Binary Search Trees

Luciano Bononi
Dip. di Scienze dell'Informazione
Università di Bologna

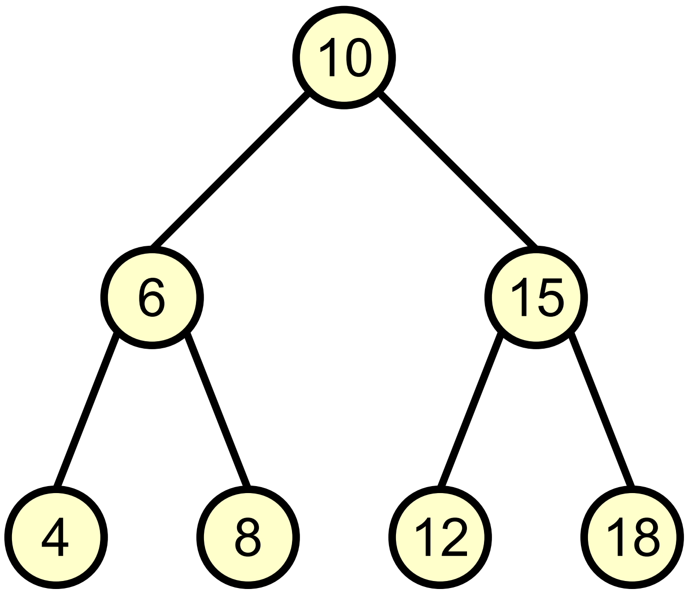
bononi@cs.unibo.it

Dictionary

- Dictionary
  - Dynamic set implementing the functions
    - Item search(Key key)
    - void insert(Key key, Item item)
    - void delete(Key key)
- Fundamental data structure for many applications
  - ex. to find a DB record by knowing the Key
- Possible examples of implementations
  - Sorted array
    - Search O(log n), insert/delete element O(n)
  - Unsorted list
    - Search/delete O(n), insert O(1)

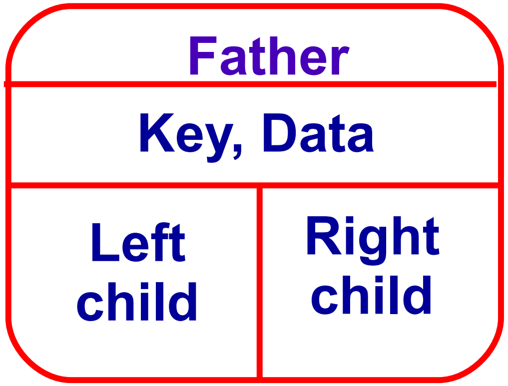
Binary Search Trees (BST)

- Idea
  - Implement a binary search in a tree
- Definition
  1. Every node v contains a set of datai v.data associated to a key v.key taken from a totally ordered domain (duplicate keys are possible)
  2. Keys of nodes in the <u>left subtree</u> of v are ≤ (=?) v.key
  3. Keys of nodes in the <u>right subtree</u> of v are ≥ (=?) v.key



Binary Search Trees (BST)

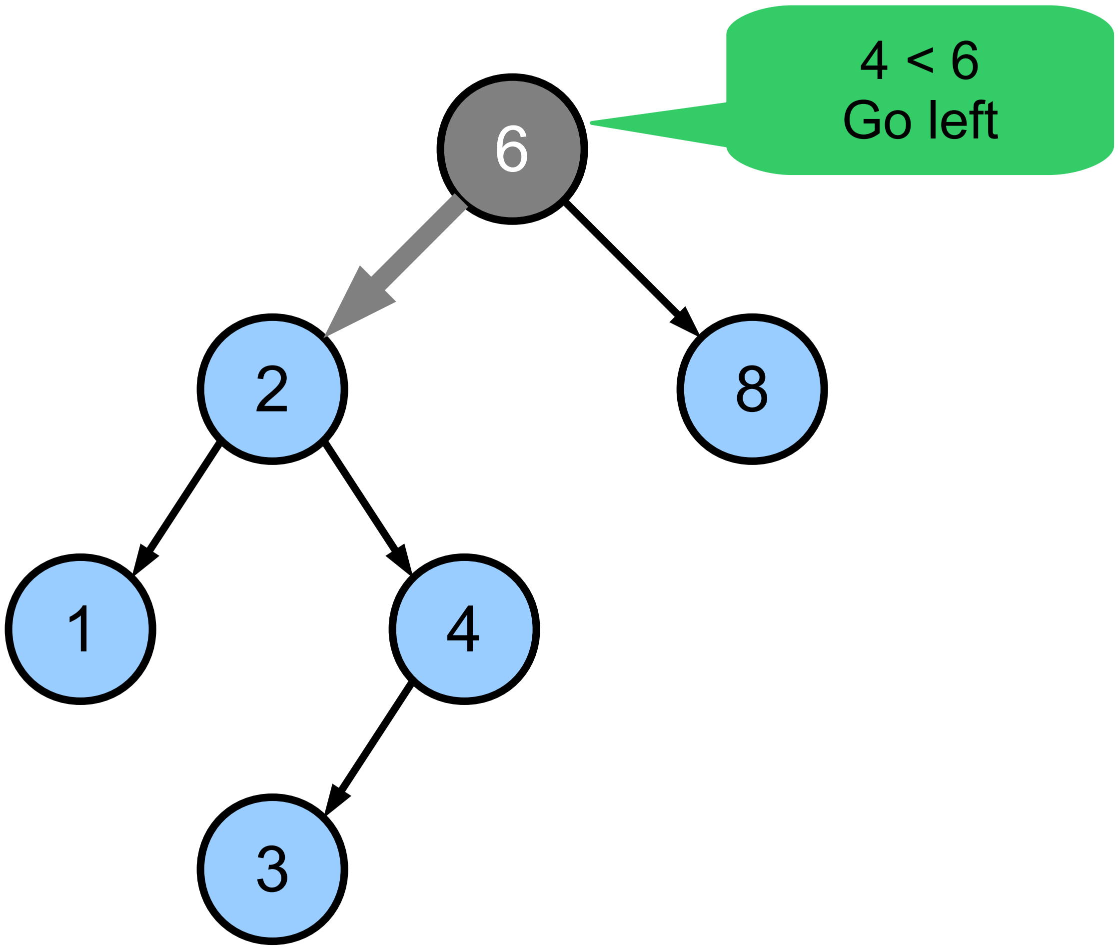
- Search property
  - Properties 2 and 3 allows to implement a dicotomic search algorithm
- Question: order property
  - How should I visit the tree to get a list of ordered values?
- Implementation details
  - Every node in the tree should maintain
    - Left and right child
    - Father
    - Key
    - Satellite data



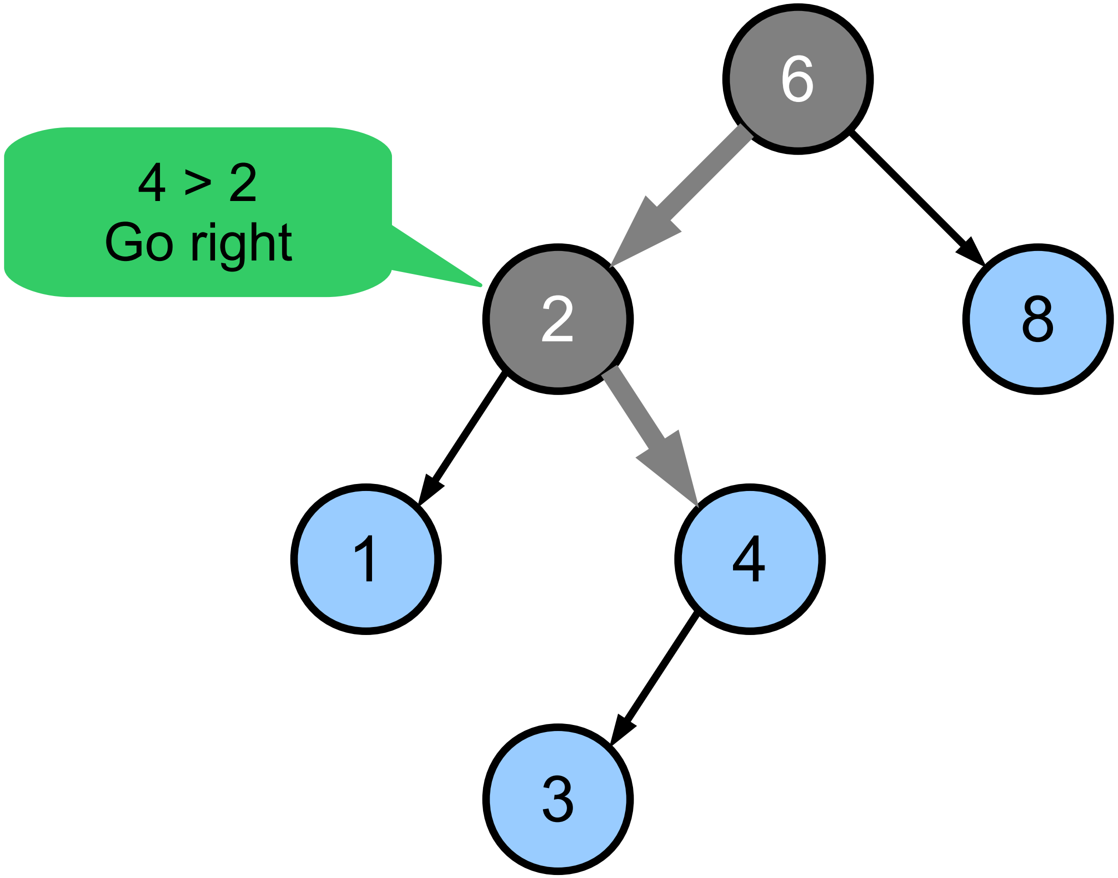
Dictionary interface

```
public interface Dictionary {
    /**
     * add the pair (e,k) to dictionary
     */
    public Rif insert(Object e, Comparable k);

    /**
     * deletes element u from dictionary
     */
    public void delete(Rif u);

    /**
     * returns element <code>e</code> with key k.
     * In case of duplicate keys, it returns
     * an arbitrary selected element with key k.
     */
    public Object search(Comparable k);
}
```

