Algorithms and Data Structures 2

13 – Trees

Dr Michele Sevegnani

School of Computing Science
University of Glasgow

michele.sevegnani@glasgow.ac.uk

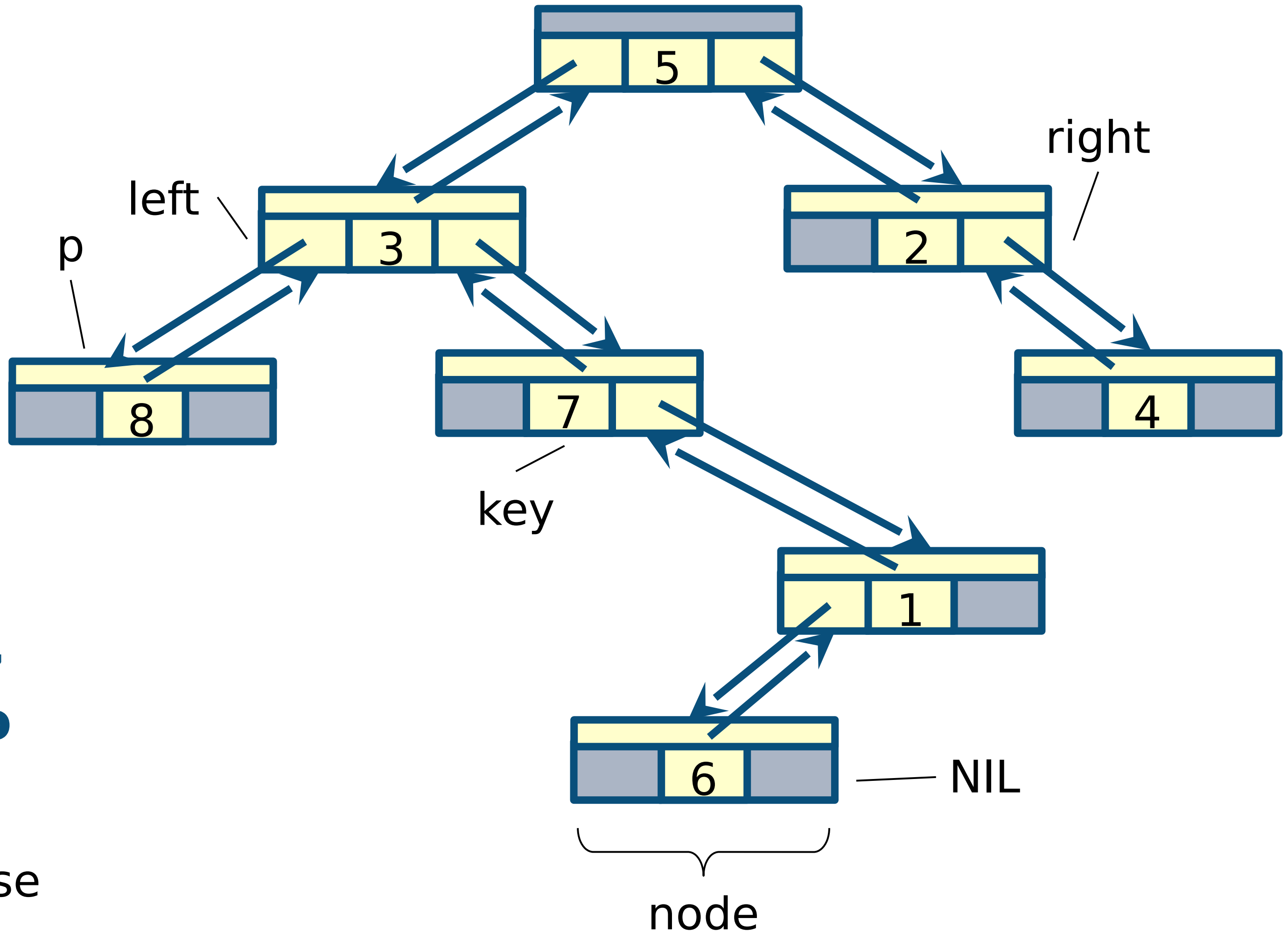
Outline

- **Binary trees**
  - Properties
  - Traversals
- **Rooted trees with unbounded branching**
- **Other tree representations**

Binary trees

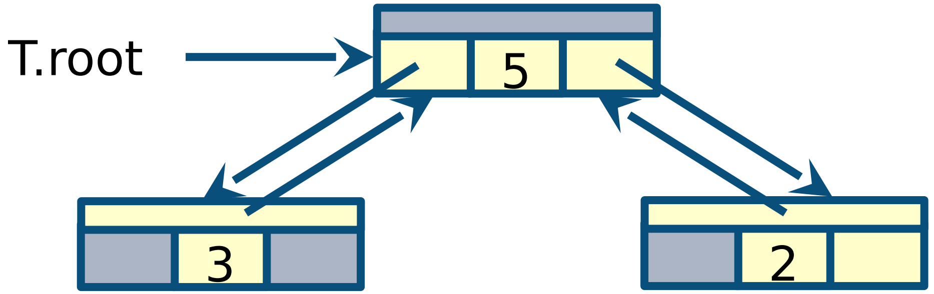
- **A binary tree is a linked data structure in which each node x has an attribute key and three pointer attributes**
  - x.p points to its parent
  - x.left points to its left child
  - x.right points to its right child

- **If a child or the parent is missing, the appropriate attribute is set to NIL**
  - The root of the three is the only node whose parent is NIL
  - Node x is a leaf when x.left = x.right = NIL



Root pointer

- An attribute **T.root** points to the root element of tree **T**



- If **T.root = NIL**, the tree is empty

Properties

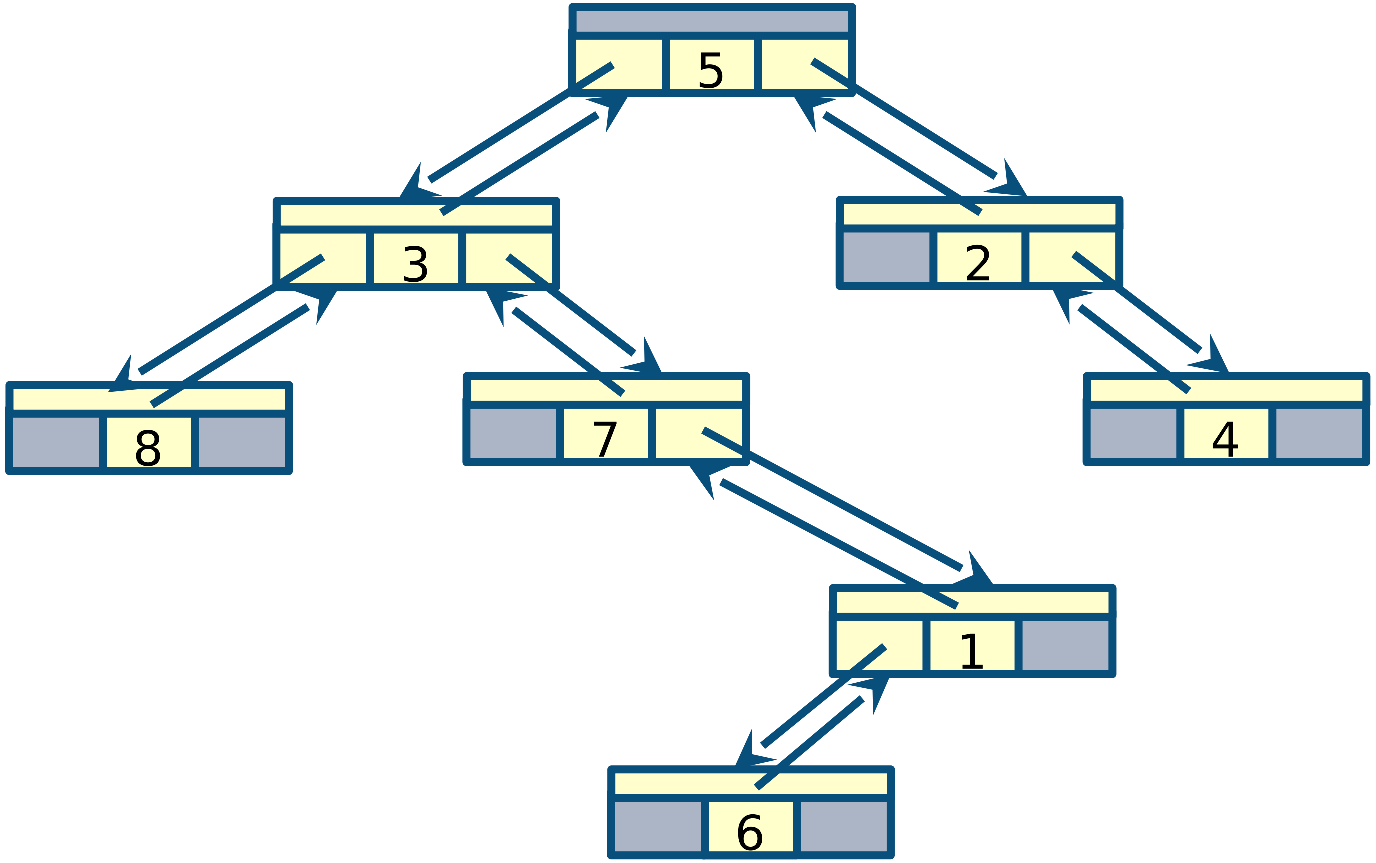
- **A binary tree can be defined recursively**
  - Empty
  - A single node r (the root) whose left and right subtrees are binary trees

- **For any two nodes n_1 and n_k, a path from n_1 to n_k is a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_i+1 for $1 \leq i < k$**
  - For any path, the length is the number of edges on the path

