Estruturas de Informação

Trees

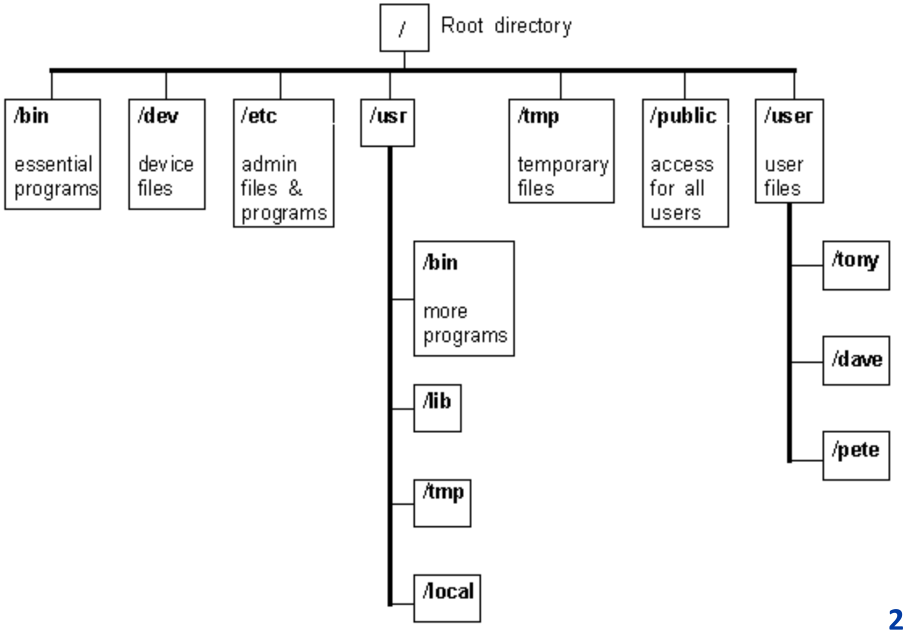
Fátima Rodrigues
mfc@isep.ipp.pt
Departamento de Engenharia Informática (DEI/ISEP)

Trees

- In computer science, a tree is an ADT which stores elements hierarchically
- Tree consists of nodes with a parent-child relation

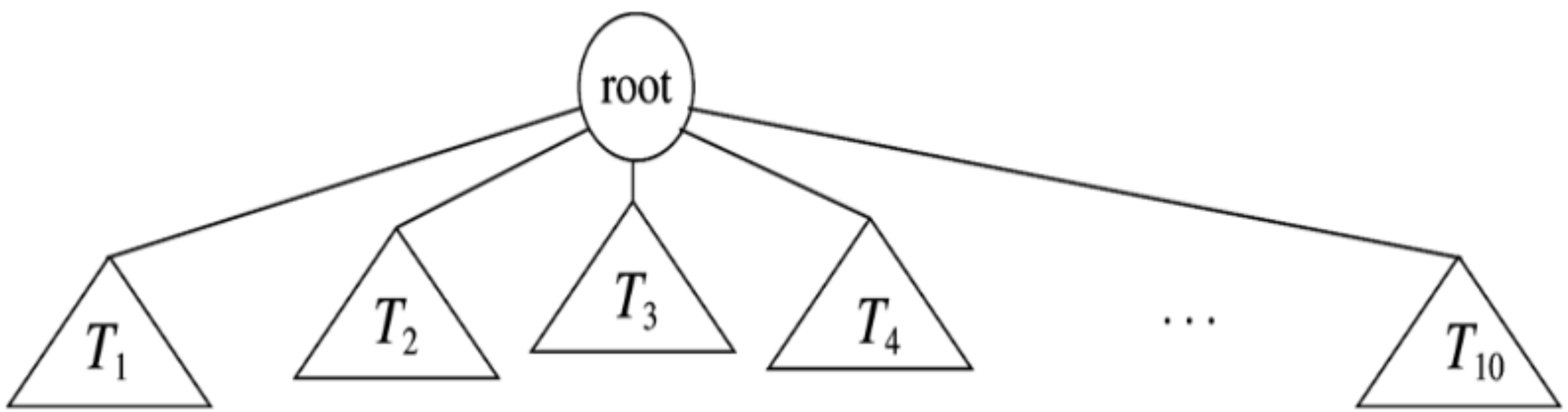
Applications:

- File systems
- Programming environments
- Taxonomies
- Image Representation
- Database Indexes
- ….