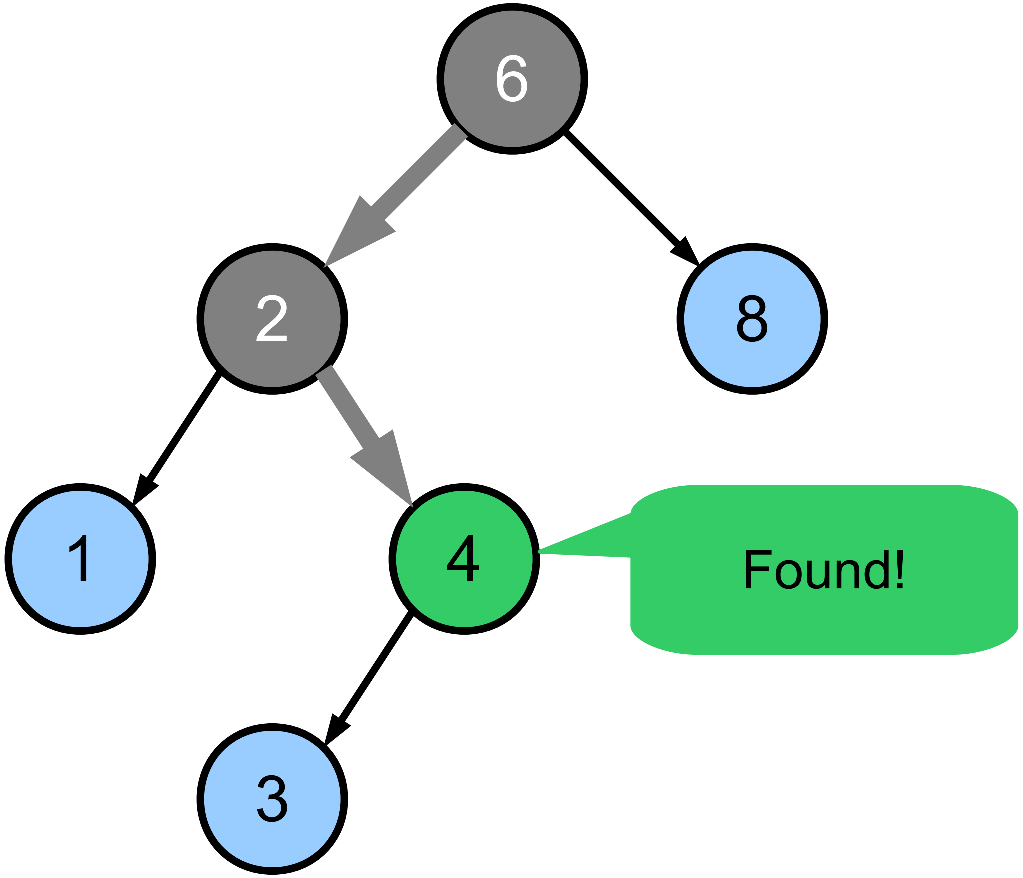
Example
searching key 4



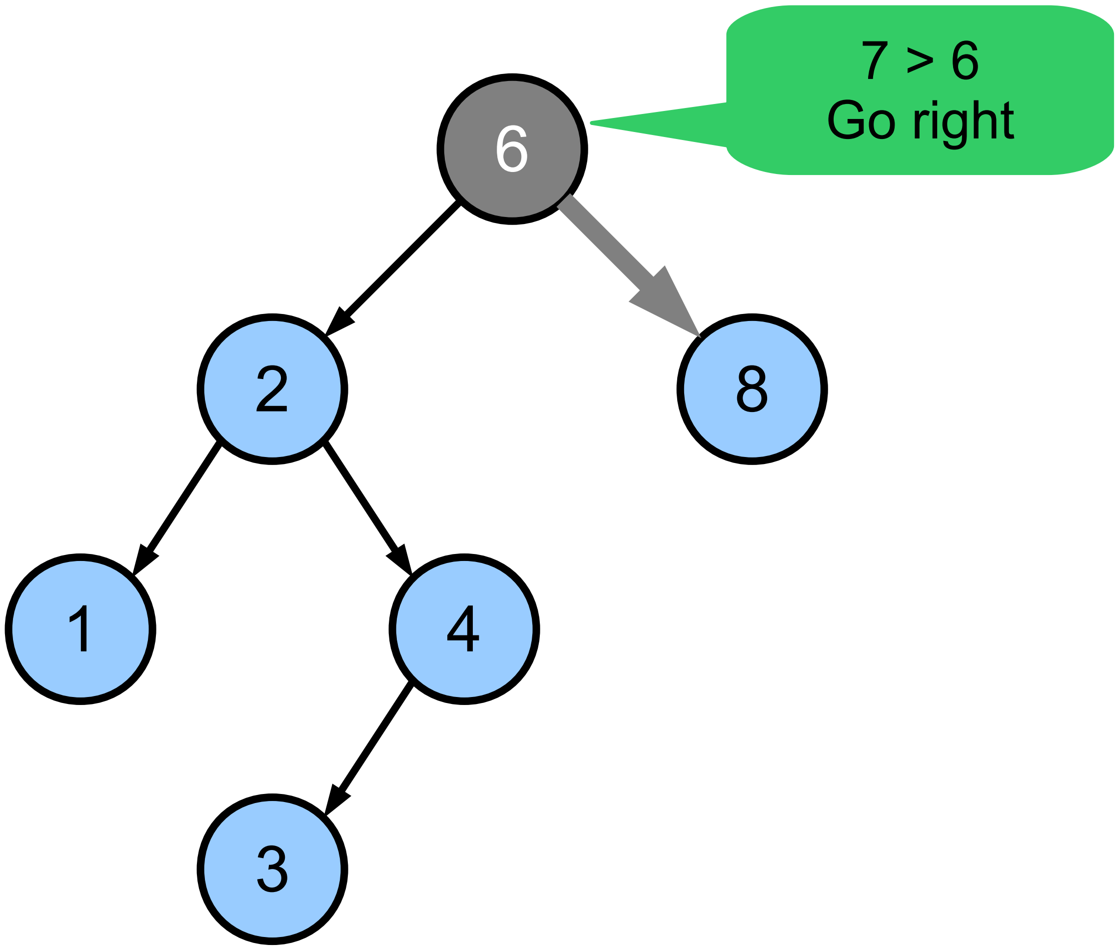
Example
searching key 4



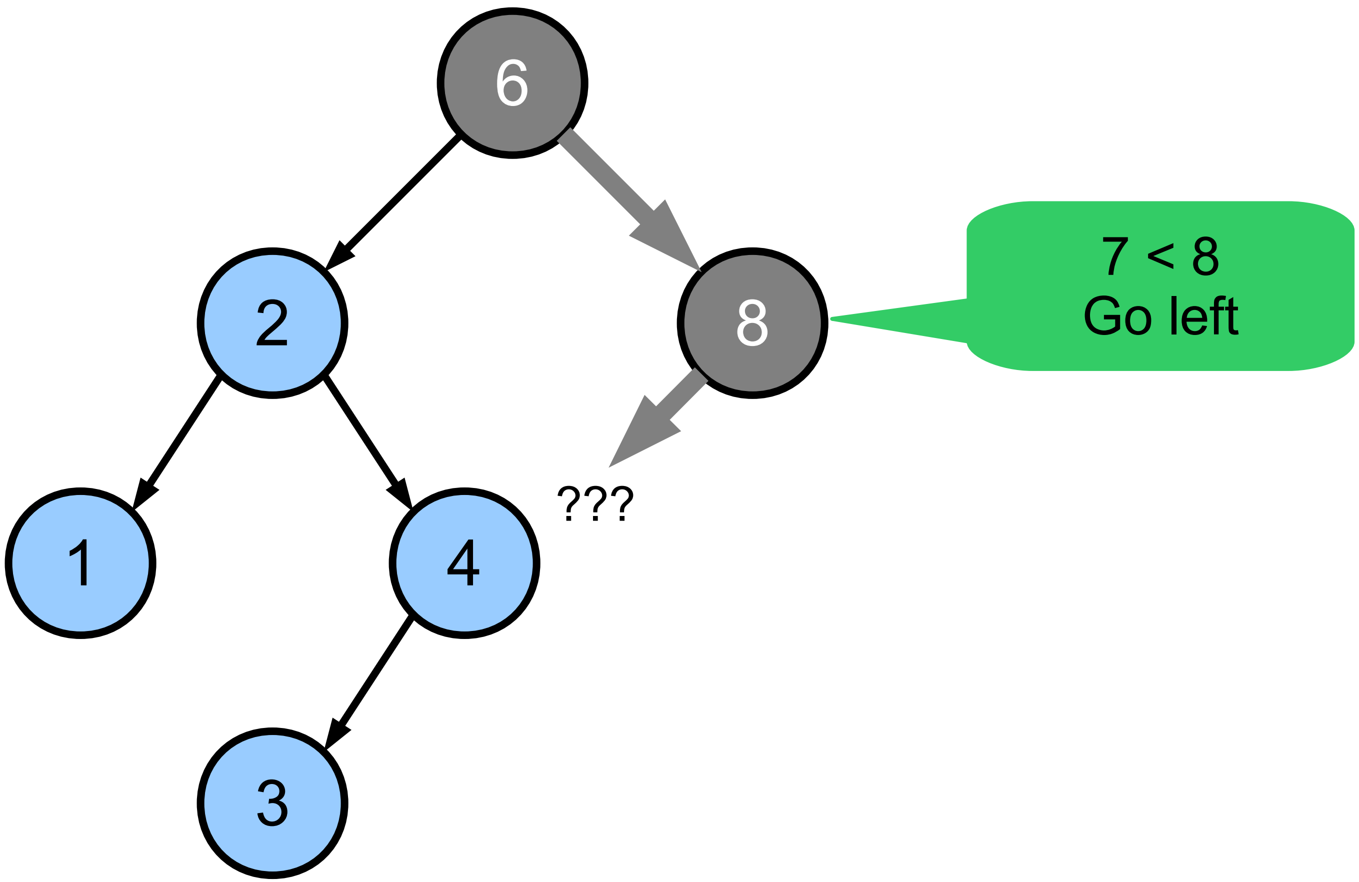
Example
searching key 4



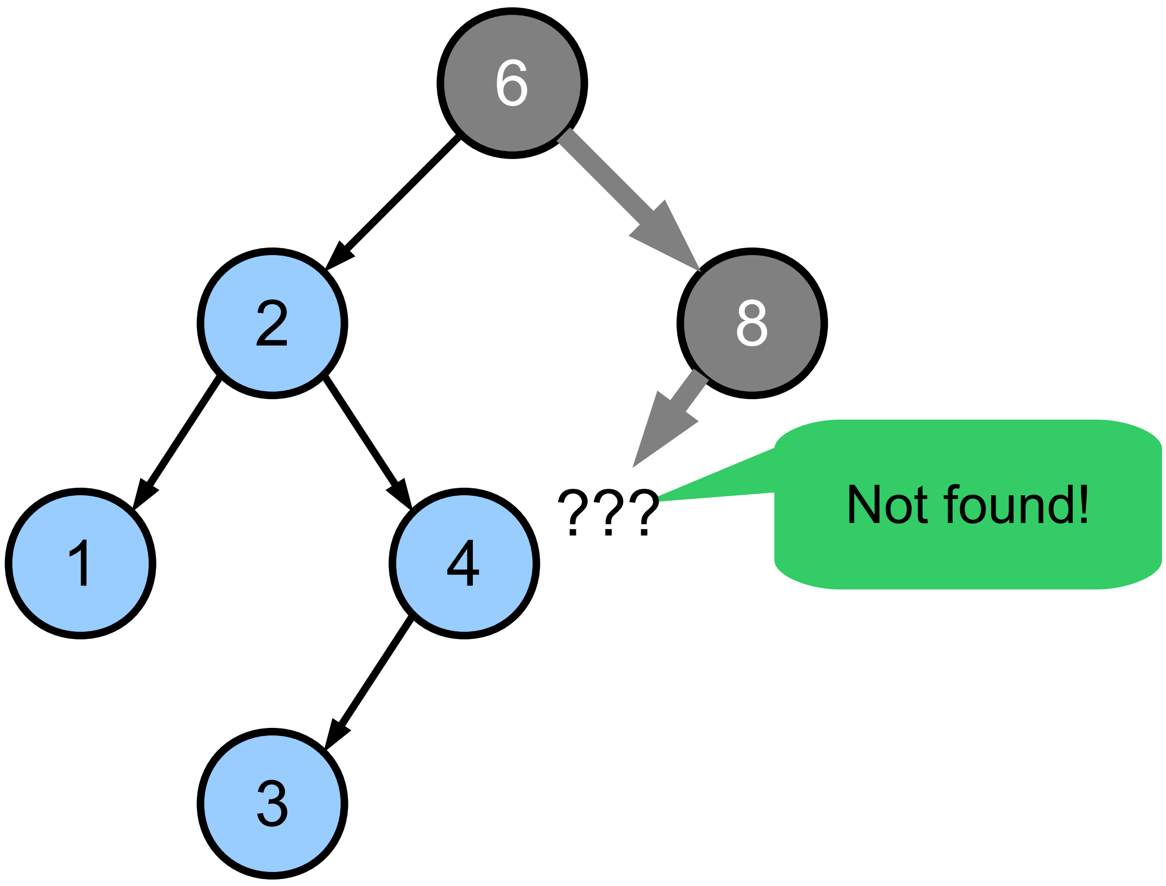
New example
searching key 7



New example
searching key 7



New example
searching key 7



Search: pseudocode

Recursive version

```
algorithm search(Nodo T, Key k) → Nodo
  if (T == null || k == T.key) then
    return T;
  elseif (k < T.key) then
    return search(T.left,k)
  else
    return search(T.right,k)
  endif
```