- **The height (h) of a tree is the length of the longest path from the root**

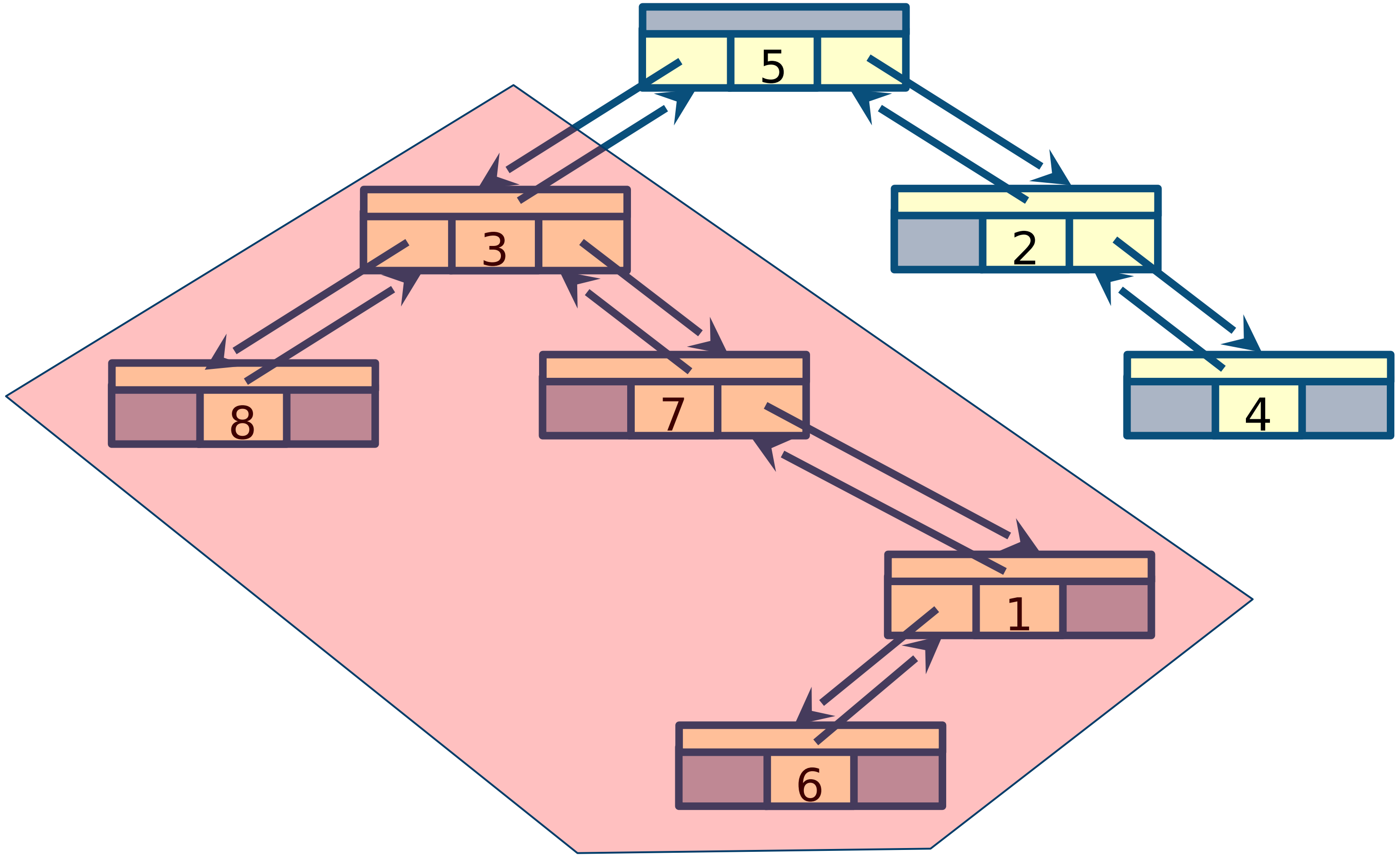
Example

- **Left subtree**



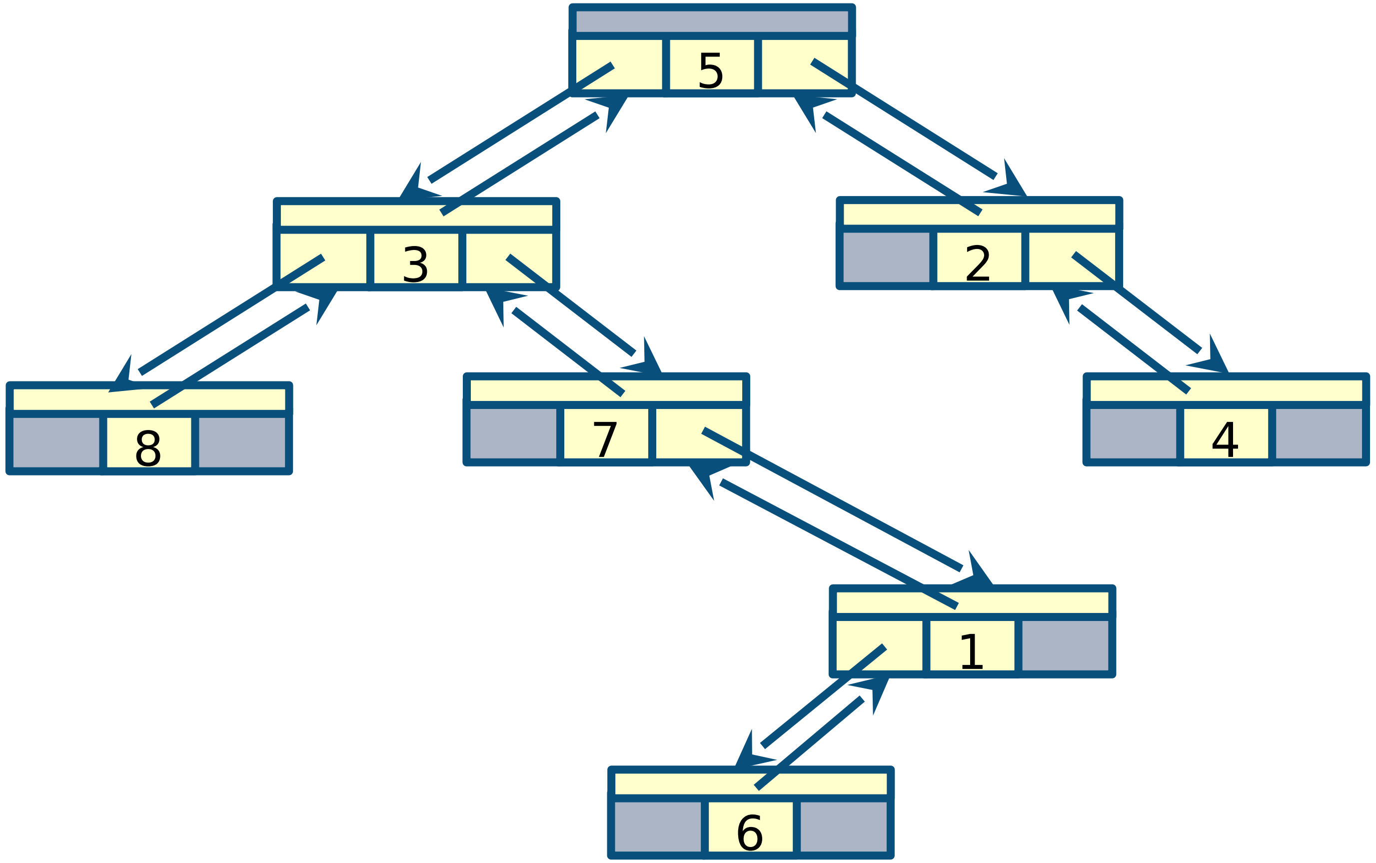
Example

- **Left subtree**



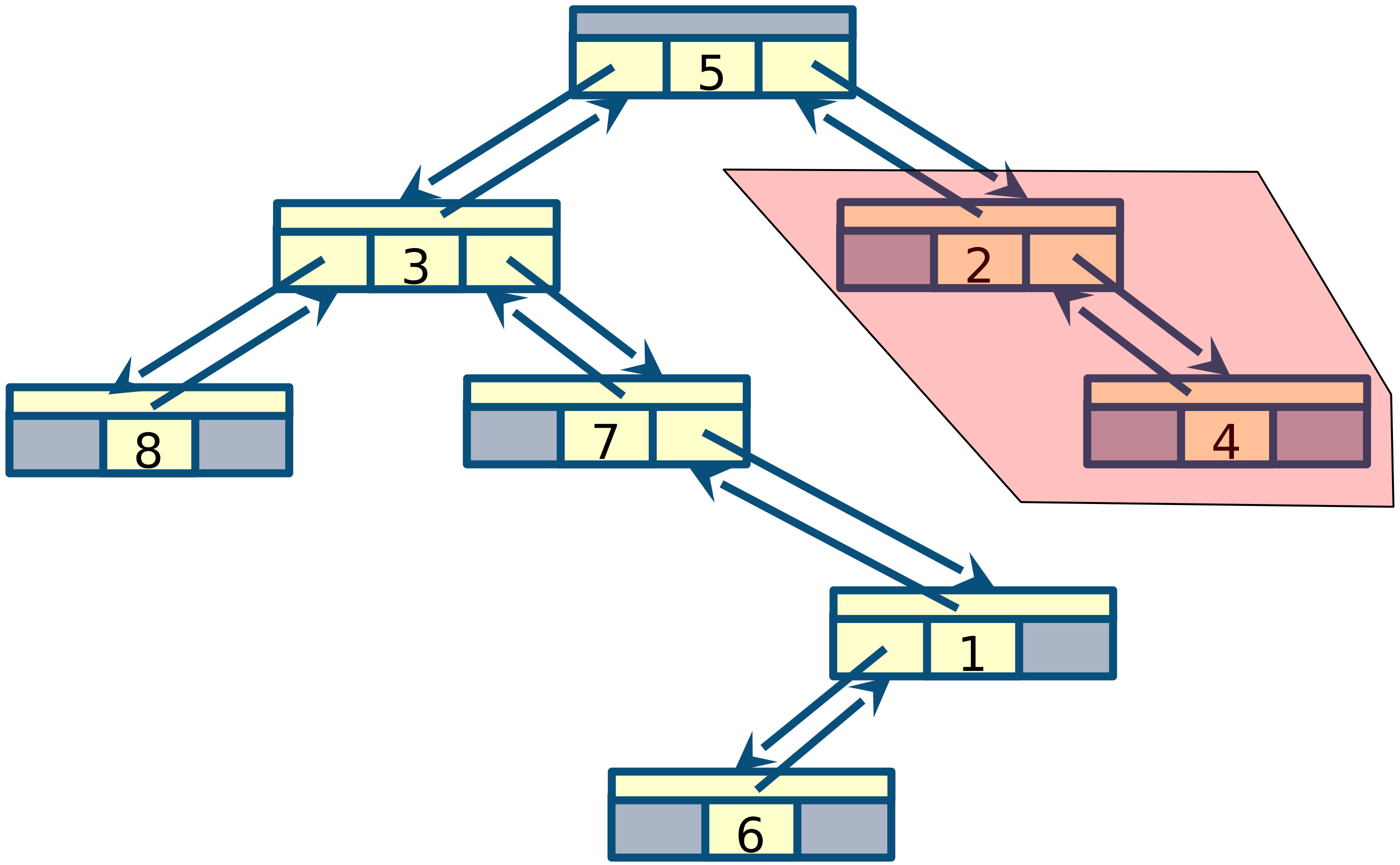
Example

- **Right subtree**



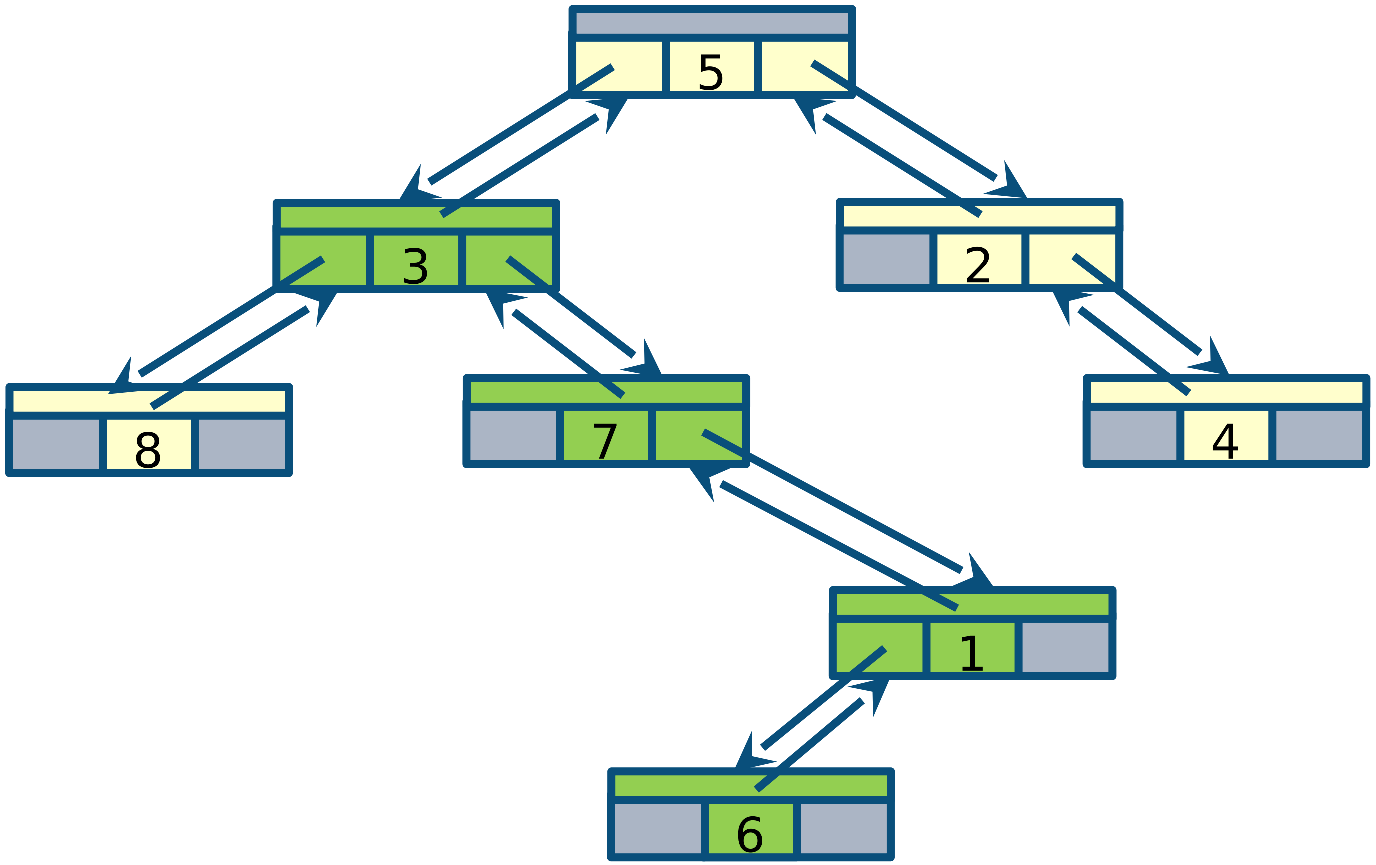
Example

- **Right subtree**



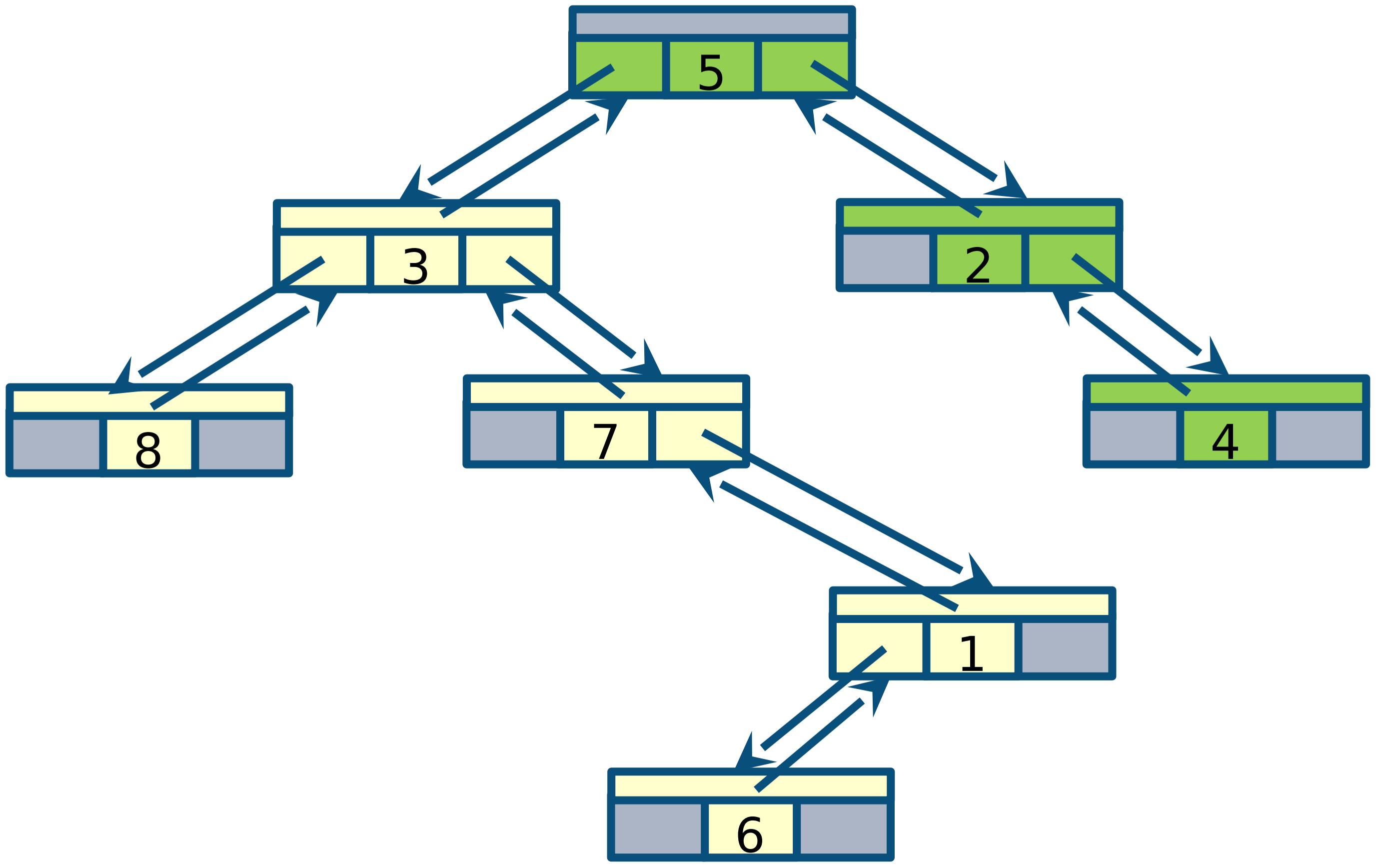
Example

- **Path from 3 to 6**
  - 3,7,1,6
  - Length 3



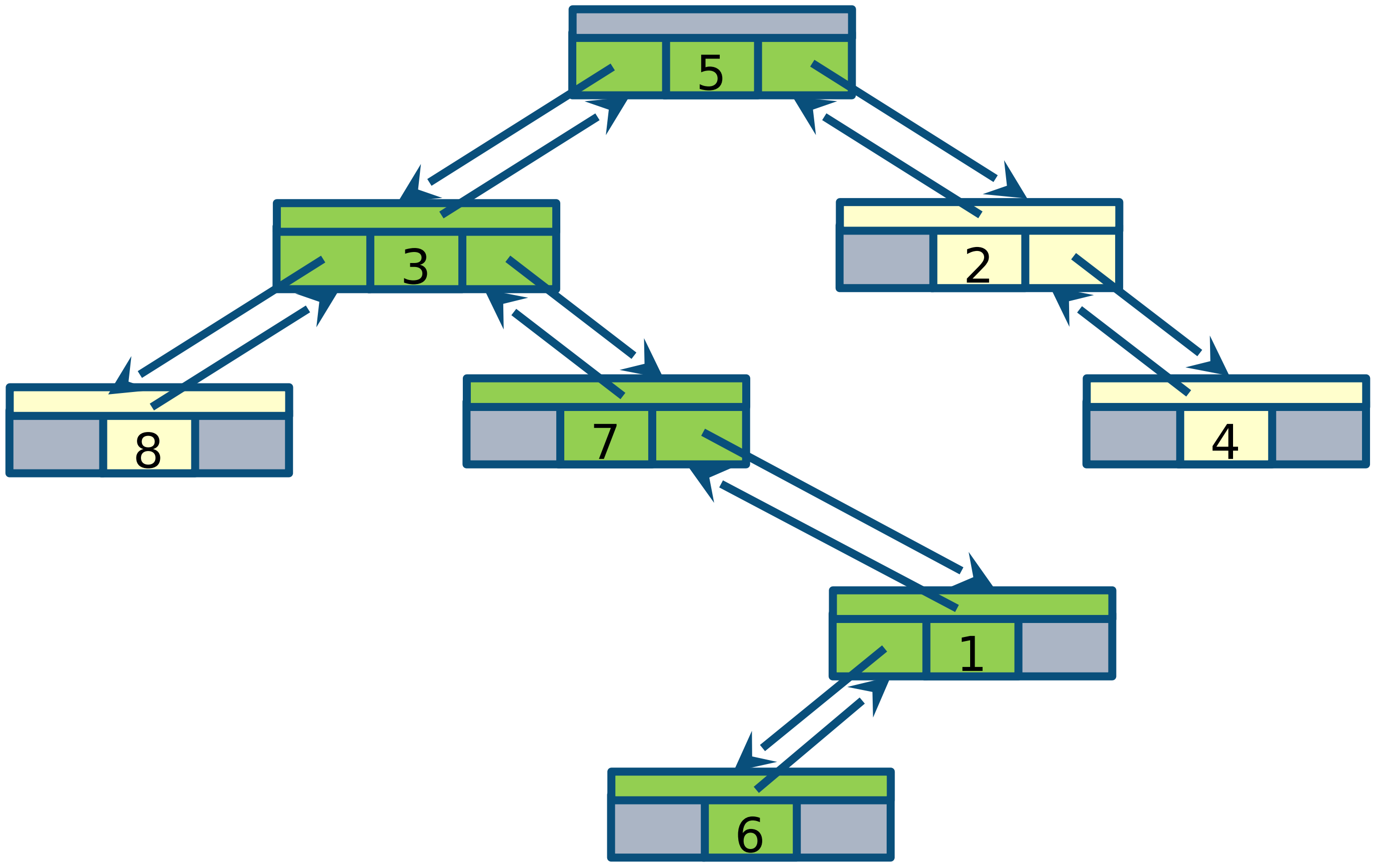
Example

- **Path from 5 to 4**
  - 5,2,4
  - Length 2



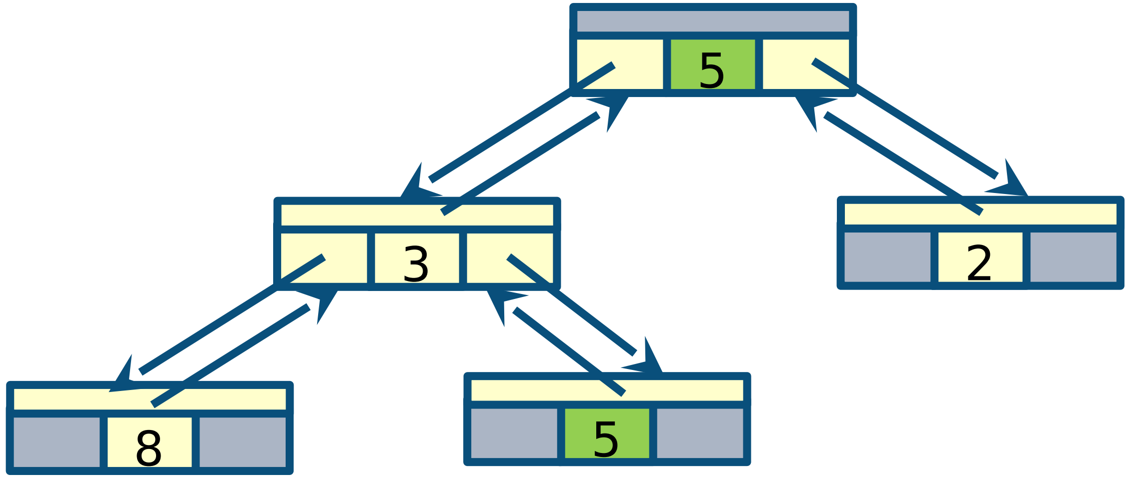