Tree – Definition

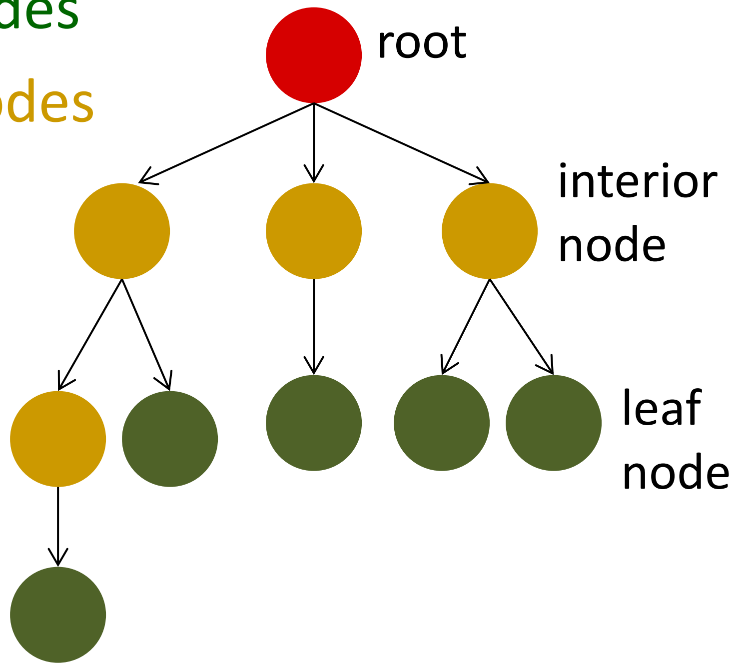
- A tree is a set of nodes that may be empty
- If not empty, then there is a distinguished node r, called root and zero or more non-empty subtrees T_1, T_2, ... T_k, each of whose roots are connected by a directed edge from r
- Every node in a tree is the root of a subtree
- Each node of the tree, different from the root, has a unique parent node



Generic tree

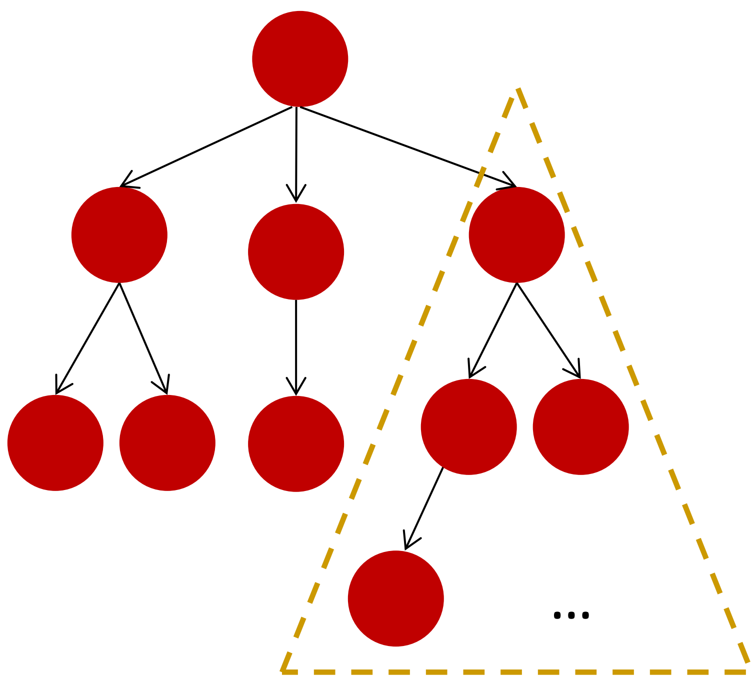
Tree Terminology

- Tree = Set of nodes connected by arcs (or edges)
- Every tree has a single root node – node without parent node
- A parent node points to (one or more) other nodes
- Nodes pointed to are children
- Every node (except the root) has exactly one parent
- Nodes with no children are leaf nodes
- Nodes with children are interior nodes