Iterative version

```
algorithm search(Nodo T, Key k) → Nodo
  while (T ≠ null)  do
    if (k == T.key) then
      return T;
    elseif (k < T.key) then
      T := T.left;
    else
      T := T.right;
    endif
  endwhile
  return null
```

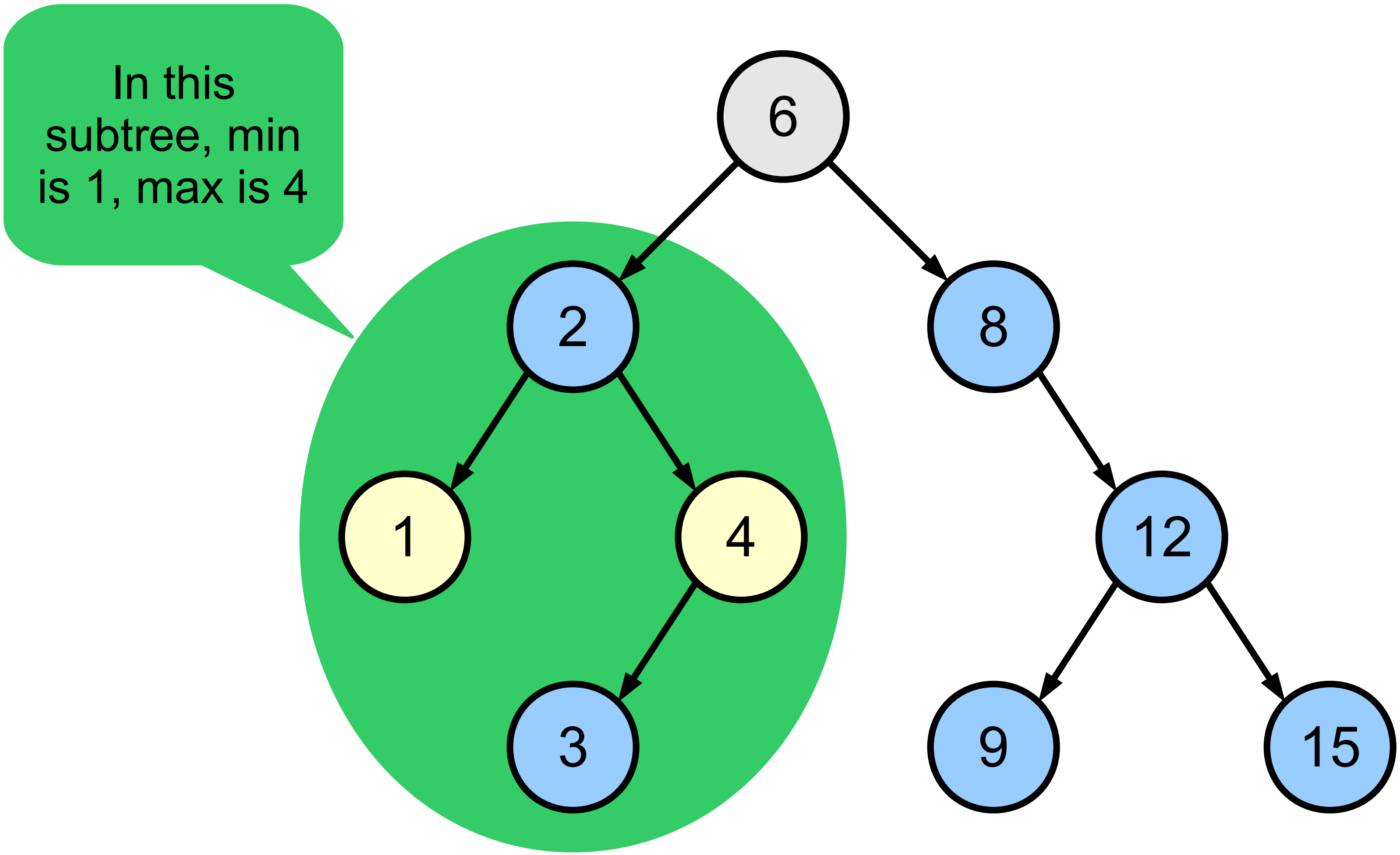
Class BSTree
package asdlab.libreria.AlberiRicerca

```
public class BSTree implements Dictionary {

    protected class InfoBR implements Rif {
        protected Object elem;
        protected Comparable key;
        protected Node node;
        protected InfoBR(Object e, Comparable k){
            elem = e; key = k; node = null;
        }
    }

    // Data structure containing the informations
    protected BinTree tree;

    public BSTree() { ... }

    // additional operations ...
}
```

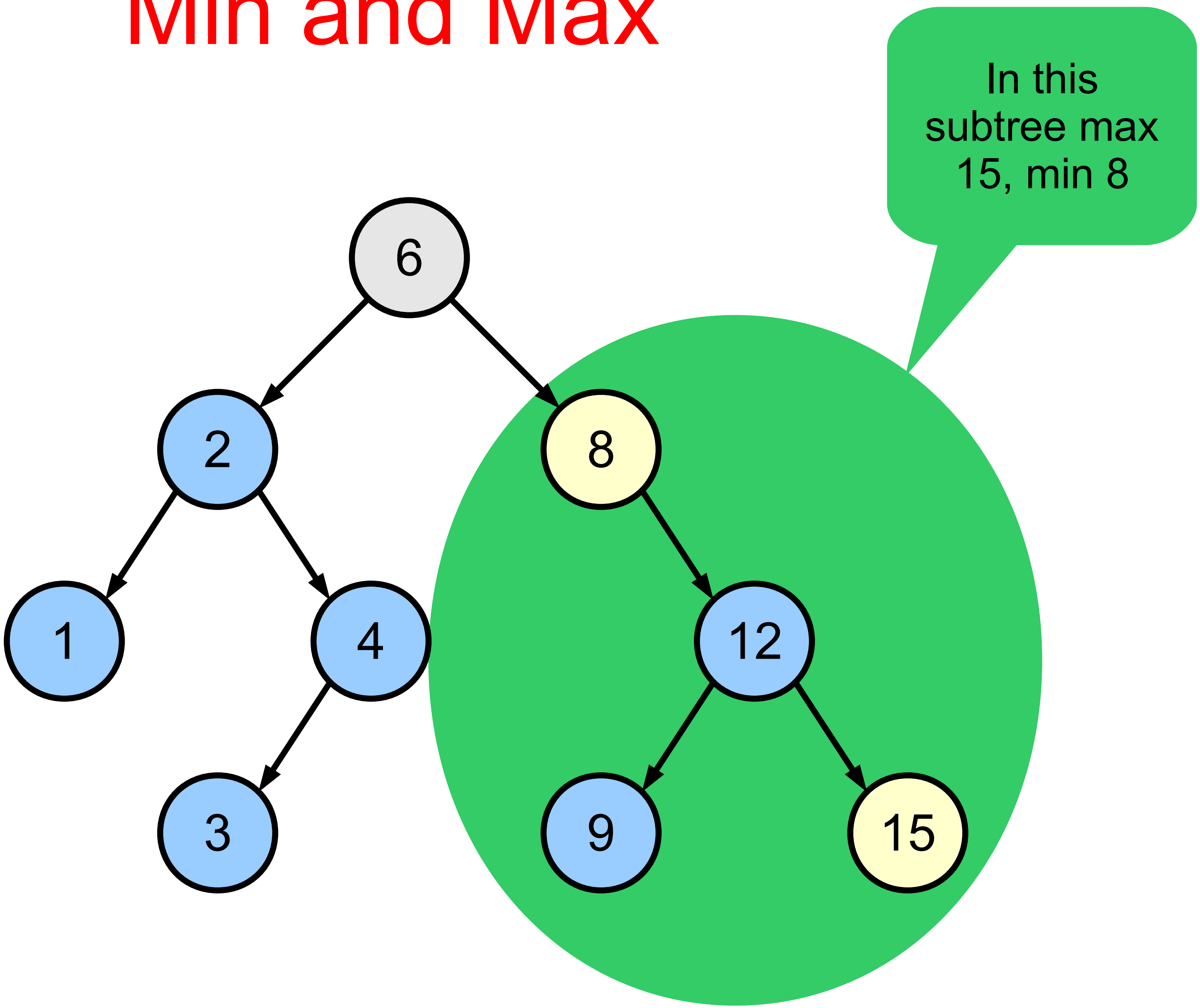
search: java implementation (iterative version)

```java
public Object search(Comparable k) {
    Node v = tree.root();
    while (v != null) {
        InfoBR i = (InfoBR)tree.info(v);
        if (k.equals(i.key))
            return i.elem;
        if (k.compareTo(i.key) < 0)
            v = tree.sx(v);
        else
            v = tree.dx(v);
    }
    return null;
}
```