Example

- **Longest path from 5 leads to 6**
  - 5,3,7,1,6
  - Length 4
  - Height of the tree is 4

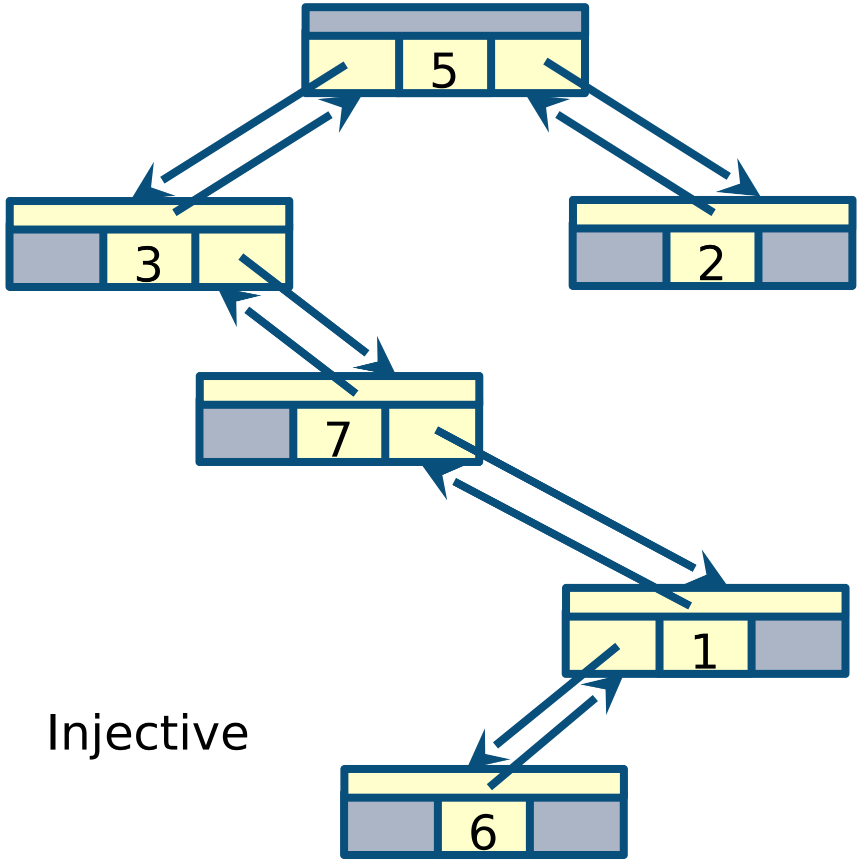


Duplicate keys

- **A binary tree in which no key occurs in more than one node is injective**
  - We will mostly deal with injective binary trees



Non injective



Injective

Balanced trees

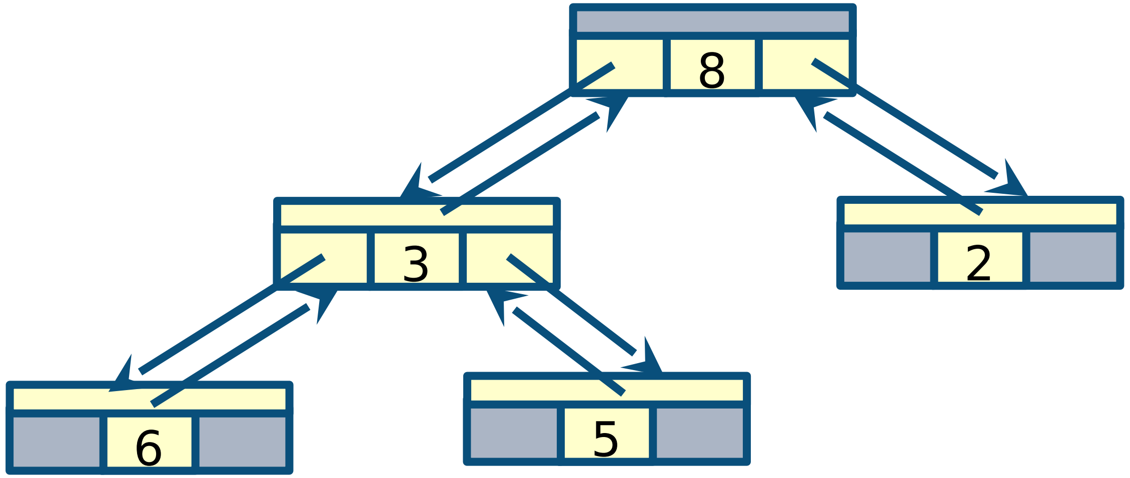
- **A balanced binary tree is a binary tree in which the left and right subtrees of every node differ in height by no more than 1**
  - Height O(log n)

- **An extremely unbalanced tree might have no right/left pointers**