Tree Terminology

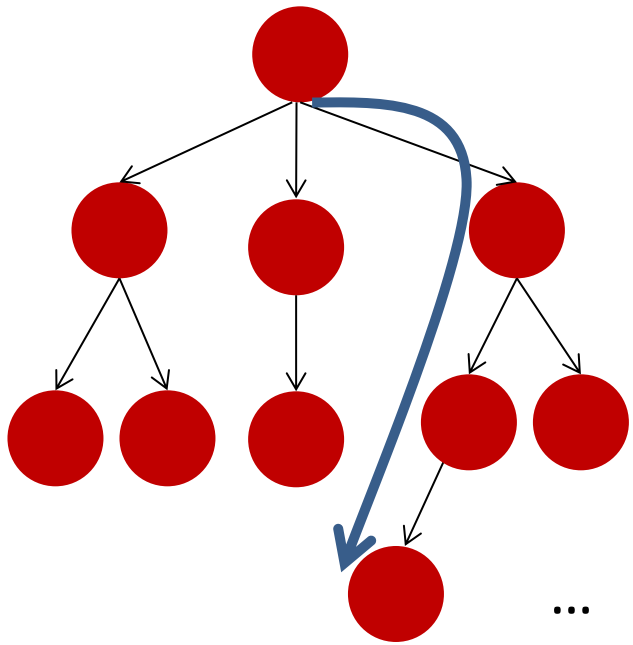
- Any node can be considered the root of a subtree



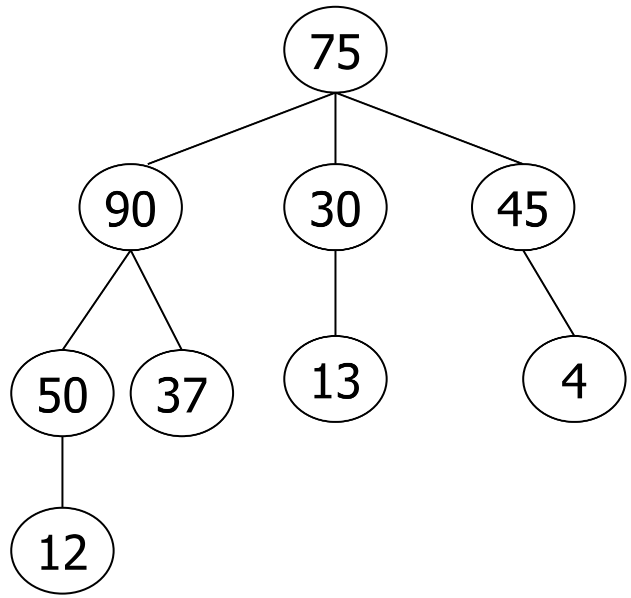
How many subtrees are there?

Tree Terminology

- There is a single, unique path from the root to any node
- A path's length is equal to the number of arcs traversed



Ascendants and descendants of a Node

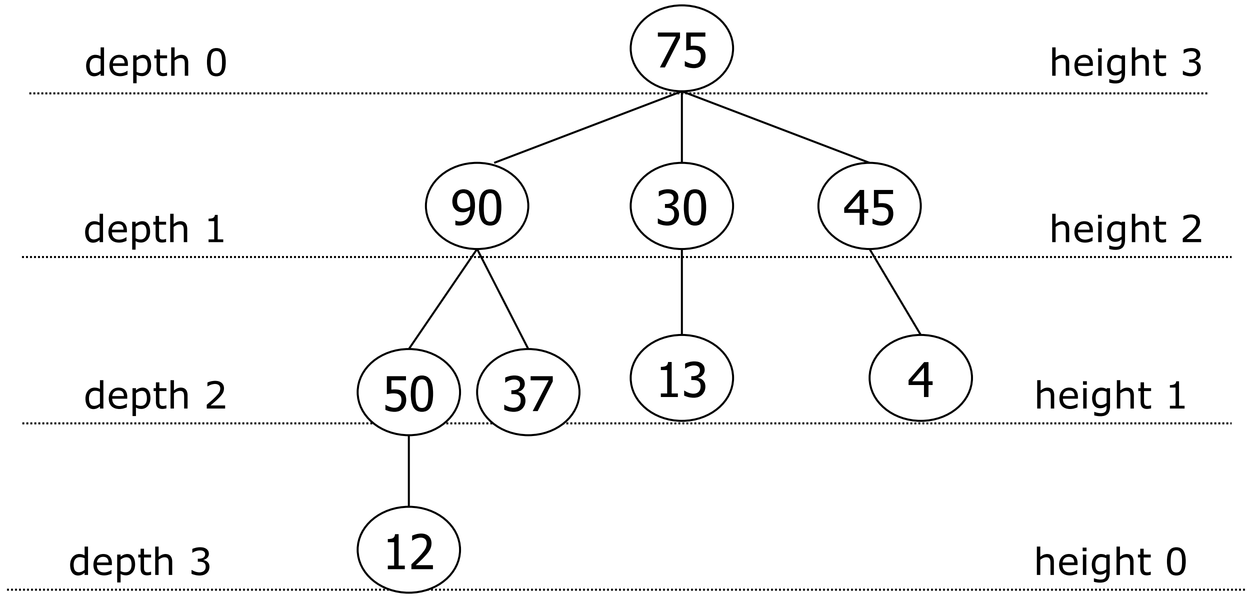


- The ascendants of a node are the nodes that are in the path from the node to the root of the tree. Ascendants of node 12: 50, 90, 75
- The descendants of a node are all the nodes reachable from that node. Descendants of node 90: 50, 37, 12
- All nodes in a tree are descendants of the root

