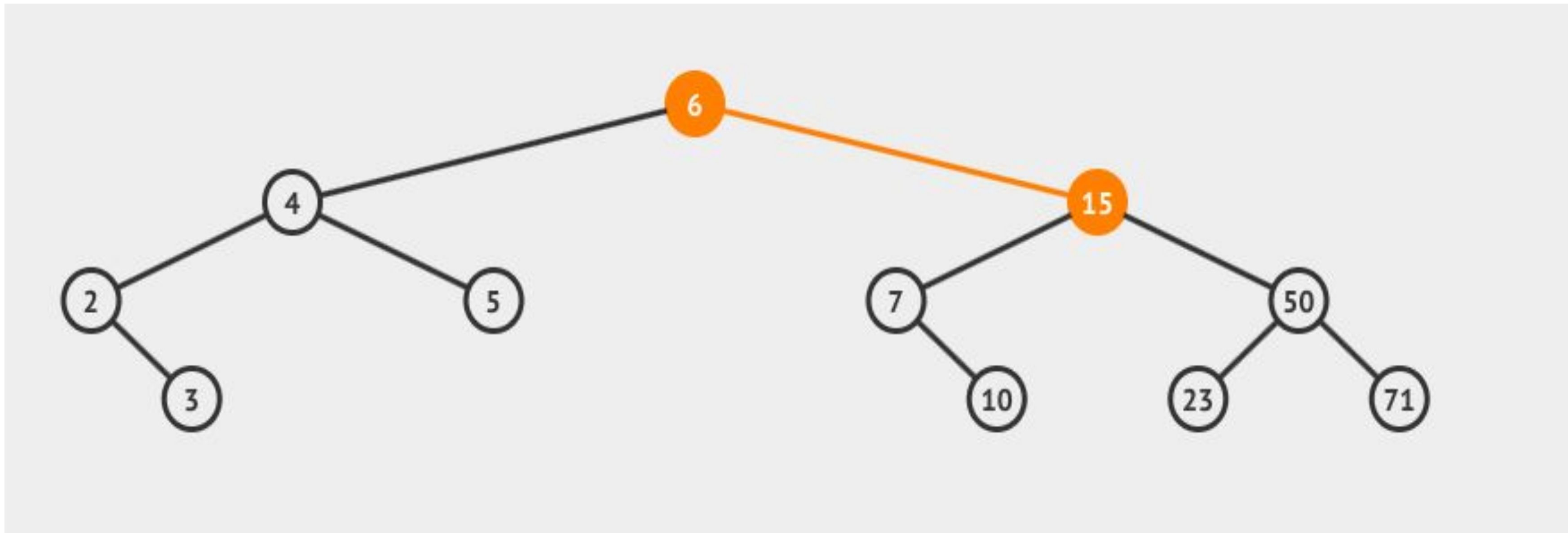
Example (with subtrees) for double rotation (left-right):



- Node 15 is unbalanced (fe=2). Left-right rotation can be applied
- **First rotation moves 6 as the left child of node 15**
- **Second rotation will put 15 as right subtree of 6 node (and this is the new root)**

Height balanced BSTs (AVL)