  - It looks almost like a linked list
  - Height O(n)

- **The worst case running time of most tree operations is proportional to the height of the tree**
  - Generally tree-based algorithms work most efficiently on balanced trees

Examples

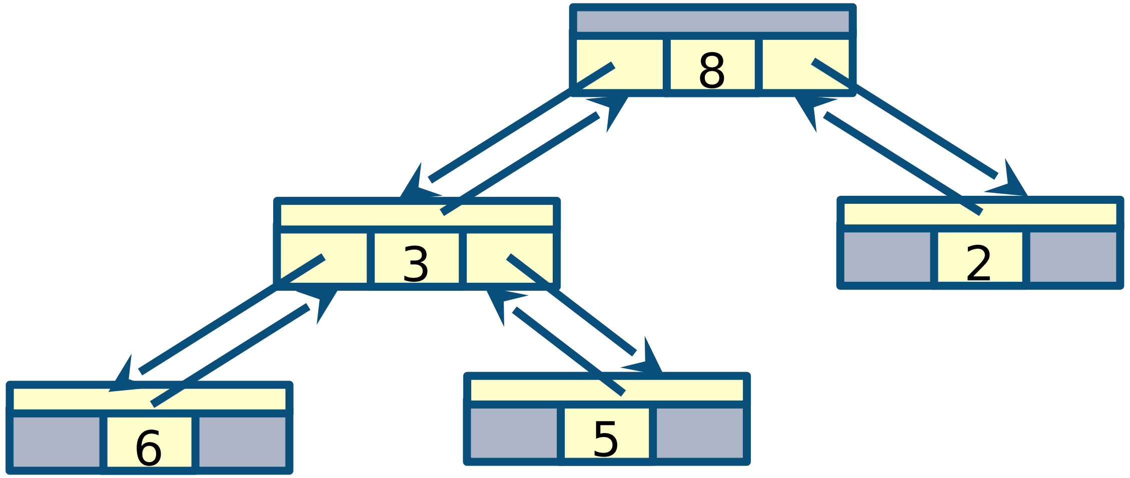
- **Is this tree balanced?**



Examples

- **Is this tree balanced?**

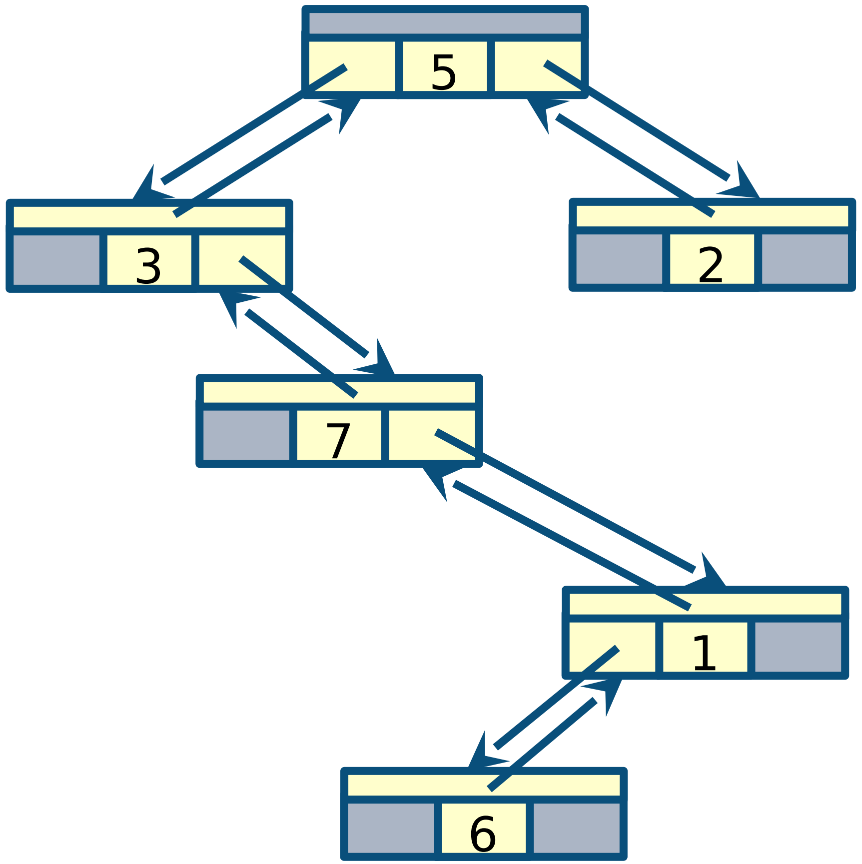
- **We need to check the height of the right and left subtrees of every node**
  - Height of leaves is 0
  - Consider 3: height of left = height right = 0
  - Consider 8: height left = 1 and height right = 0