  (except for the root)

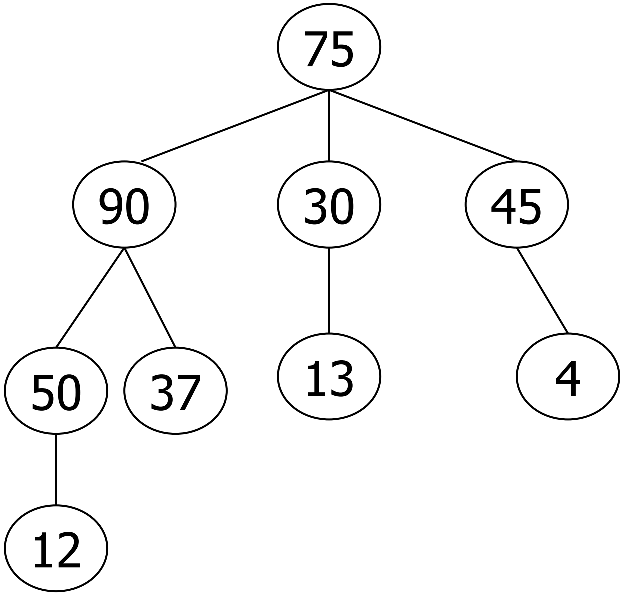
Height and depth of a tree

- Height of a node = max. path length from the node to a leaf
  - Height of a leaf node = 0 (greater depth)
  - Height of the tree = Height of the root
- Depth of a node = path length from the root to that node
  - Depth of the root = 0



Degree of a tree

- Degree of a tree is the maximum degree of its nodes
- Degree of a node is equal the number of its children's



Degree of node 90: 2
Degree of a leaf node: 0
Degree of the tree: 3

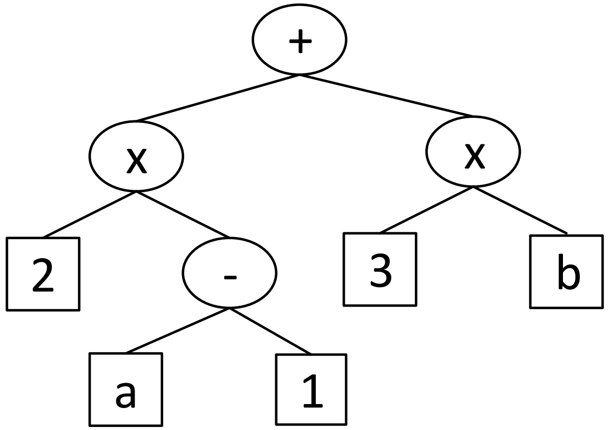
Binary Trees

Binary Tree – Definition

- A binary tree is a special case of a K-ary tree whose nodes have exactly two child references
- A binary tree is a rooted tree in which no node can have more than two children AND the children are distinguished as left and right

Applications:

- Arithmetic expressions
- Decision processes
- Searching
- ….



Arithmetic expression: ((2 x (a - 1)) + (3 x b))