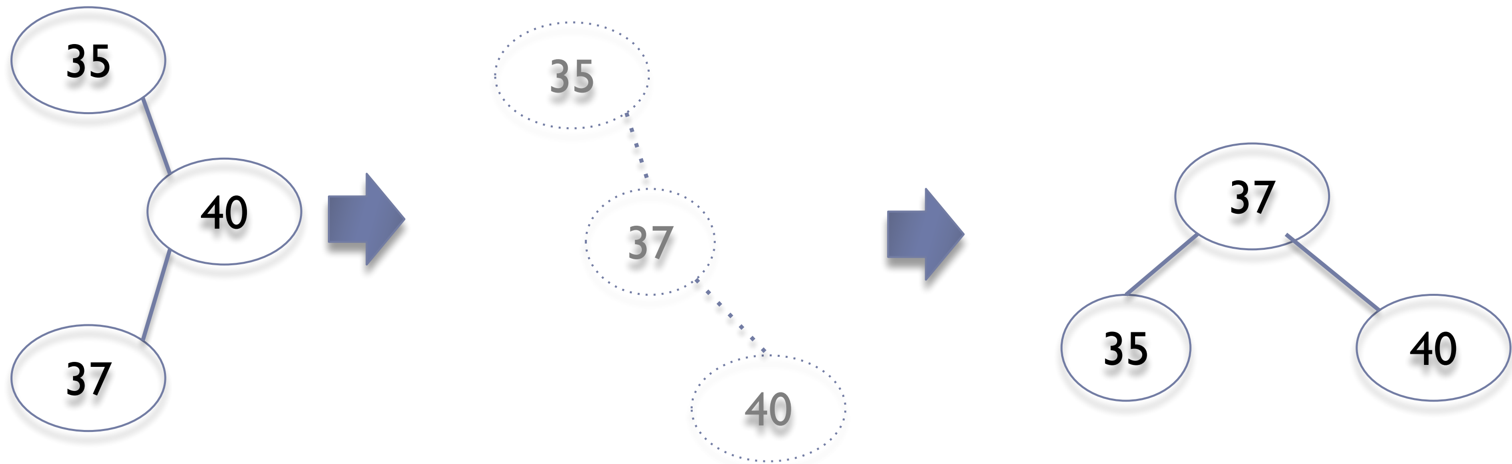
Example (with subtrees) for double rotation (left-right):



- Node 15 is unbalanced (fe=2). Left-right rotation can be applied
- First rotation moves 6 as the left child of node 15
- **Second rotation will put 15 as right subtree of 6 node (and this is the new root)**

Height balanced BSTs (AVL)

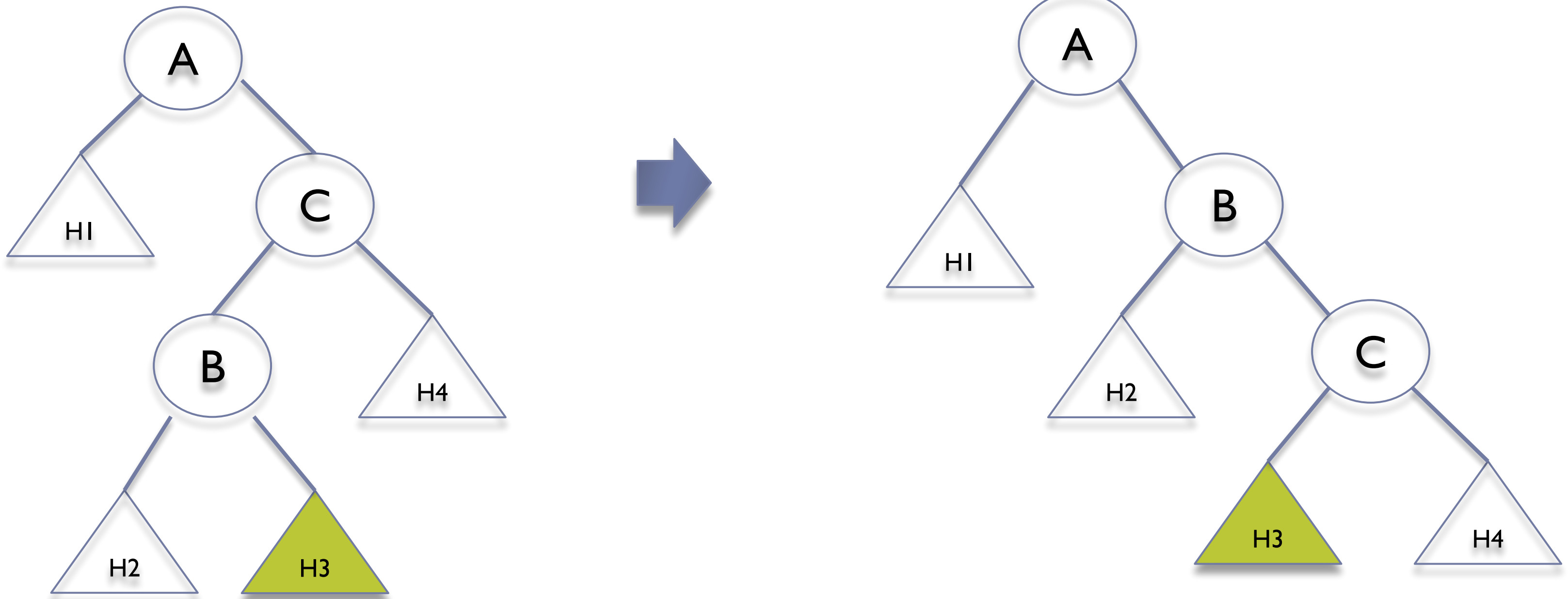
Double rotation example (right-left):