- **Yes**

Examples

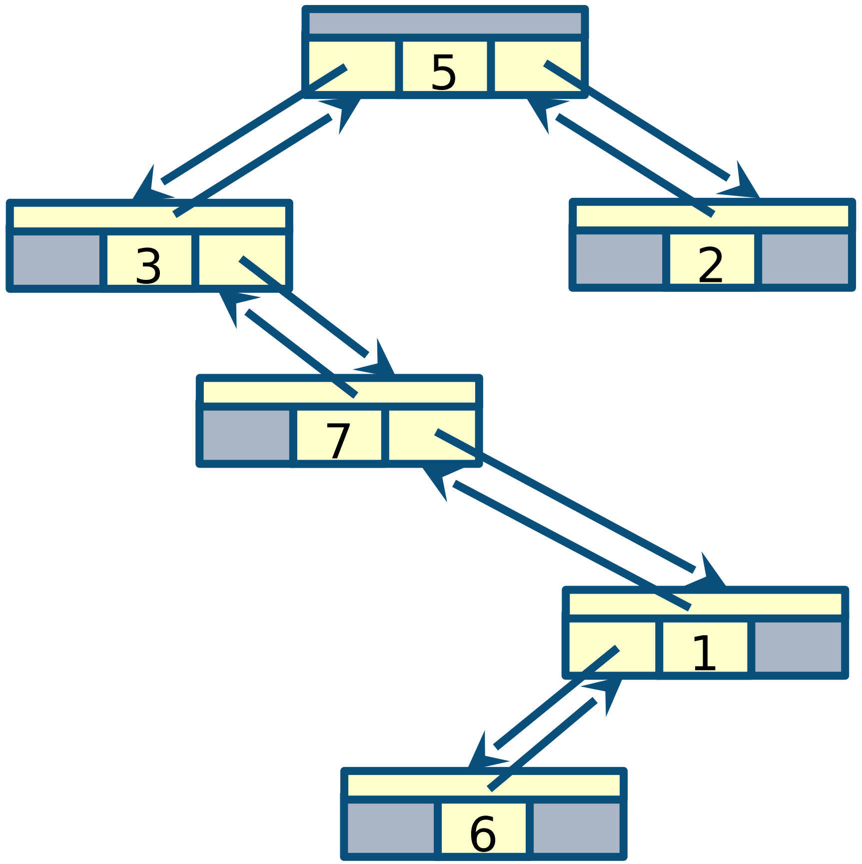
- **Is this tree balanced?**



Examples

- **Is this tree balanced?**
  - Consider root 5
  - Height of left = 3
  - Height of right = 0

- **Not balanced**



Number of different binary trees

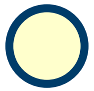
- **How many different binary trees with n nodes is it possible to construct?**
  - We ignore key attribute

- **Consider different values of n**
  - n=0 One tree, the empty tree

Number of different binary trees

- **How many different binary trees with n nodes is it possible to construct?**
  - We ignore key attribute

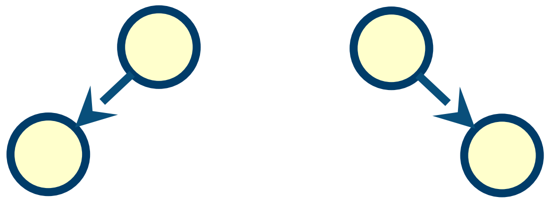
- **Consider different values of n**
  - n=0 One tree, the empty tree
  - n=1 One tree (a root with no children)



Number of different binary trees

- **How many different binary trees with n nodes is it possible to construct?**
  - We ignore key attribute

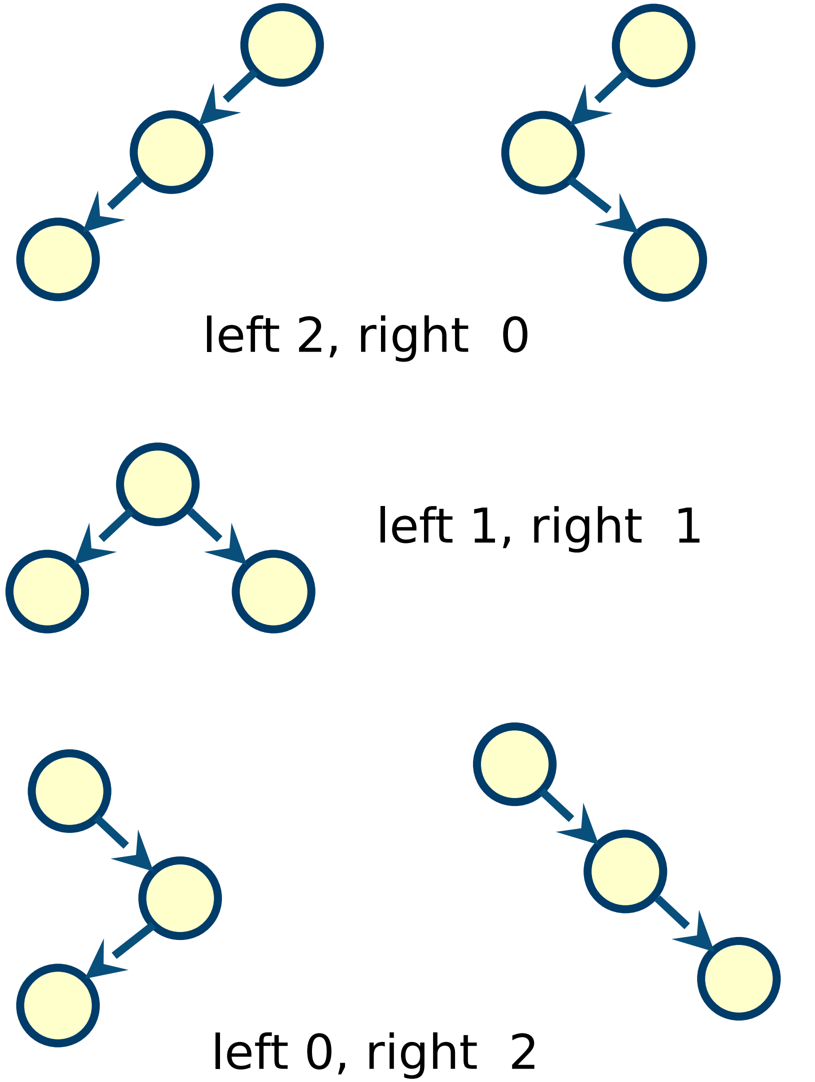
- **Consider different values of n**
  - n=0 One tree, the empty tree
  - n=1 One tree (a root with no children)
  - n=2 2 trees (a root and either a left or right child)



Number of different binary trees

- **How many different binary trees with n nodes is it possible to construct?**
  - We ignore key attribute

- **Consider different values of n**
  - n=0 One tree, the empty tree
  - n=1 One tree (a root with no children)
  - n=2 2 trees (a root and either a left or right child)
  - n=3 5 trees (a root and combination of left/right subtrees from size 0 to 2)



Number of different binary trees

- **Define C_n as the number of binary trees with n nodes**
  - $C_0 = C_1 = 1$
  - $C_2 = C_0C_1 + C_1C_0 = 1 + 1 = 2$
  - $C_3 = C_0C_2 + C_1C_1 + C_2C_0 = 2 + 1 + 2 = 5$
  - $C_4 = C_0C_3 + C_1C_2 + C_2C_1 + C_3C_0 = 5 + 2 + 2 + 5 = 14$

- **The following recurrence emerges (with $C_0=1$)**
  - with $n \geq 0$

- **This is the recurrence for the Catalan numbers**
  - 1, 1, 2, 5, 14, 42, 132, 429, 1430, ….

Traversals

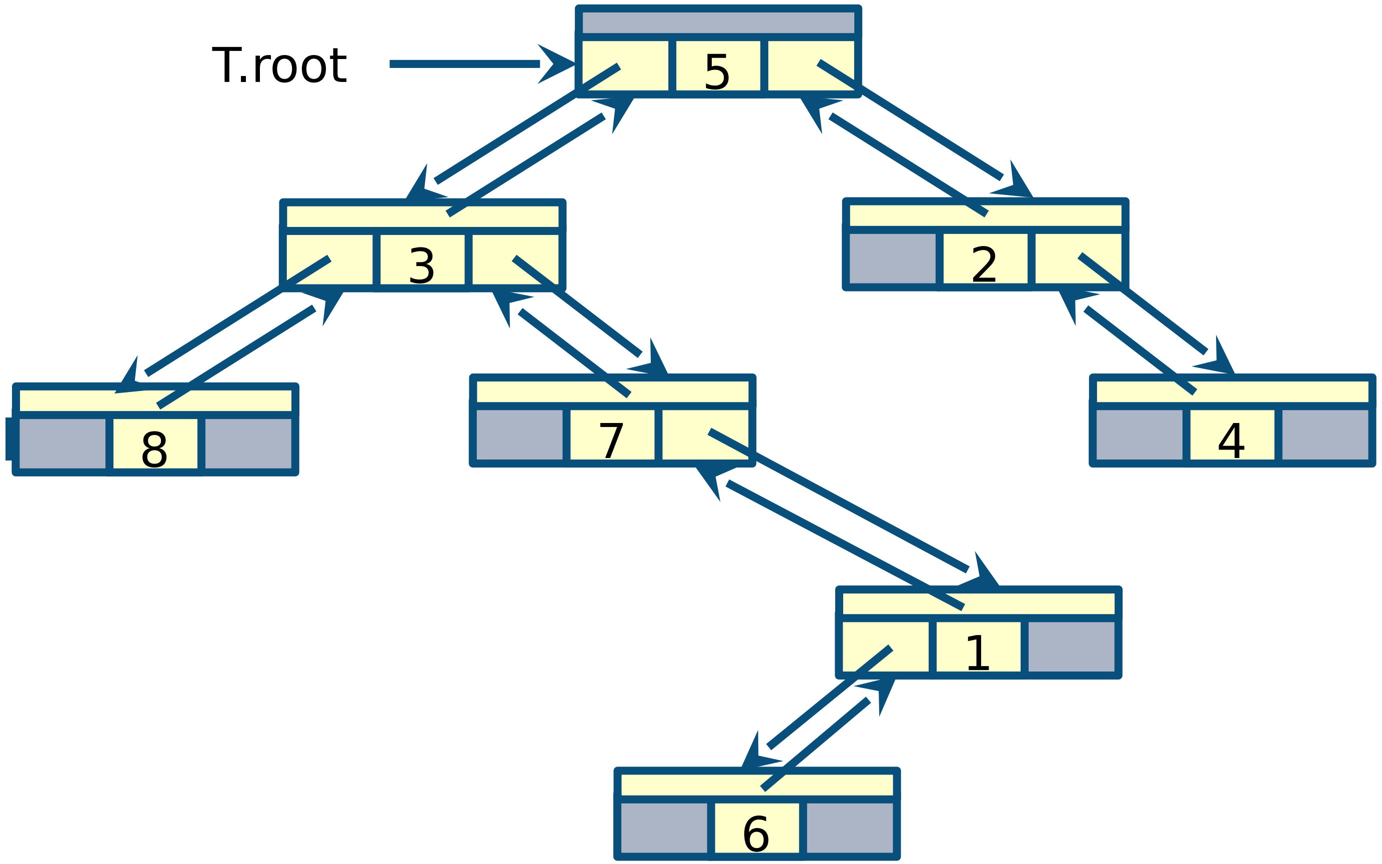
- **A binary tree traversal (or walk) is the process of visiting each node of the tree, exactly once**