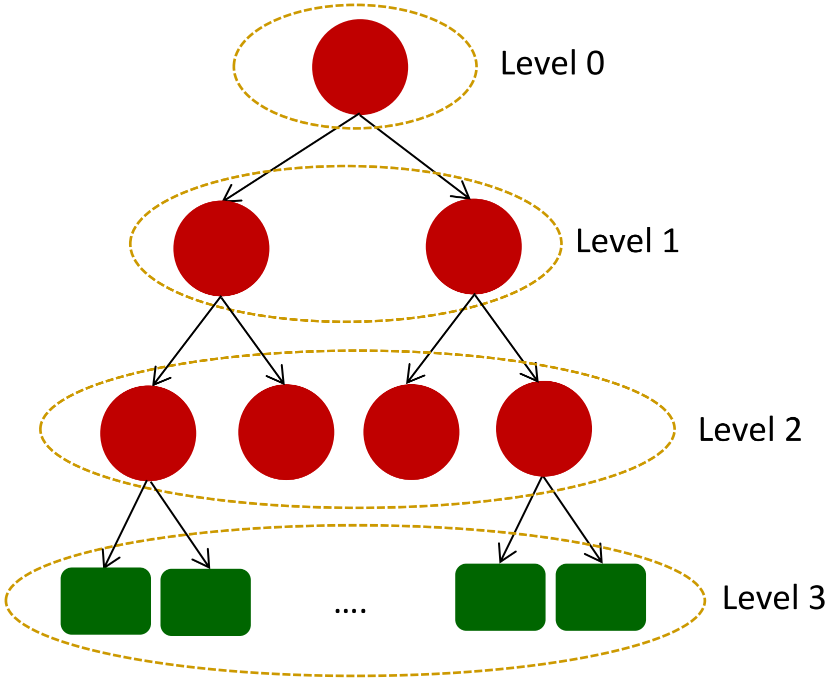
Binary Tree – Properties

In a binary tree

- level 0 has at most 1 = 2^0 node
- level 1 has at most 2 = 2^1 nodes
- level 2 has at most 4 = 2^2 nodes

…

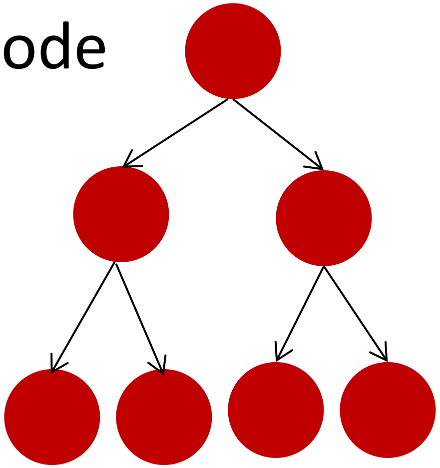
- level d has at most 2^d nodes



A binary tree of height h has:

- minimum: h + 1 nodes
- maximum: $2^{h+1} - 1$ nodes

Binary Tree – Properties



A full binary tree is a binary tree in which every node is a leaf or has exactly two children

- A full binary tree with n internal nodes has n + 1 leaves

A perfect binary tree is a full binary tree in which all leaves have the same depth

- The number of nodes in a perfect binary tree is $2^{h+1} - 1$ nodes, where h is height

$$n = 2^{h+1} - 1$$

$$2^{h+1} = n + 1$$

$$\log_2 (2^{h+1}) = \log_2 (n + 1)$$

$$h = \log_2 (n + 1) - 1$$

Tree Traversals

Depth-First Traversals

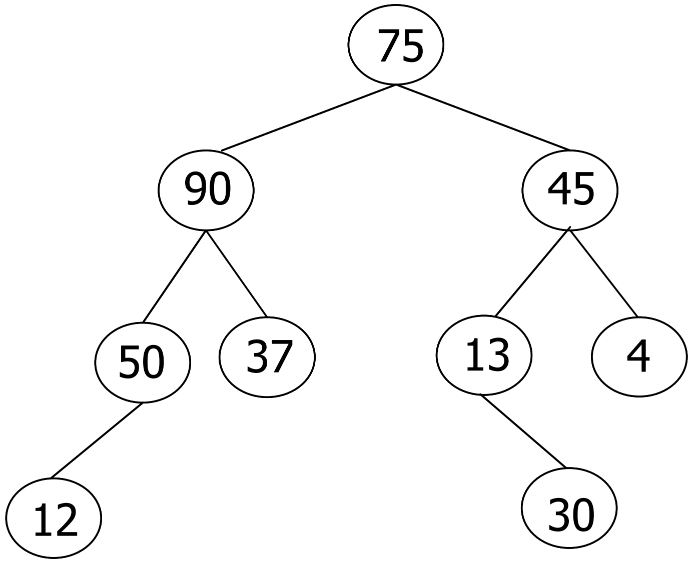
- Pre-order – root, left subtree, right subtree
- In-order – left subtree, root, right subtree
- Pos-order – left subtree, right subtree, root

Breadth-First Traversal

- Level-order – all the positions at depth *d are visited* before the positions at depth $d + 1$

- A breadth-first traversal is a common approach used in software for playing games

Tree Traversals - Exemplification



Depth-First

- Pre-order 75, 90, 50, 12, 37, 45, 13, 30, 4
- In-order 12, 50, 90, 37, 75, 13, 30, 45, 4
- Pos-order 12, 50, 37, 90, 30, 13, 4, 45, 75