Min and max



Min and Max



Search of the max element

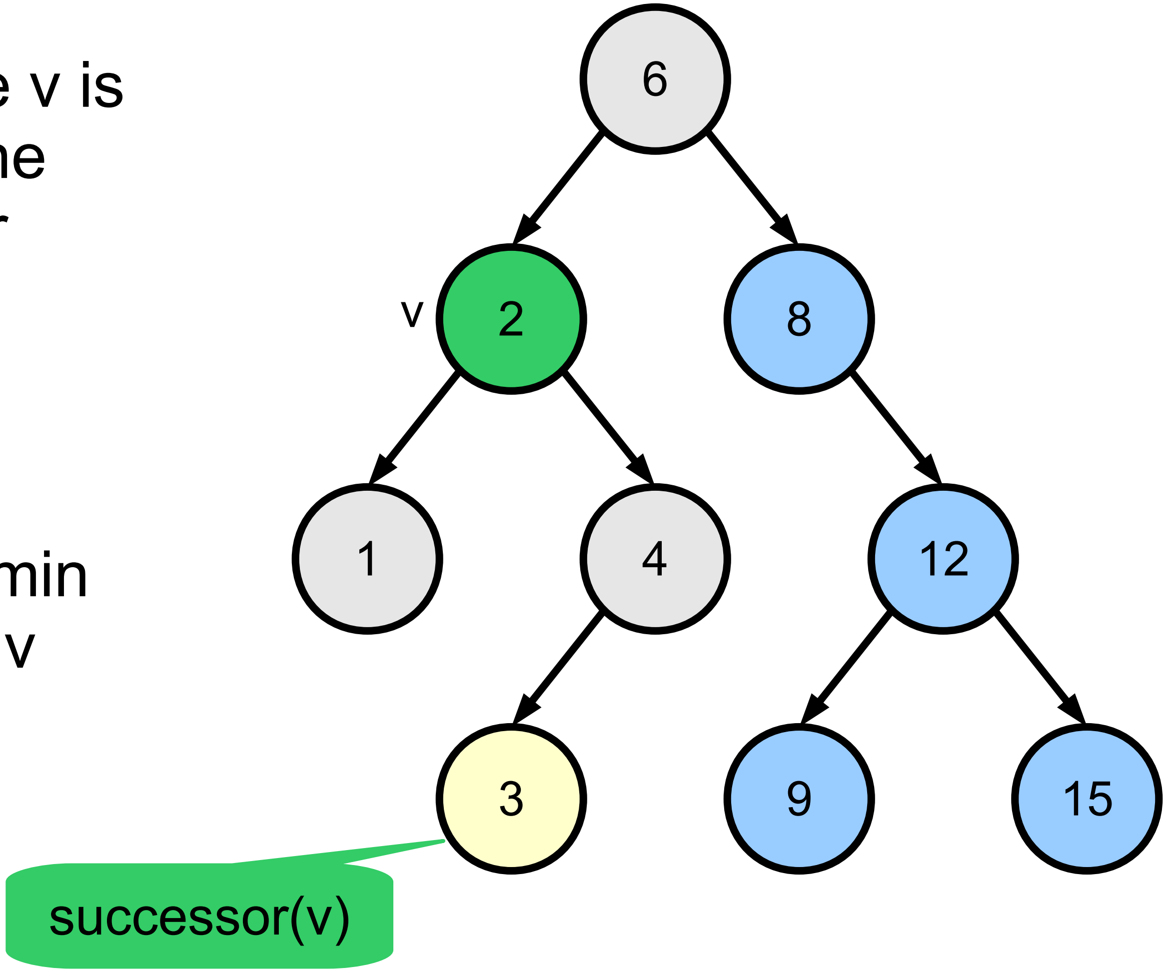
- Given a node v:
  - the maximum value in the tree rooted by v is in the "rightmost node"
  - the minmum value in the tree rooted by v is in the "leftmost node"

```
protected Node max(Node v) {
    while (v != null &&
          tree.dx(v) != null) {
        v = tree.dx(v);
    }
    return v;
}
```

```
protected Node min(Node v) {
    while (v != null &&
          tree.sx(v) != null) {
        v = tree.sx(v);
    }
    return v;
}
```

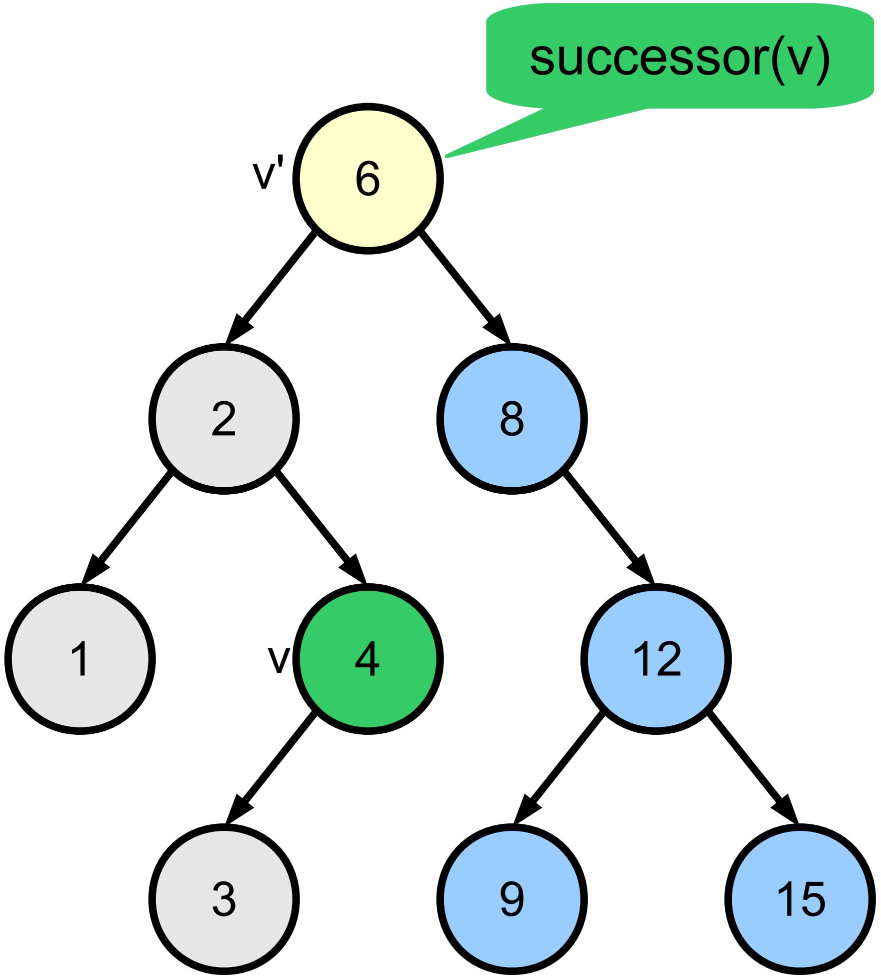
Search of the successor element

- Definition
  - A successor of a node v is the node containing the smallest value greater than v
- Two possible cases
  - v has a right child
  - The successor is the min of the right subtree of v
  - Example
    successor of 2 is 3



Search of the successor element

- Definition
  - A successor of a node v is the node containing the smallest value greater than v
- Two possible cases
  - v does not have a right child
  - The successor is the first ancestor v' such that v is in the left subtree of v'
  - Example
    successor of 4 is 6

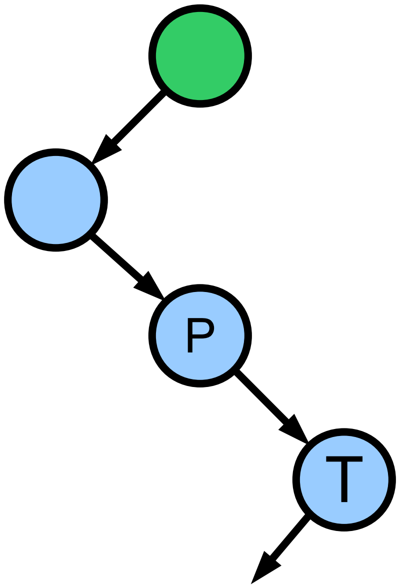


Search of successor
Pseudo-code (iterative)

Case 1: right child exists

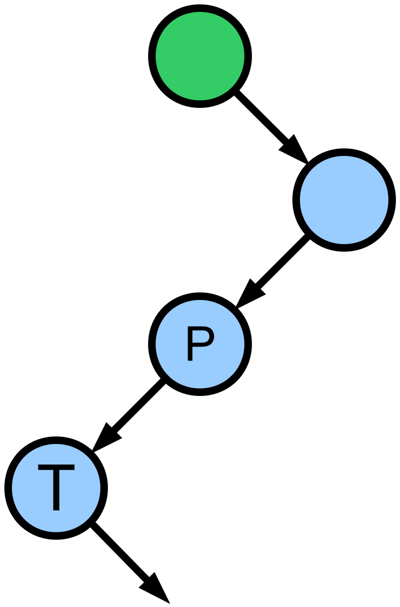
Case 2: right child missing

```
algorithm BST successor(BST T)
  if (T == null) then
    return null;
  endif
  if (T.right ≠ null) then
    return min(T.right);
  else
    P := T.parent
    while (P ≠ null && T == P.right) do
      T := P;
      P := P.parent;
    endwhile
    return P;
  endif
```



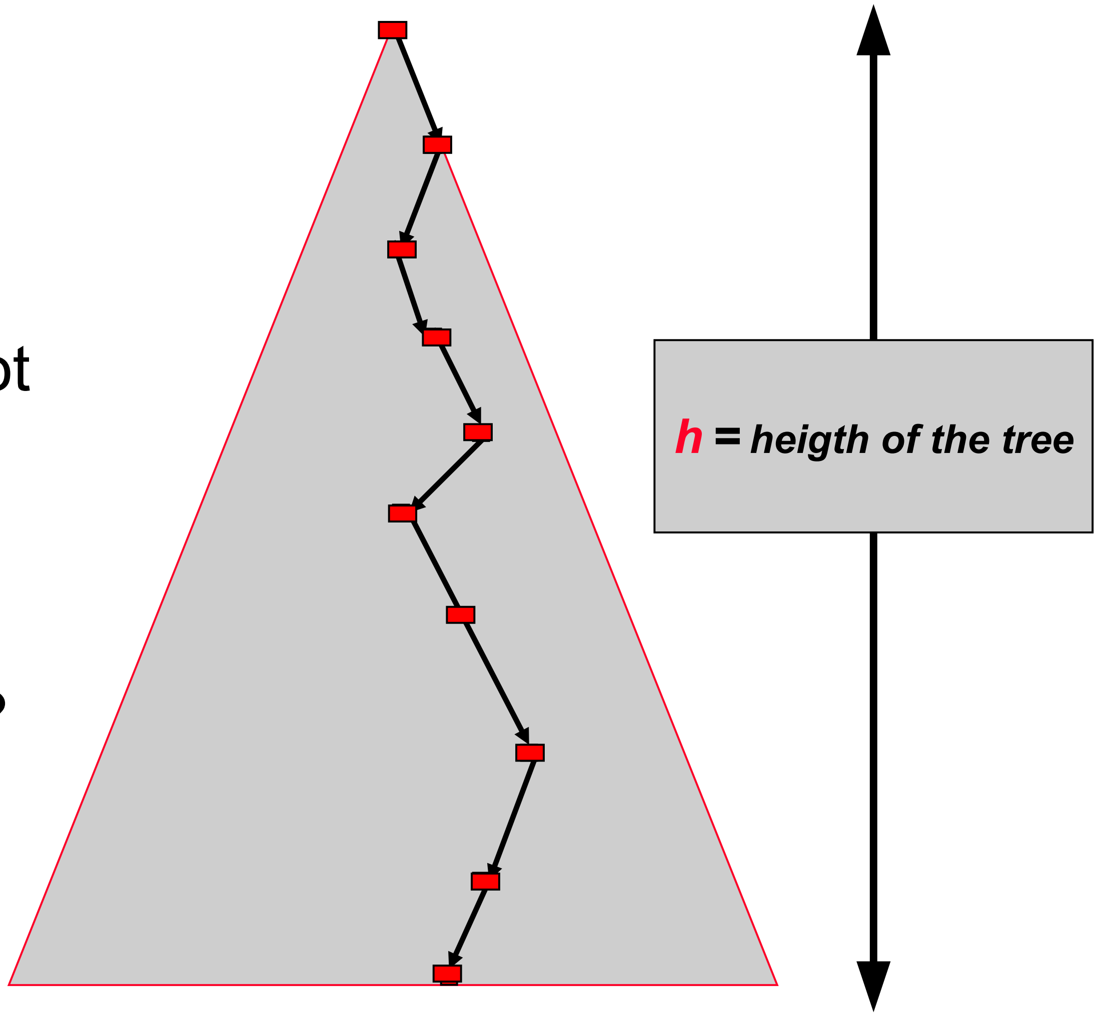
Search of predecessor Pseudo-code (iterative)

```
algorithm BST predecessor(BST T)
  if (T == null) then
    return null;
  endif
  if (T.left ≠ null) then
    return max(T.left);
  else
    P := T.parent;
    while (P ≠ null && T == P.left) do
       T := P;
       P := P.parent;
    endwhile
    return P;
  endif
```