- **We study three special traversals**
  - Inorder traversal
  - Preorder traversal
  - Postorder traversal

- **They are all defined recursively**

Inorder traversal

```
INORDER(x)
  if x != NIL
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

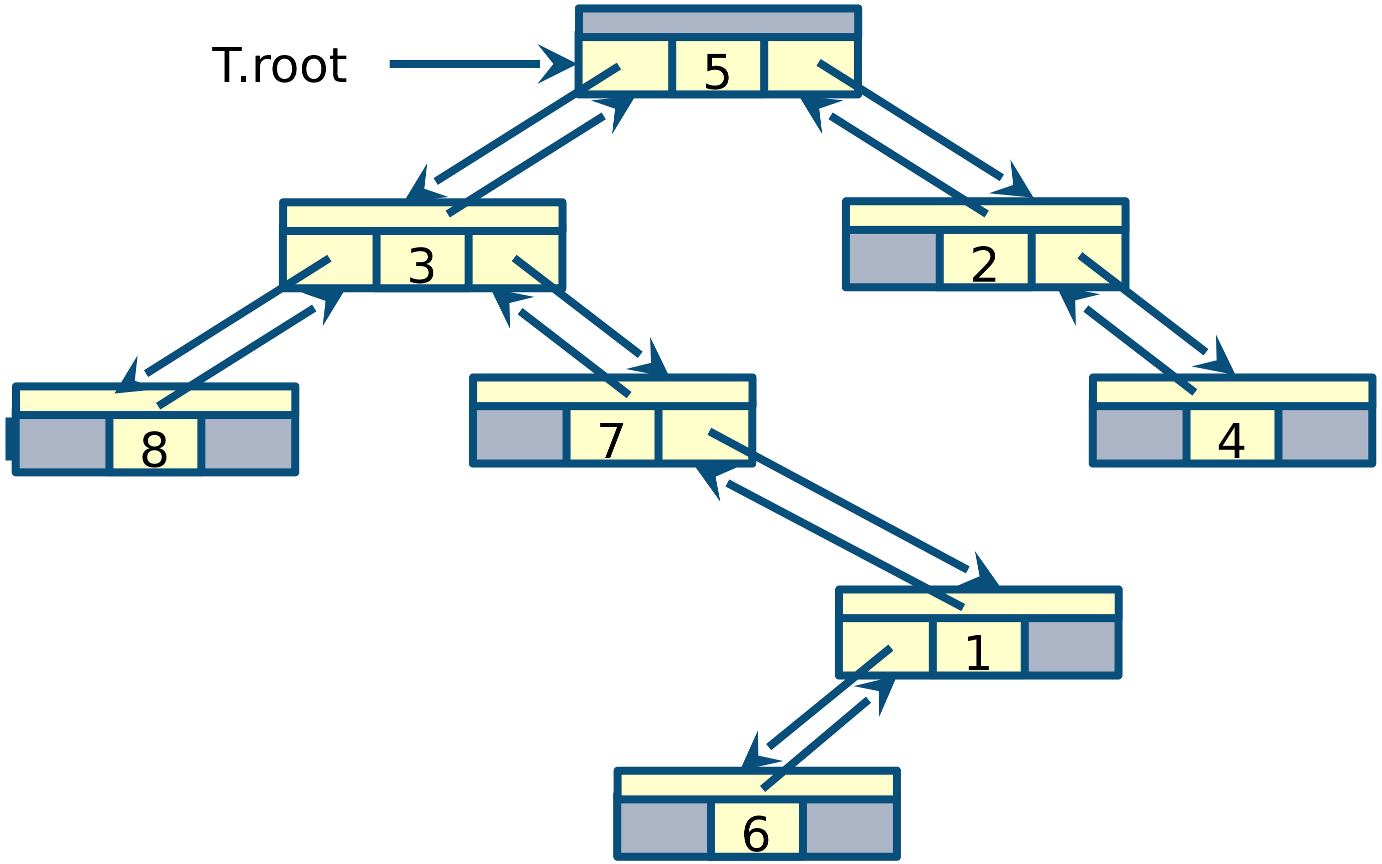


- To print all the elements we call **INORDER(T.root)**

- Example

Inorder traversal

```
INORDER(x)
  if x != NIL
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```



- **To print all the elements we call INORDER(T.root)**

- **Example**
  - 8,3,7,6,1,5,2,4

Analysis

- **Theorem: If x is the root of a binary tree with n nodes, then INORDER(x) takes $\Theta(n)$ time**
- **Proof: We need to prove**

1. $T(n) = \Omega(n)$
2. $T(n) = O(n)$

```
INORDER(x)
  if x != NIL
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

Analysis

- **Theorem: If x is the root of a binary tree with n nodes, then INORDER(x) takes $\Theta(n)$ time**
- **Proof: We need to prove**

1. $T(n) = \Omega(n)$
2. $T(n) = O(n)$

```
INORDER(x)
  if x != NIL
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

3. Trivial. The running time $T(n) = \Omega(n)$ as all n nodes are visited

Analysis

- **Theorem: If x is the root of a binary tree with n nodes, then INORDER(x) takes $\Theta(n)$ time**
- **Proof: We need to prove**

1. $T(n) = \Omega(n)$
2. $T(n) = O(n)$

```
INORDER(x)
  if x != NIL
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

2. With the iterative method

– $T(0) = O(1)$ to test x != NIL

– $T(n) = T(k) + T(n-k-1) + O(1)$

– The left subtree has k nodes and the right subtree has $n - k - 1$ nodes

Analysis

- **Theorem: If x is the root of a binary tree with n nodes, then INORDER(x) takes Θ(n) time**
- **Proof: We need to prove**
  1. $T(n) = \Omega(n)$ □
  2. $T(n) = O(n)$

```
INORDER(x)
  if x != NIL
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

2. With the iterative method

– $T(0) = O(1)$ to test x != NIL

– $T(n) = T(k) + T(n\text{-}k\text{-}1) + O(1)$

$= T(0) + T(n\text{-}1) + O(1)$

$= O(1) + T(n\text{-}1) + O(1)$

$= T(n\text{-}1) + O(1)$

Set $k=0$ to hit the base case on the left subtree

Analysis

- **Theorem: If x is the root of a binary tree with n nodes, then INORDER(x) takes $\Theta(n)$ time**
- **Proof: We need to prove**

1. $T(n) = \Omega(n)$
2. $T(n) = O(n)$

```
INORDER(x)
  if x != NIL
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

2. With the iterative method

- $T(0) = O(1)$ to test x != NIL
- $T(n) = T(n-1) + O(1)$

We have already seen this equation

Analysis

- **Theorem: If x is the root of a binary tree with n nodes, then INORDER(x) takes Θ(n) time**
- **Proof: We need to prove**

1. $T(n) = \Omega(n)$
2. $T(n) = O(n)$

```
INORDER(x)
  if x != NIL
    INORDER(x.left)
    print x.key
    INORDER(x.right)
```

2. With the iterative method

– $T(0) = c_1$

– $T(n) = T(n-1) + c_2$

$= T(n-2) + 2c_2$

$= \ldots$

$= T(n-j) + jc_2 = T(0) + nc_2 = c_1 + nc_2 = O(n)$

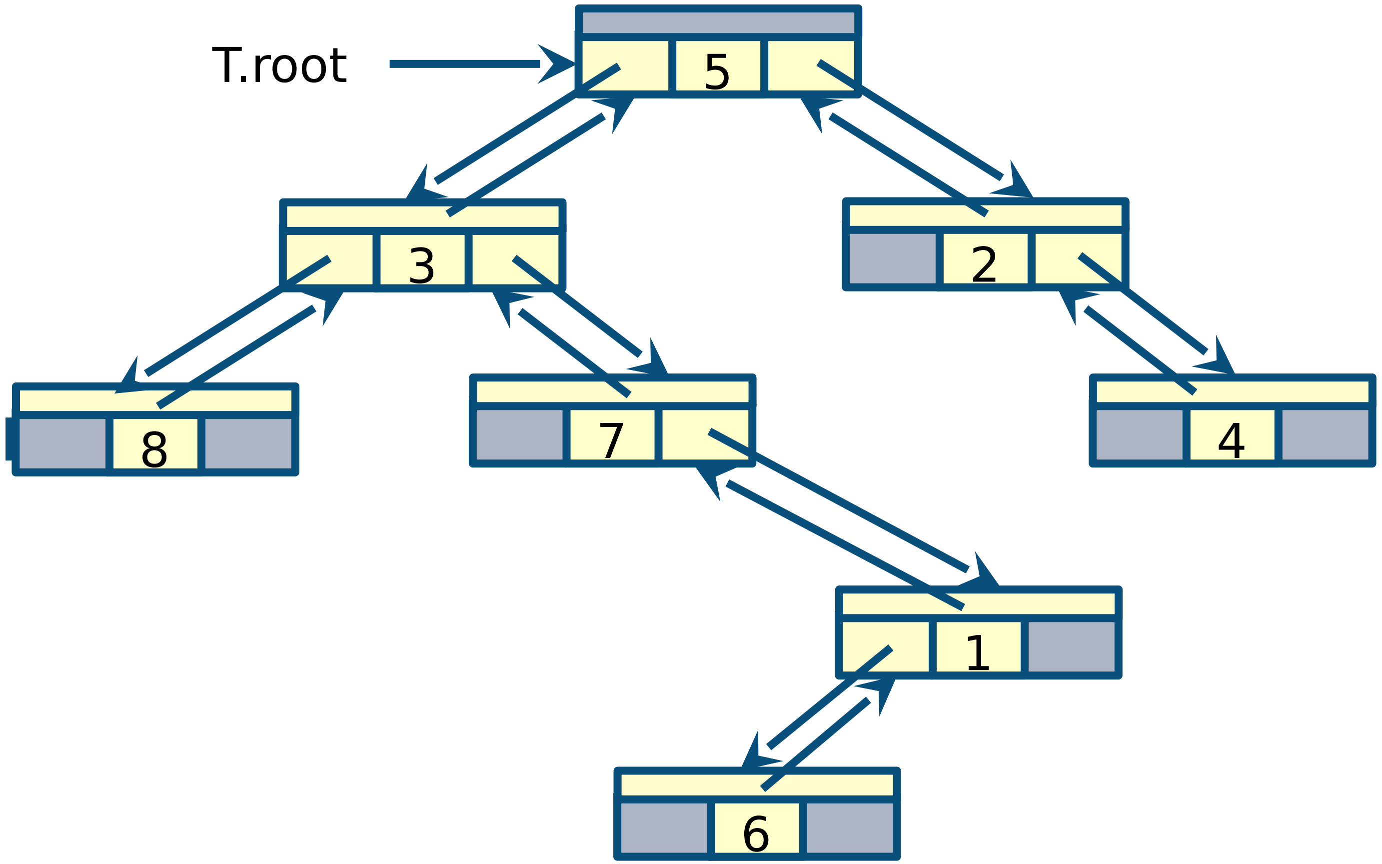
Set $j = n$ to hit the base case

Preorder traversal