Height balanced BSTs (AVL)

Double rotation (right-left):

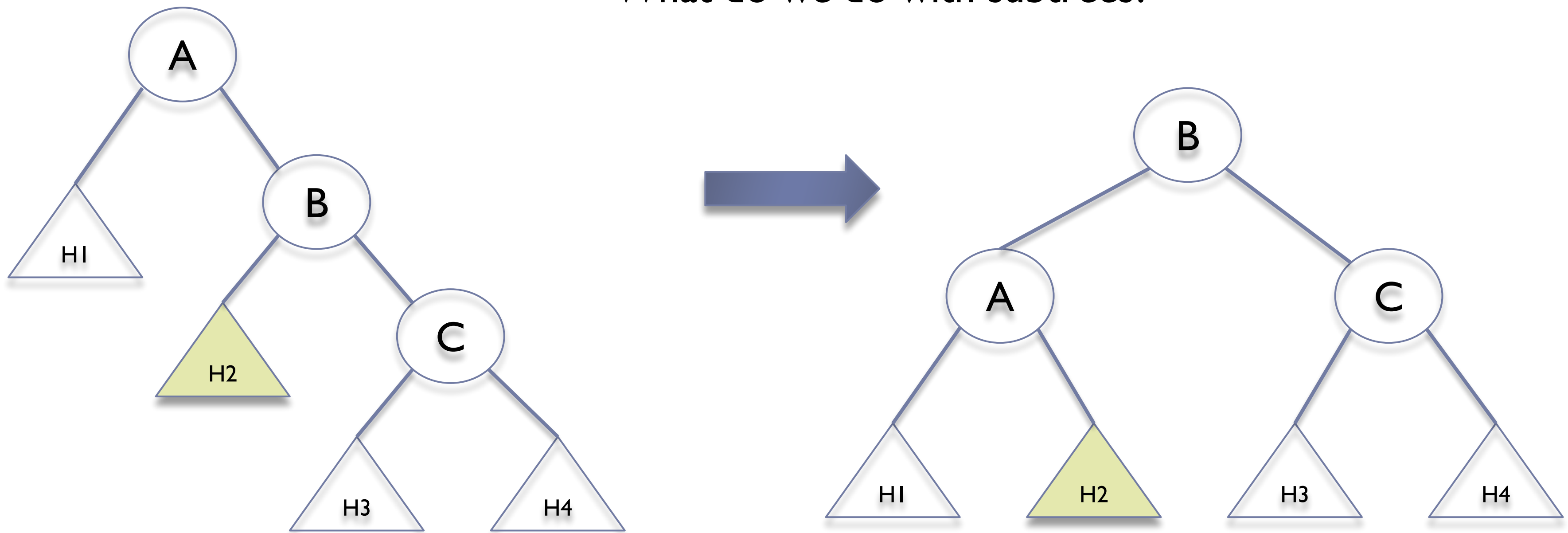
What do we do with subtrees??