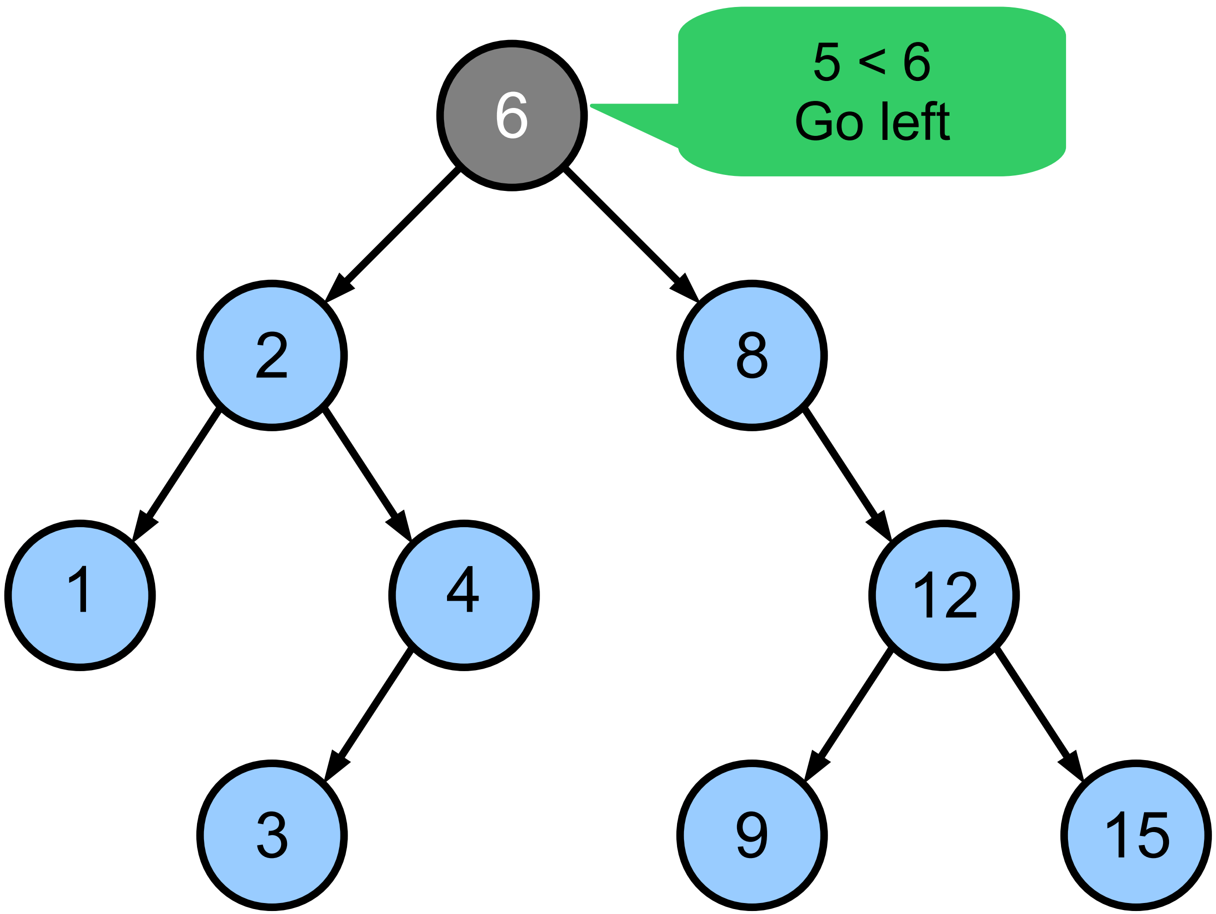


Search: computational cost

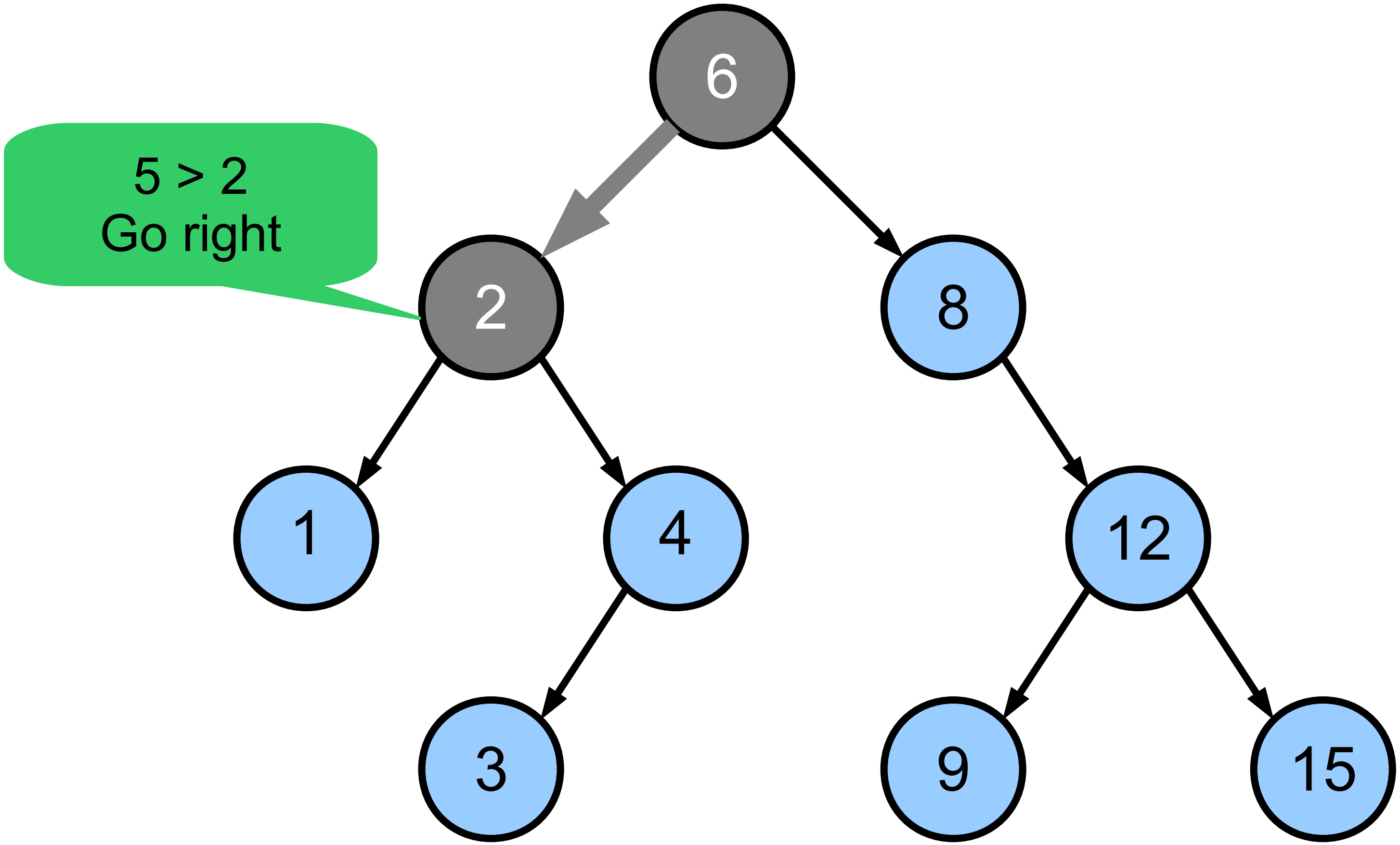
- In general
  - Search operations are limited to positions along a single path from the root to the leaf
  - Time needed: O(h)
- question
  - Which is the Worst case?
- question
  - Which is the best case?