```
PREORDER(x)
  if x != NIL
     print x.key
     PREORDER(x.left)
     PREORDER(x.right)
```

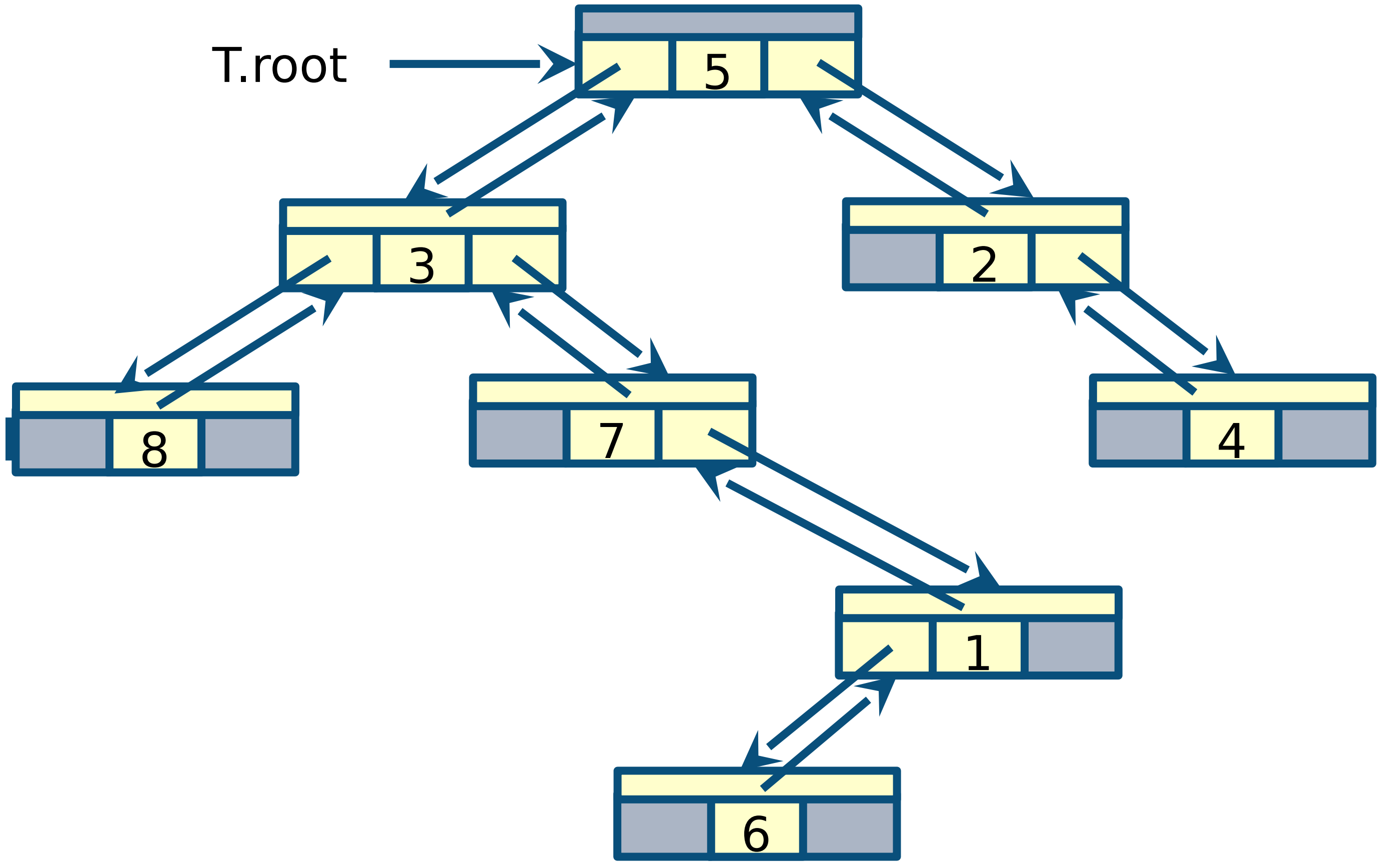
- **To print all the elements we call** **PREORDER(T.root)**

- **Example**



Preorder traversal

```
PREORDER(x)
  if x != NIL
     print x.key
     PREORDER(x.left)
     PREORDER(x.right)
```



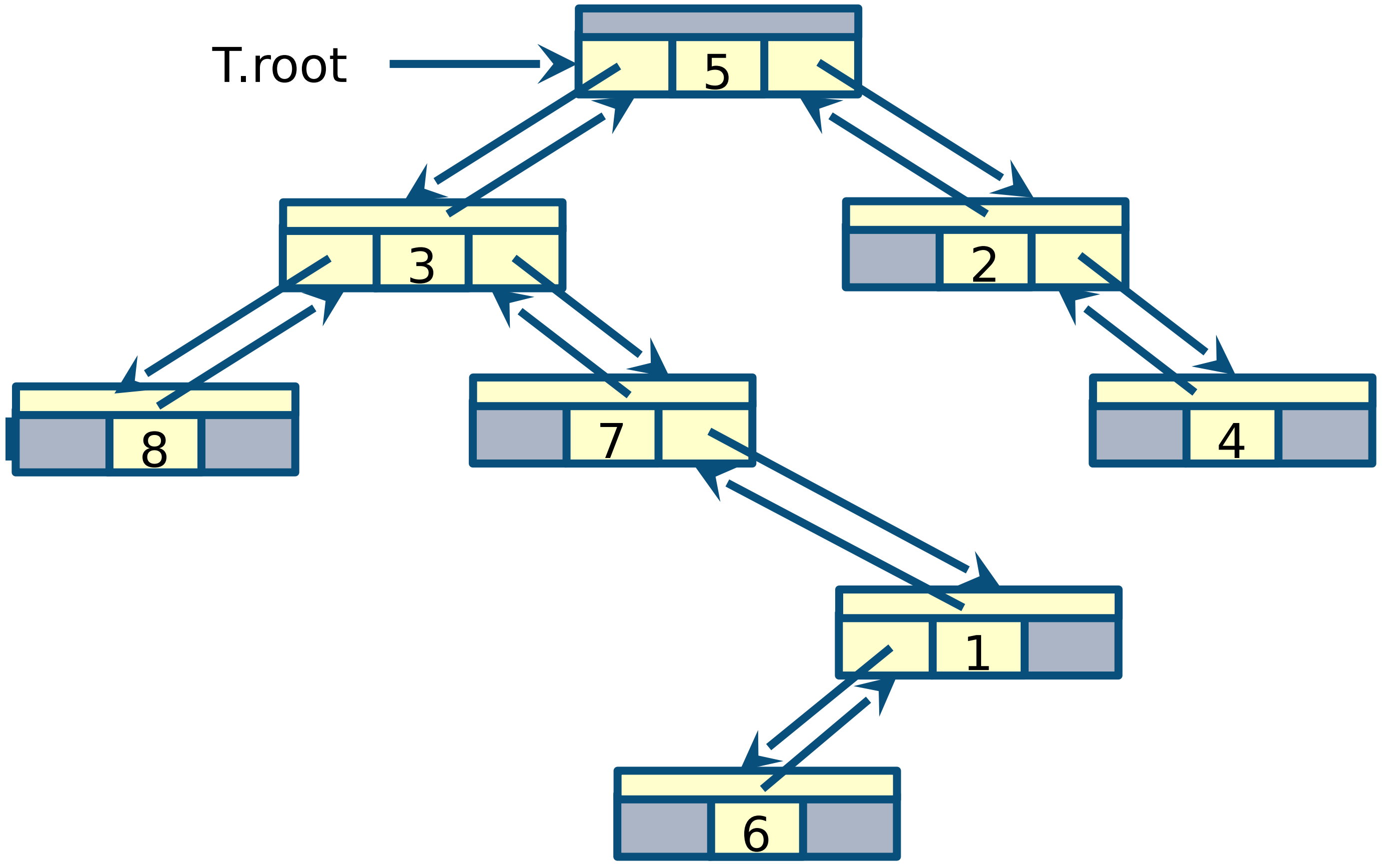
- **To print all the elements we call PREORDER(T.root)**

- **Example**
  - 5,3,8,7,1,6,2,4

Postorder traversal