First rotation: B node as right child
fo A node

Height balanced BSTs (AVL)

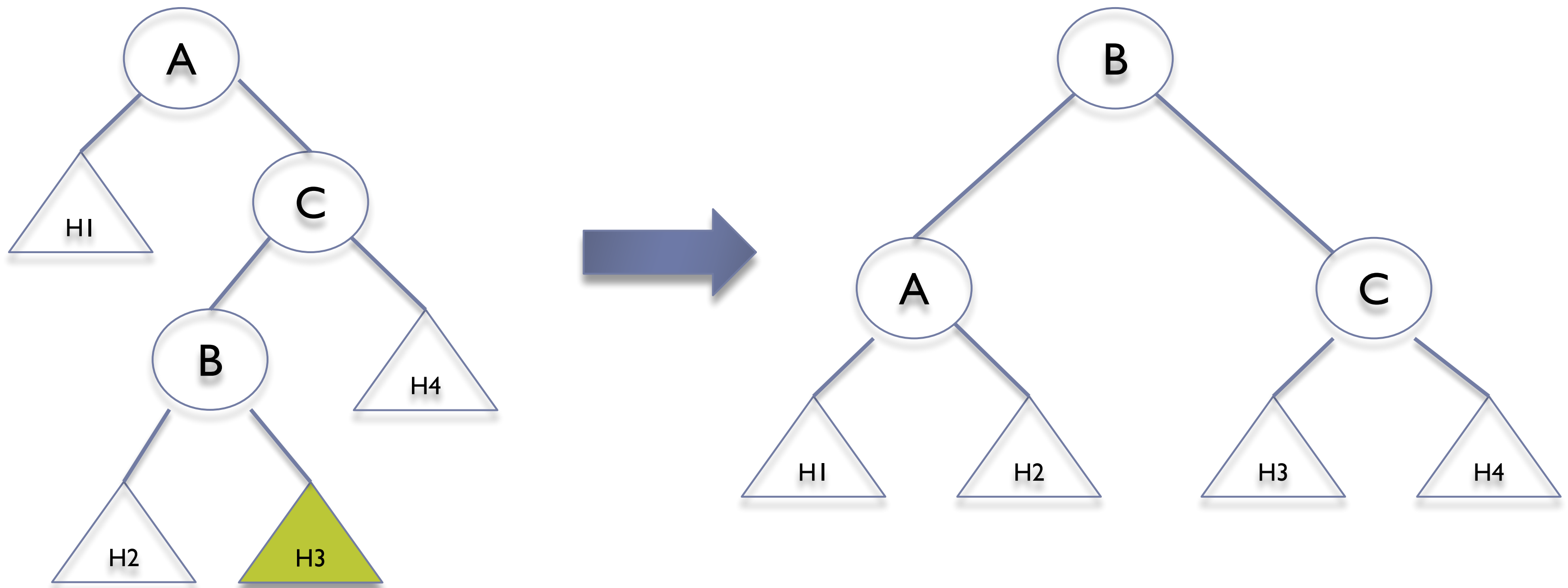
Double rotation (right-left):



Second rotation: A node as left child of B node

Height balanced BSTs (AVL)