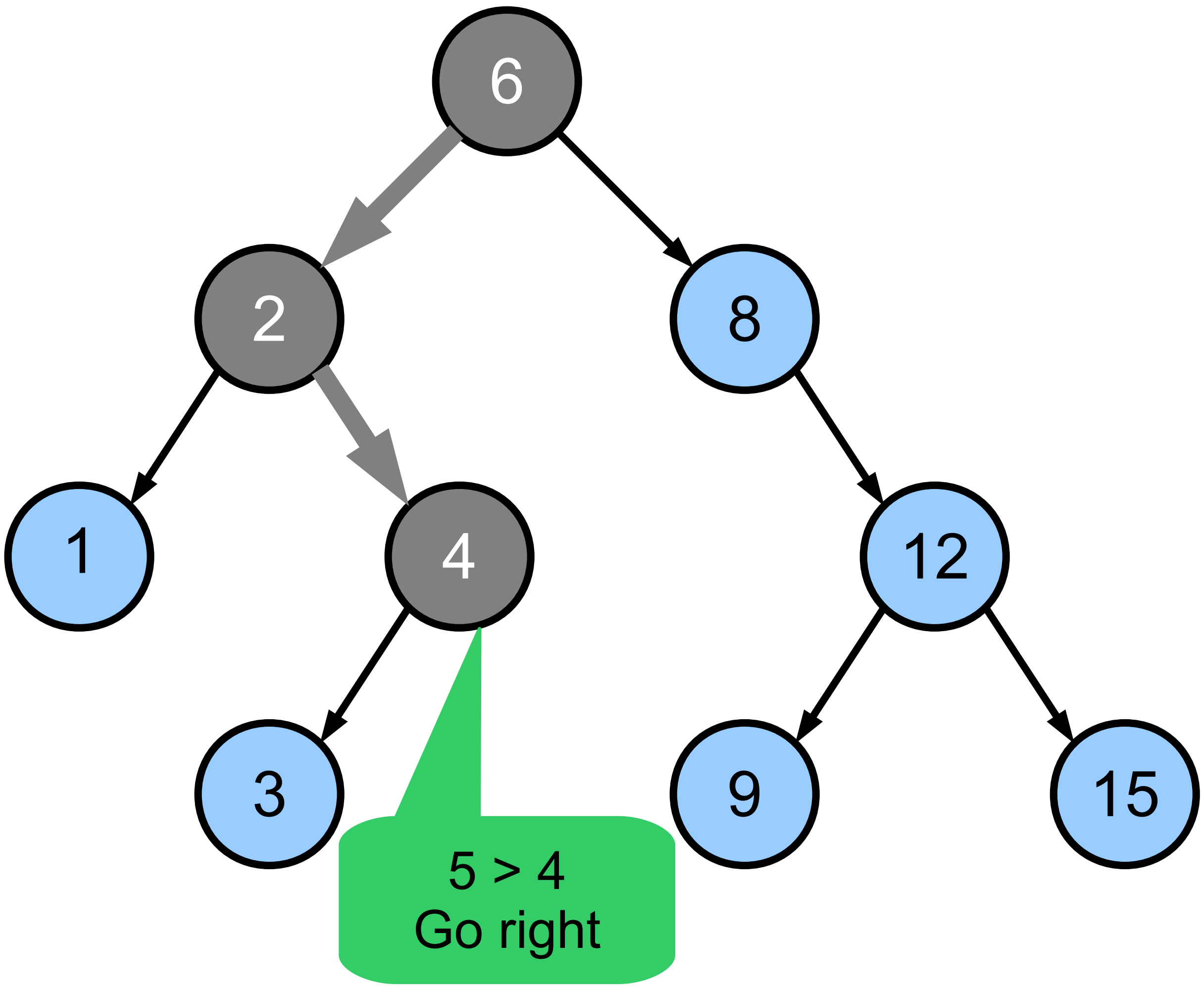
Insertion
Insertion of value 5



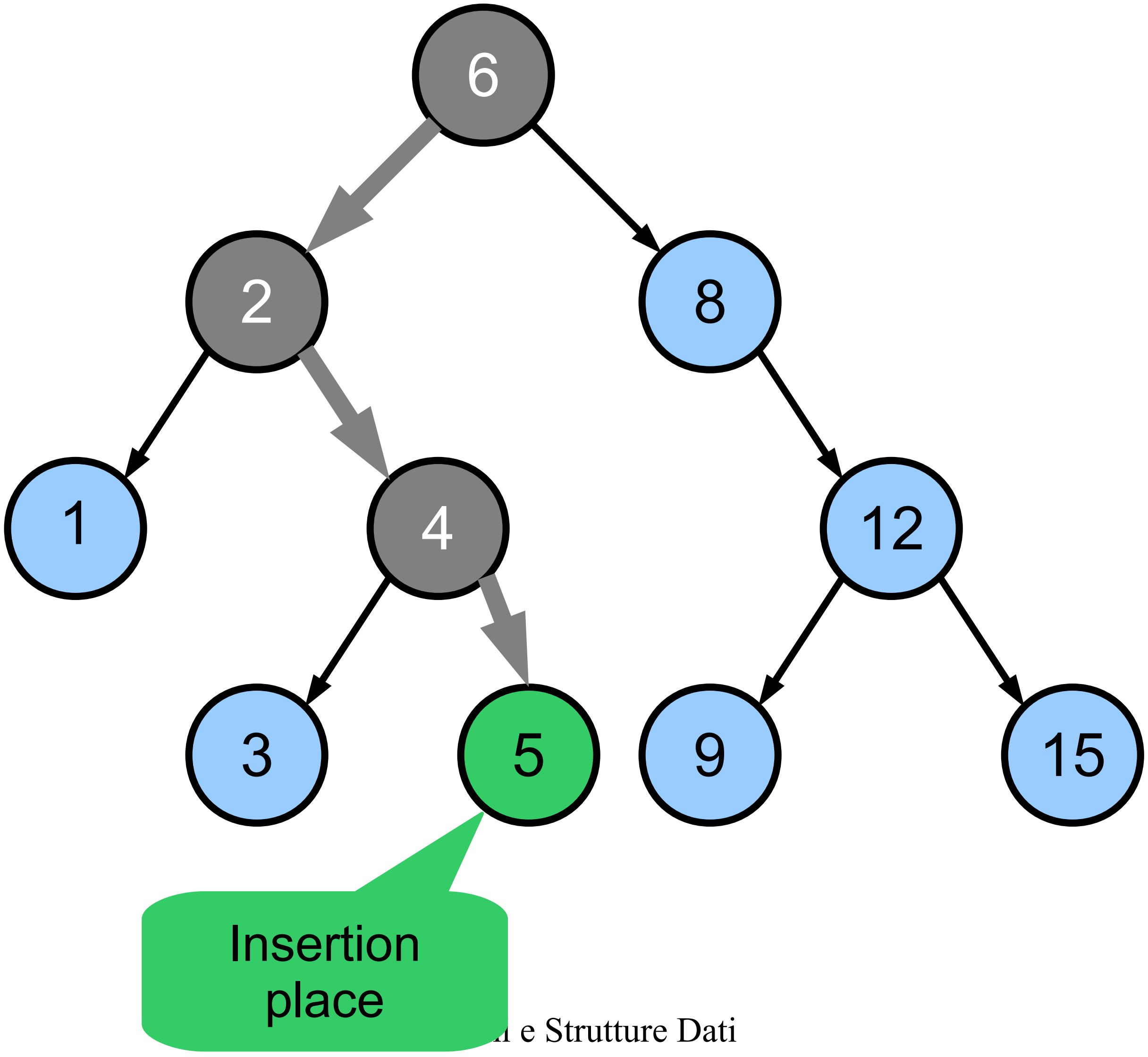
Insertion
Insertion of value 5



Insertion
Insertion of value 5



Insertion
Insertion of value 5



Insertion: pseudo-code (iterative)

```
Algorithm BST_insert(BST T, Key k, Data d)
  P := nil
  while (T != null) do
    P := T
    if T.key > key then
      T := T.left
    else
      T := T.right
    endif
  endwhile
  N := new BST(k,d)
  N.parent := P
  if (P == null) then
    return N;
  if (k < P.key) then
    P.left := N
  else
    P.right := N
```

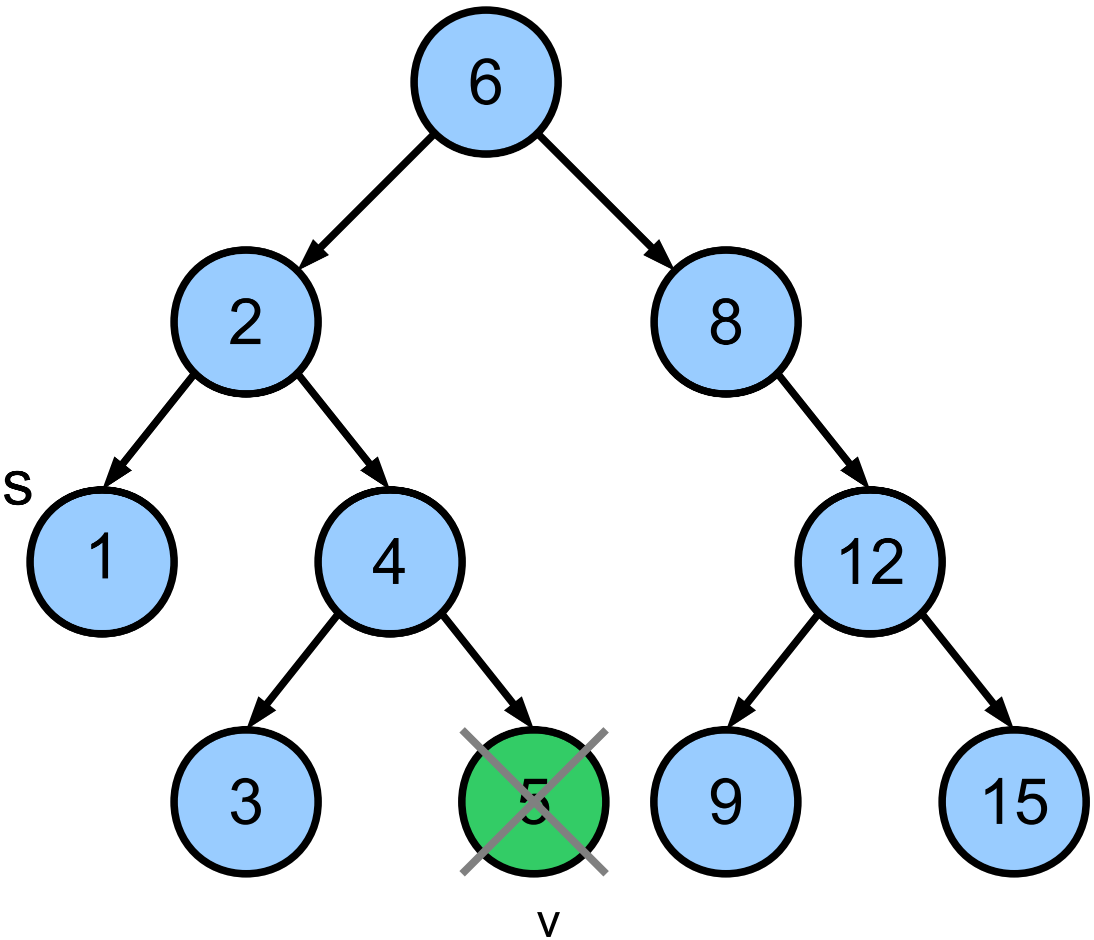
Search position of new node

Base case (insert in empty tree)

General case

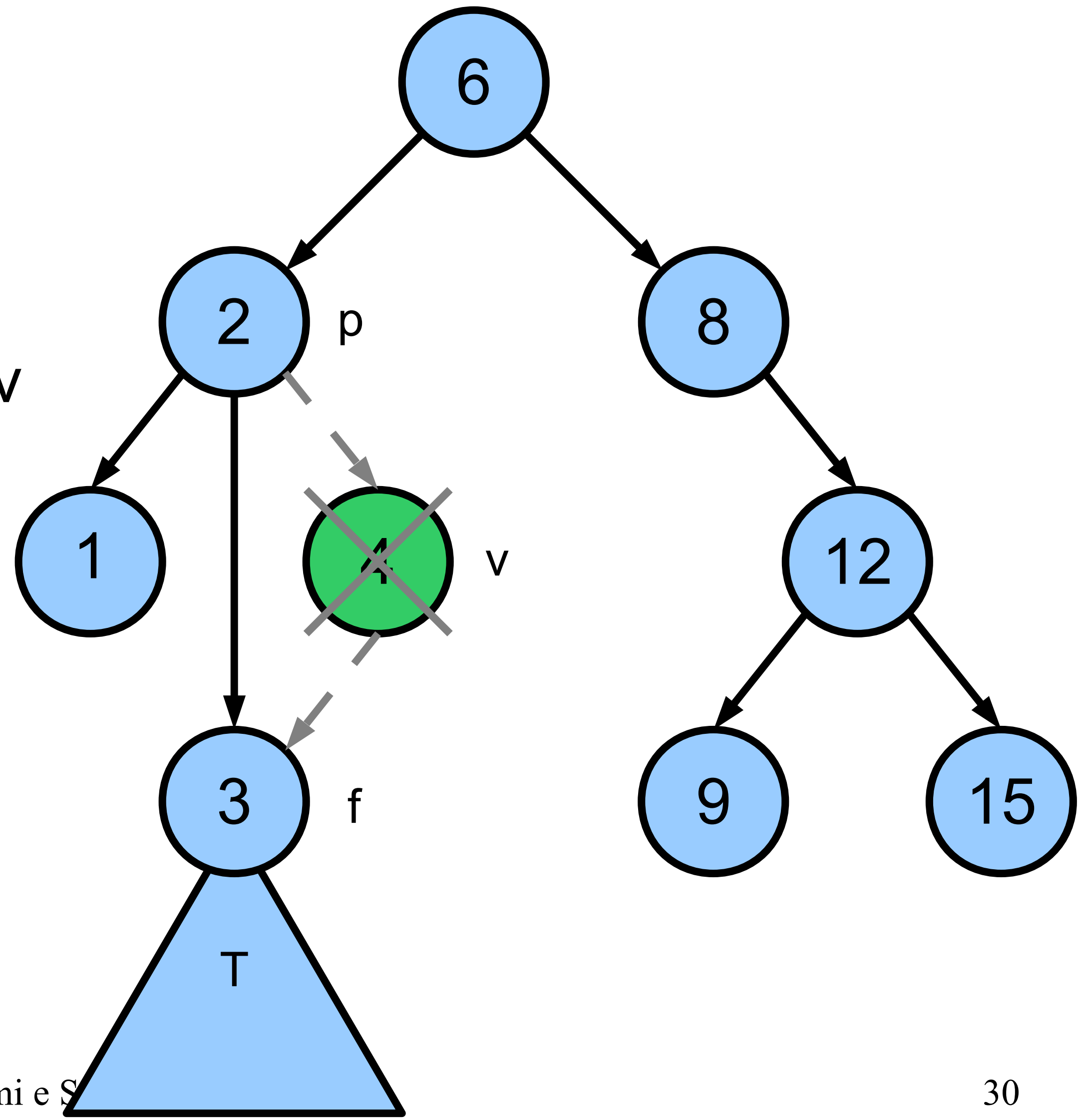
deletion

- Case 1
  - Deleted node v has no children
  - Simply delete it....
- Correctness?
  - Deleting a leaf node does not modify the ordering property of any other node in the BST