```
POSTORDER(x)
  if x != NIL
     POSTORDER(x.left)
     POSTORDER(x.right)
     print x.key
```

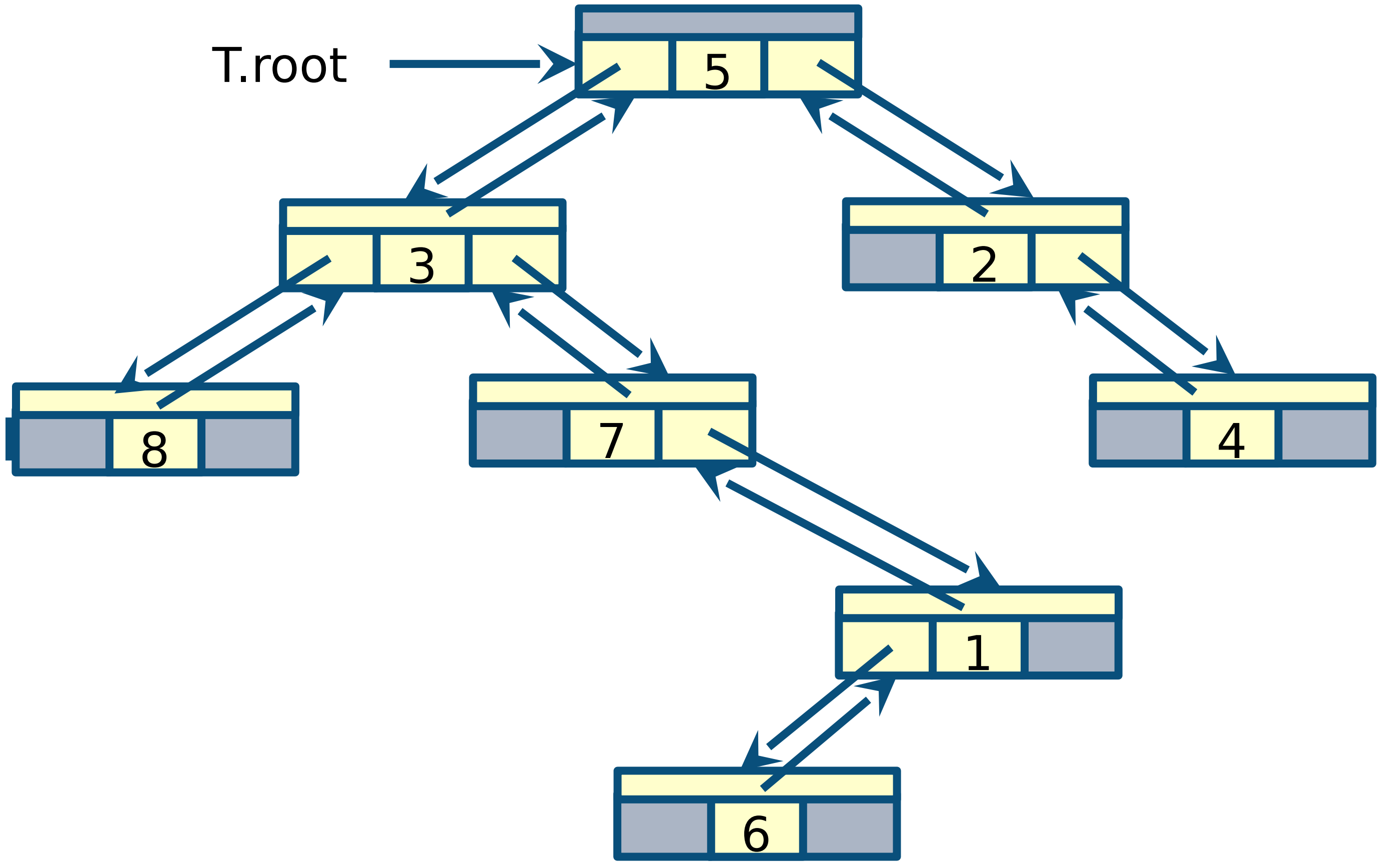
- **To print all the elements we call POSTORDER(T.root)**



- **Example**

Postorder traversal

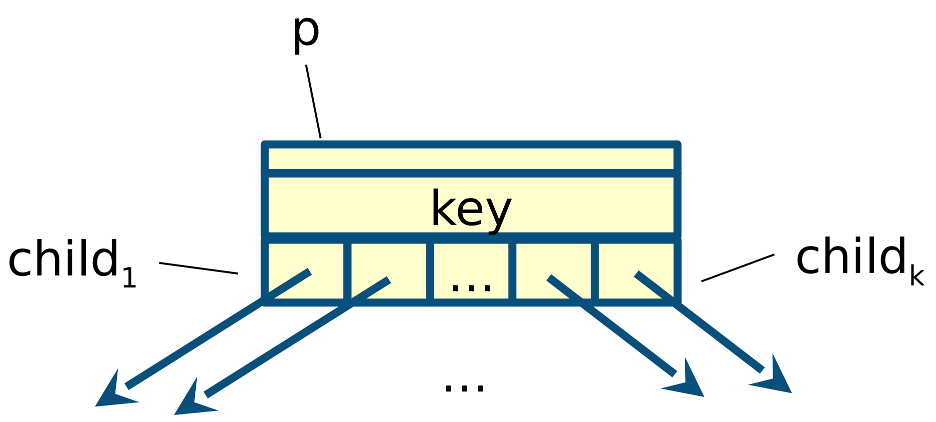
```
POSTORDER(x)
  if x != NIL
     POSTORDER(x.left)
     POSTORDER(x.right)
     print x.key
```



- **To print all the elements we call POSTORDER(T.root)**
- **Example**
  - 8,6,1,7,3,4,2,5

K-ary trees

- **Trees with a bounded number of children (k-ary trees) can be easily represented with additional pointer attributes $child_1, \ldots, child_k$**



- **Limitations**
  - A lot of NIL values if many nodes have less than k children
  - k is fixed (it needs to be known in advance)

Rooted trees with unbounded branching

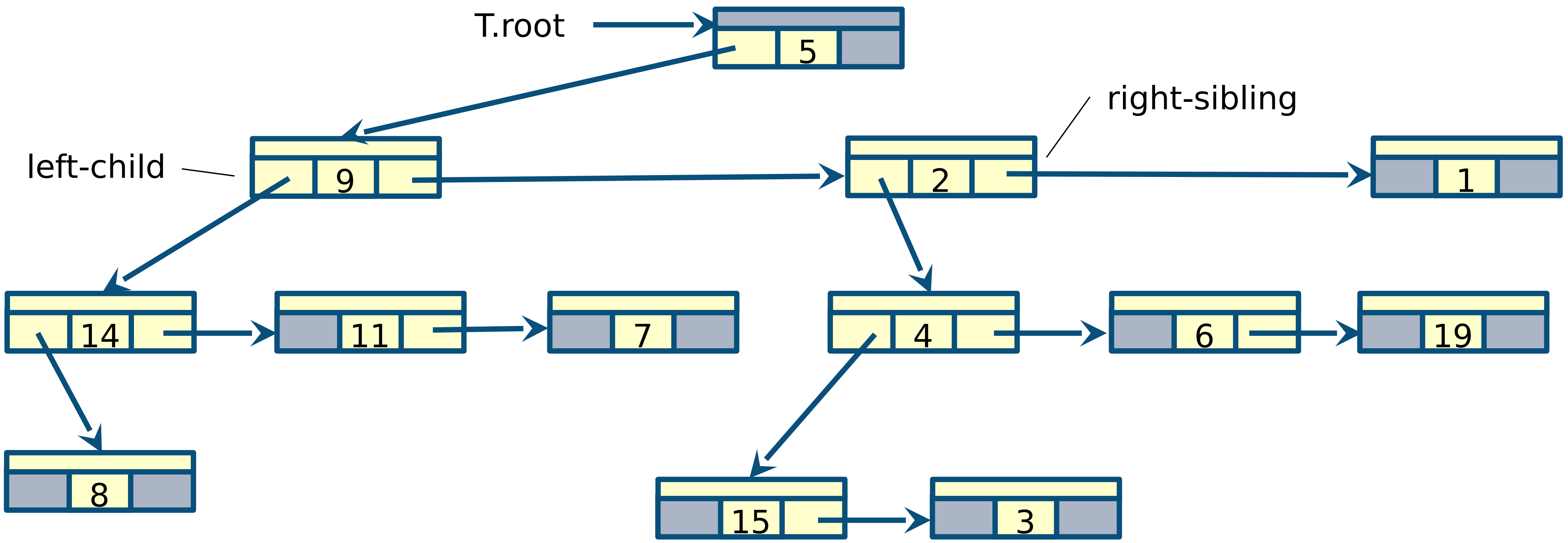
- **Trees with an unbounded number of children can easily be represented with additional attributes**
  - x.left-child points to the leftmost child of x
  - x.right-sibling points to the sibling of x immediately to its right

- **If x has no children x.left-child = NIL**
- **If x is the rightmost child x.right-sibling = NIL**

- **Limitations**
  - Accessing a child is $O(k)$ as we need to scan all the list of children to access the rightmost child
  - k is the maximum branch degree

Example

- **Attribute p is not shown**



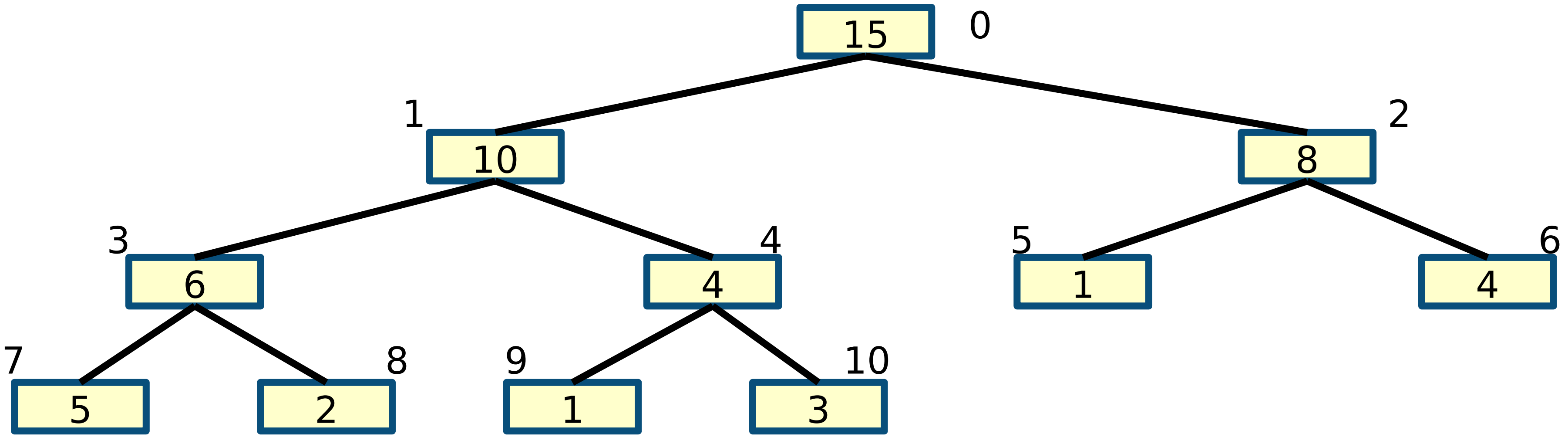
Other tree representations

- **Some algorithms only need to traverse the tree bottom up**
  - Only the parent pointers are present; no pointers to children

- **Computing the size of a tree with n nodes is $O(n)$**
  - Include a size attribute in each node: x.size = 1 + x.left.size +x.right.size

- **Strategies to handle duplicate keys**
  - Keep a list of nodes with equal keys at x
  - More examples in Lab 4

Array based representation (see Lecture 8)




```
LEFT(i)
  return (2 * i) + 1
```

```
RIGHT(i)
  return (2 * i) + 2
```

```
PARENT(i)
  return (i - 1) / 2
```

– A max-heap is represented as an array by assigning index 0 starting from the root and then increasing the index while going downwards from left to right on each tree level

15	10	8	6	4	1	4	5	2	1	3

Summary

- **Binary trees**
  - Properties
  - Traversals
- **Rooted trees with unbounded branching**
- **Other tree representations**