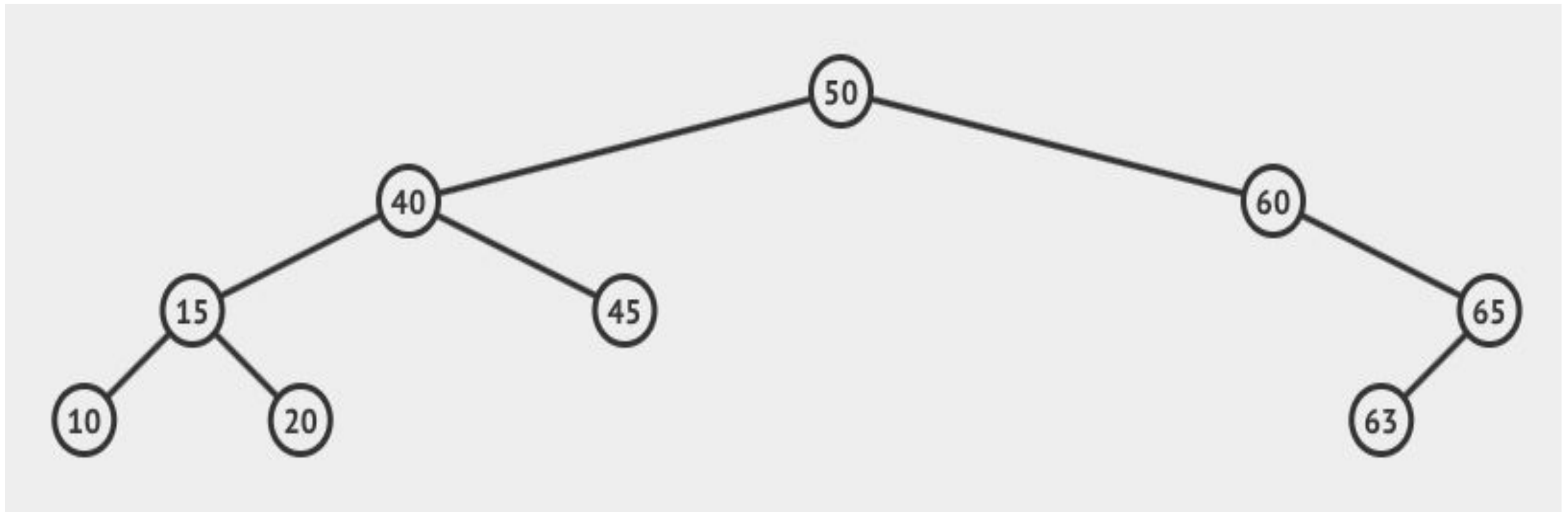
Double rotation (right-left):



InOrder: $H_1AH_2BH_3CH_4$ = InOrder: $H_1AH_2BH_3CH_4$

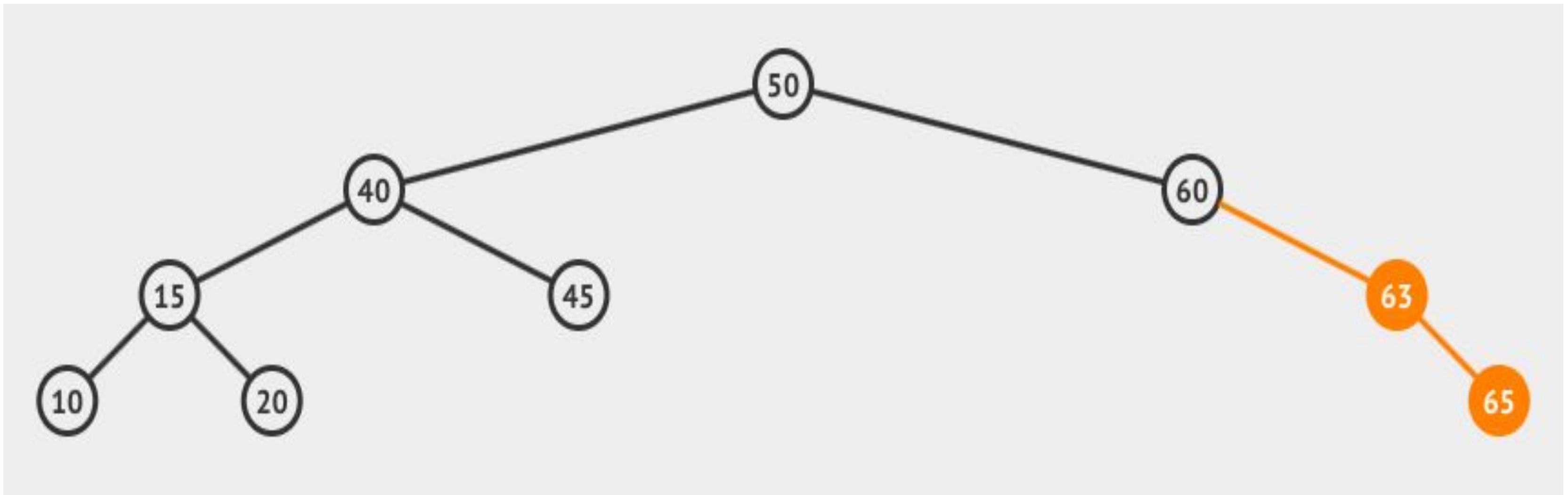
Height balanced BSTs (AVL)