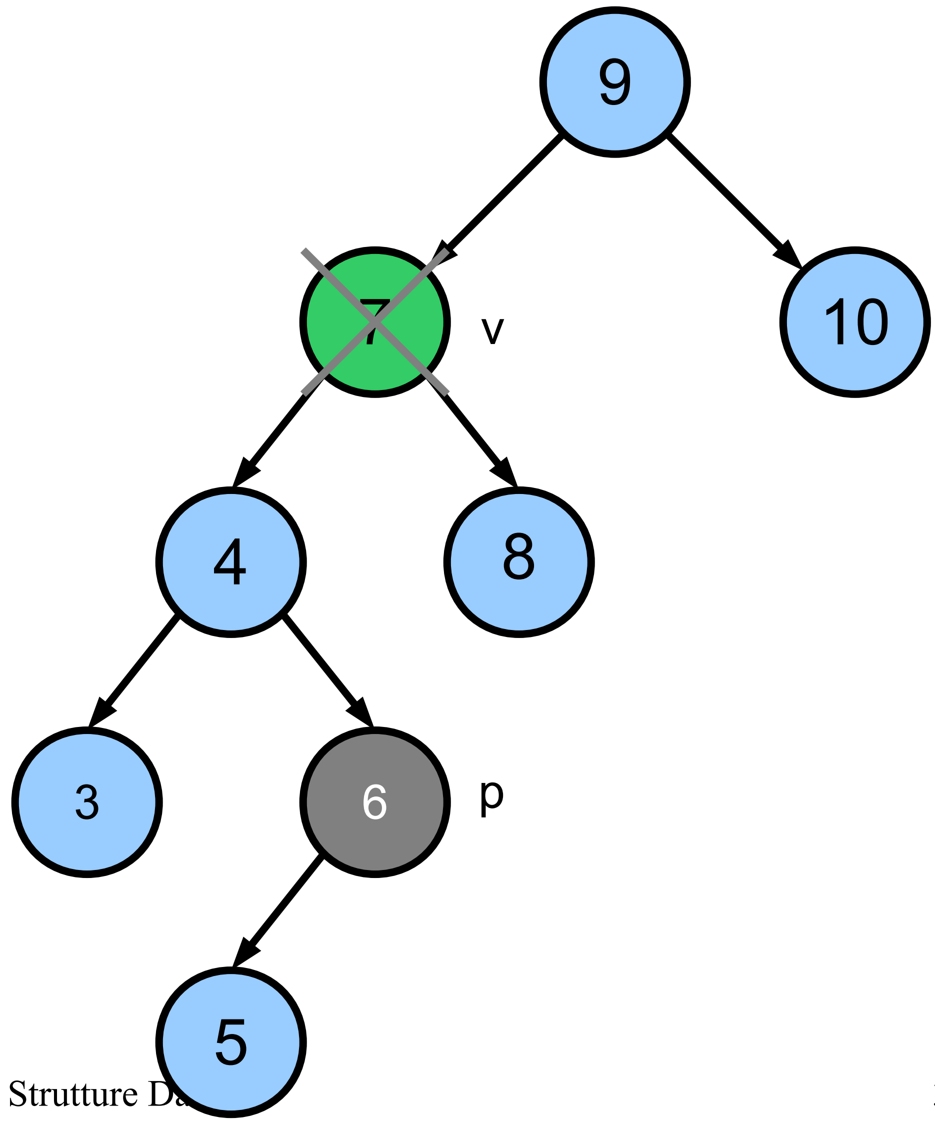
deletion

- Case 2
  - Deleted node v has only one child f
  - delete v
  - attach f to ex-father p of v in substitution of v
- Correctness
  - Given the ordering property, all the key values in subtree T are ≥ p



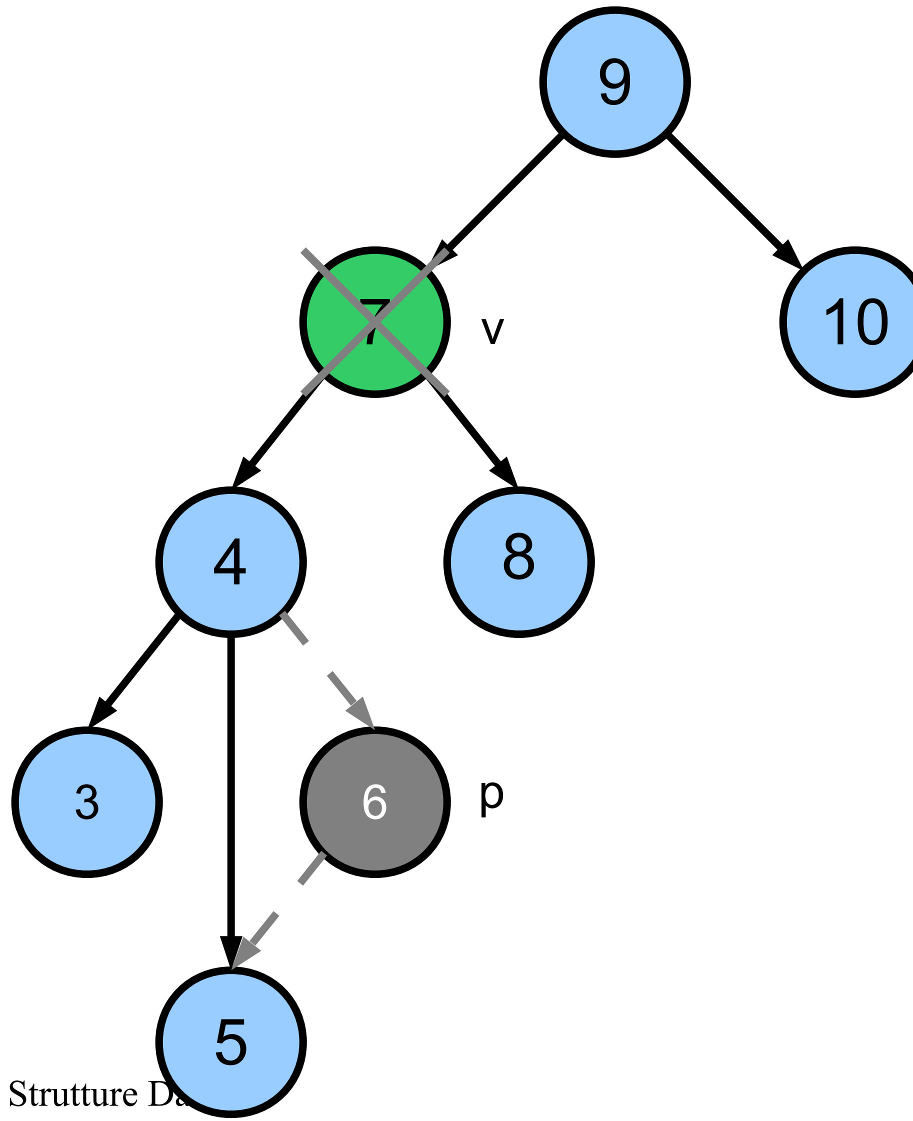
deletion

- Case 3
  - Deleted node v has two children
  - Search predecessor p of v
  - The predecessor p has not a right child
    - why?



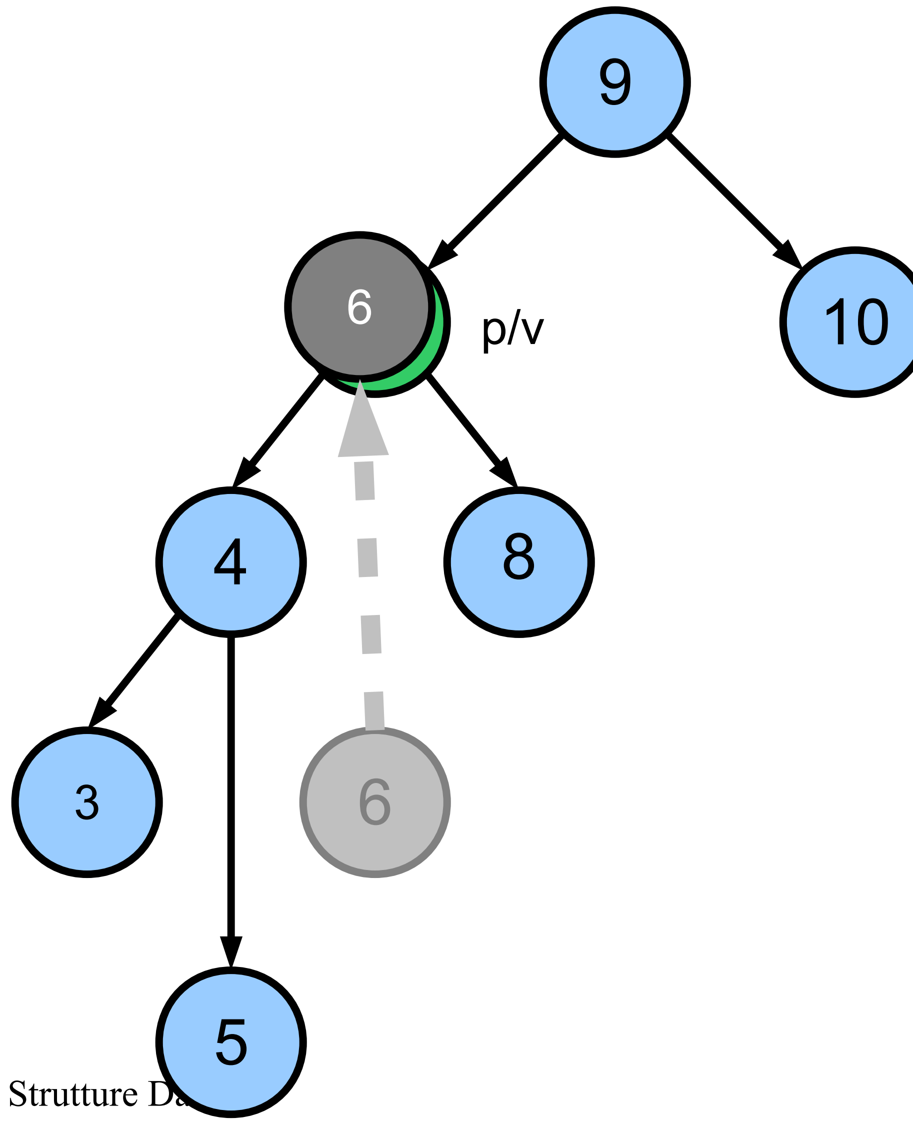
deletion

- Case 3
  - Deleted node v has two children
  - Search predecessor p of v
  - The predecessor p has not a right child
  - Detach the predecessor
  - Attach the (if existing) left child of p to the father of p