Double Rotation example (right-left):



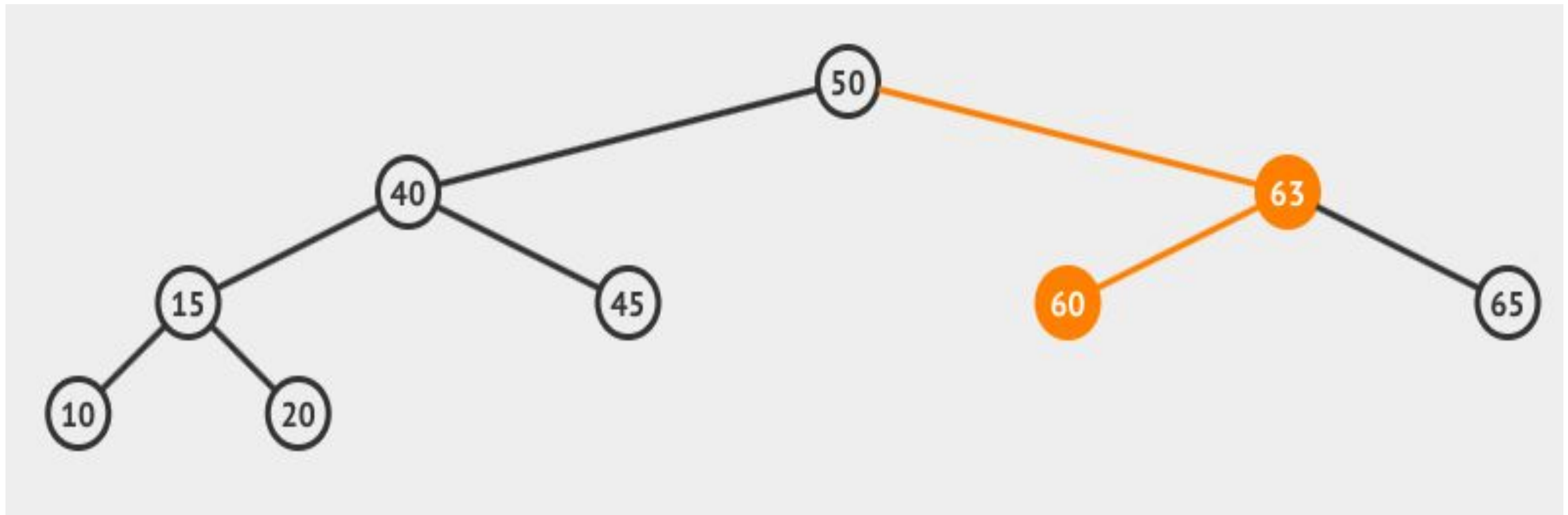
Height balanced BSTs (AVL)

Double Rotation example (right-left):



Height balanced BSTs (AVL)

Double Rotation example (right-left):



Height balanced BSTs (AVL)

If you can choose different rotations:

1. Choose the rotation from the longest branch.
2. Choose the simplest rotation.

Height balanced BSTs (AVL)

- **Advantage**: Rebalancing is made from down, only in the path from the inserted or removed node to the root
  - So, rebalancing is O(log n)
- **Disadvantage**: The tree does not get so compacted as in size balanced BSTs. Even so, search complexity is also O(log n)