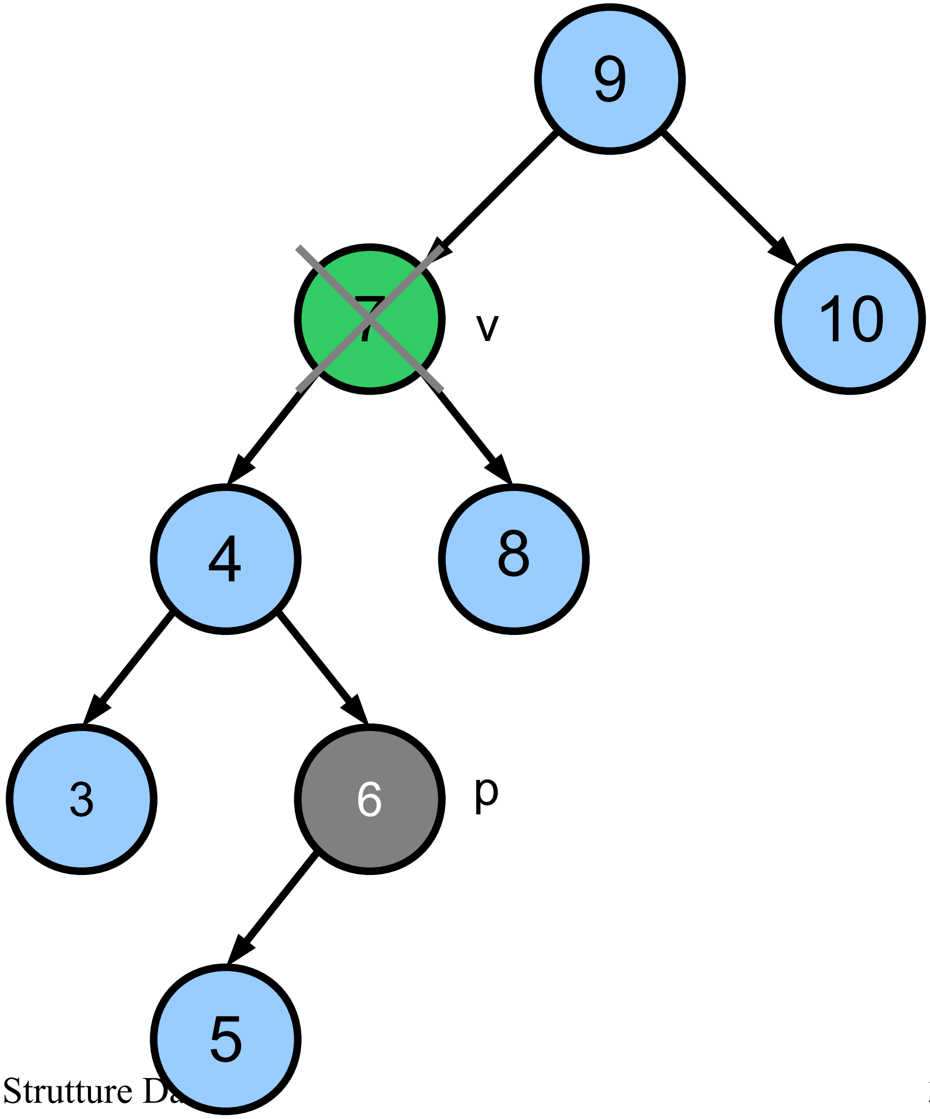
deletion

- Case 3
  - Deleted node v has two children
  - Search predecessor p of v
  - The predecessor p has not a right child
  - Detach the predecessor
  - Attach the (if existing) left child of p to the father of p
  - Copy p on v's position



deletion

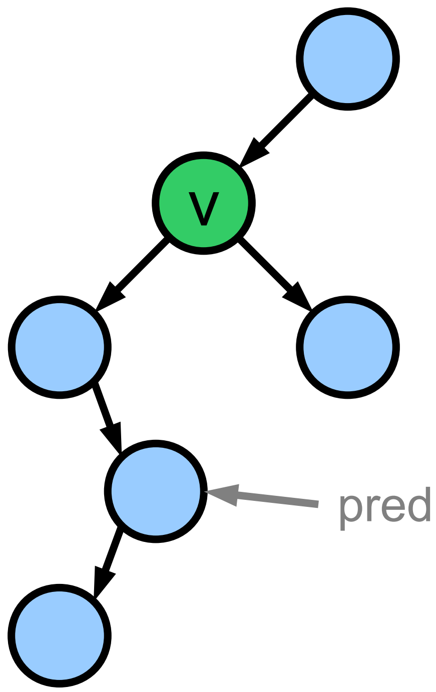
- Case 3 (correctness)
- predecessor p of v
  - Is certainly ≥ of all nodes in left subtree of v
  - Is certainly ≤ of all nodes in right subtree of v
- Then it CAN be substitute of v



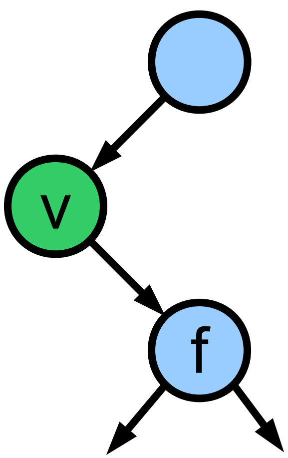
e.g. Java implementation

```
protected Node delete(InfoBR i) {
    Node v = i.node;
    if (tree.degree(v) == 2) {
        Node pred = max(tree.sx(v));
        exchangeInfo(v, pred);
        v = pred;
    }
    Node p = tree.father(v);
    compactNode(v);
    return p;
}
```

Delete node v (may have more than one child)



e.g. Java implementation




```
protected void compactNode(Node v){
    Node f = null ;
    if (tree.sx(v) != null)
        f = tree.sx(v);
    else
        if (tree.dx(v) != null)
            f = tree.dx(v);
    if (f == null)
        tree.cut(v);
    else {
        exchangeInfo(v, f);
        BinTree a = tree.cut(f);
        tree.innestSx(v, a.cut(a.sx(f)));
        tree.innestDx(v, a.cut(a.dx(f)));
    }
}
```

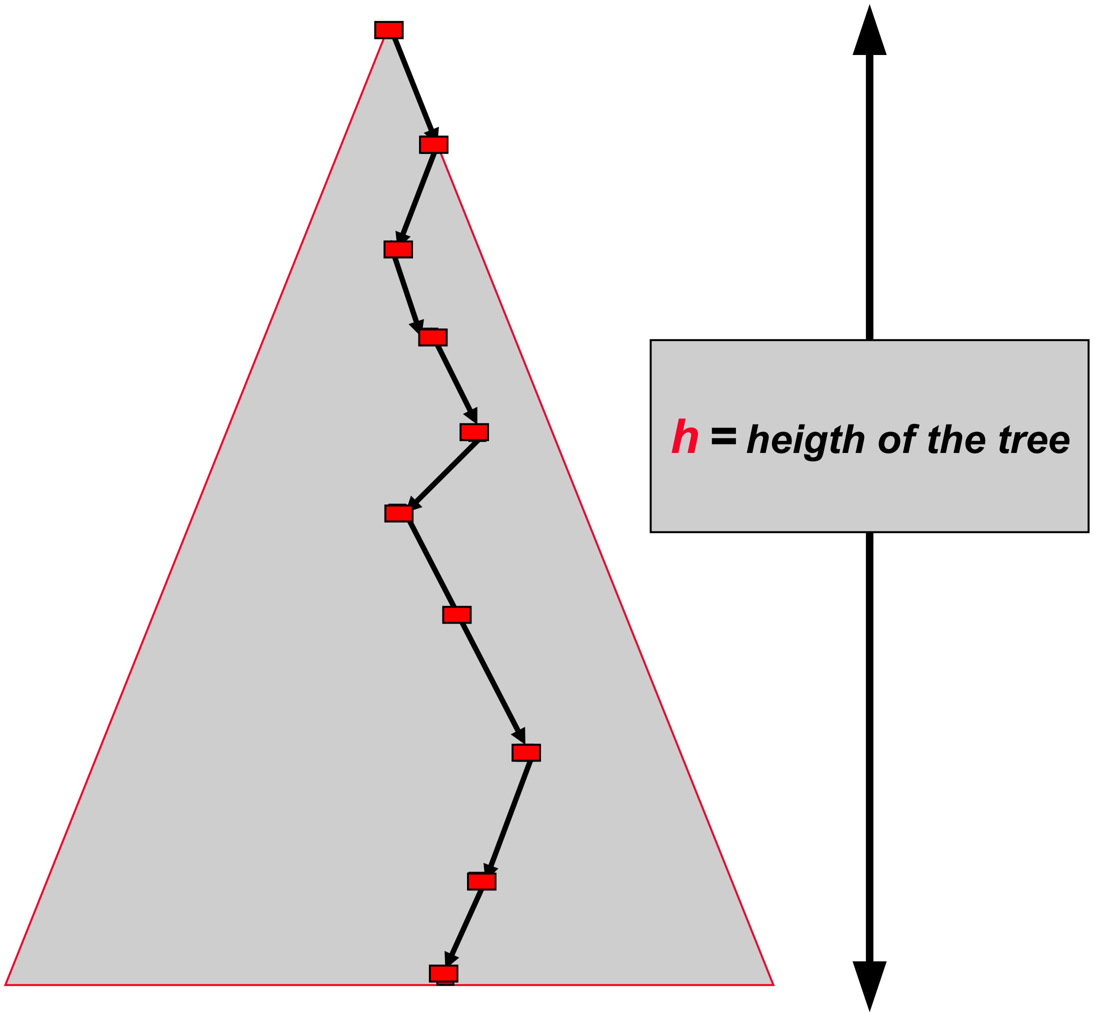
Delete node v (has at most one child)

v has no children

Detach subtree rooted in f

Modify: computational cost

- In general
  - Modify operations are located in positions along a single path from root to leaf
  - Time complexity: O(h)



Average complexity

- What is the average heigth of a BST?
  - General case (generalized insertions + deletions)
    - Hard to define
  - Simple case: random insertions
    - We can show that average heigth is O(log n)
- In general:
  - We need techniques to keep the tree balanced
  - Example os such implementations
    - AVL tree (Adelson-Velsky, Landis)
    - Red-Black Tree
    - Splay Tree
  - We will see in the next slides.

Some nice exercises

- Exercise
  - Write a non recursive algorithm to visit (in-order) a BST
- Exercise
  - Proof that any algorithm based on comparisons to realize a BST is $\Omega(n \log n)$
    - hint: sorting based on comparison of n elements is $\Omega(n \log n)$, so ...
- Exercise
  - Proof that if a BST node has two children, then the successor has not left child, and the predecessor has not right child

More exercises

- Exercise
  - The visit of a BST can be done by finding the minimum element and then calling n-1 times the successor().
  - Proof that this algorithm is $\Theta(n)$
- Exercise
  - Write a recursive version of insert()
- Exercise
  - Is the BST deletion property commutative? In other words, delete(x), delete(y) provides the same result as delete(y), delete(x) for any x,y?