Breath-First

- Level-order 75, 90, 45, 50, 37, 13, 4, 12, 30

Pre-Order Traversal

In a preorder traversal the node is visited before both its subtrees, left and right

```
Algorithm void preOrder(Node<E> node){

    if (node == null)
        return;

    visit(node)
    preOrder(node.getLeft())
    preOrder(node.getRight())
}
```

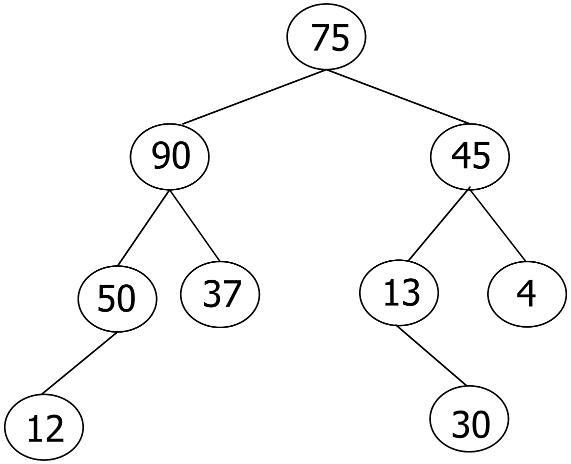
Time Complexity: O(?)

If visit(node) is O(1), then the complexity of preOrder is O(n)

Pre-Order Traversal – iterative algorithm

The iterative algorithm needs an auxiliary stack

```
Algorithm void IterativepreOrder(Node<E> node){

  r = node
  do {
       while (r != null){
          visit(r)
          stk.push(r)
          r=r.getLeft()
       }
       if (!stk.isEmpty()){
          stk.pop()
          r=r.getRight()
       }
   } while (stk.isEmpty() && r != null)
}
```



In-Order Traversal

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

```
Algorithm void inOrder(Node<E> node){

  if (node == null)
     return;

  inOrder(node.getLeft())
  visit(node)
  inOrder(node.getRight())
}
```

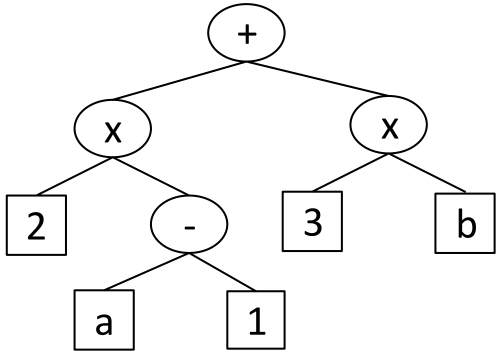
Specialization of In-Order Traversal:

writing of an arithmetic expression

Specialization of In-Order Traversal

Write an Arithmetic Expression

- write operand or operator when visiting node
- write “(“ before traversing left subtree
- write “)“ after traversing right subtree



((2 x (a - 1))+(3 x b))

```
Algorithm void  writeExpression(Node<String> node, String str){

   if (node.getLeft()){
      str += “(“
      writeExpression(node.getLeft(),str)
   }
   str += node.getElement()
   if (node.getRight()){
      writeExpression(node.getRight(),str)
      str += “)“
   }
}
```

Pos-Order Traversal

In a pos-order traversal a node is visited after both its subtrees, left and right

```
Algorithm void posOrder(Node<E> node){

  if (node == null)
     return;

  posOrder(node.getLeft())
  posOrder(node.getRight())
  visit(node)
}
```

Iterative Algorithm: needs two stacks

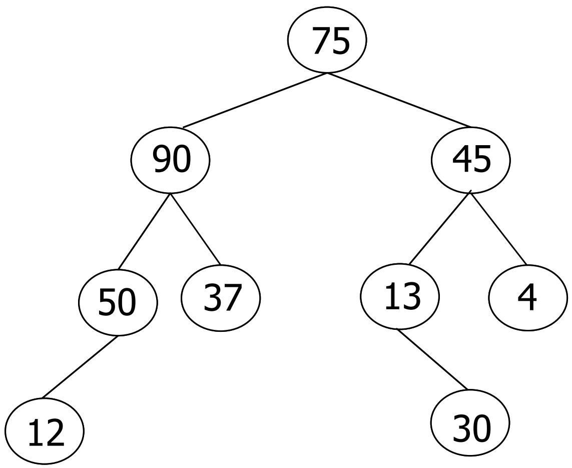
Specialization of a Pos-Order Traversal:

evaluation an arithmetic expression

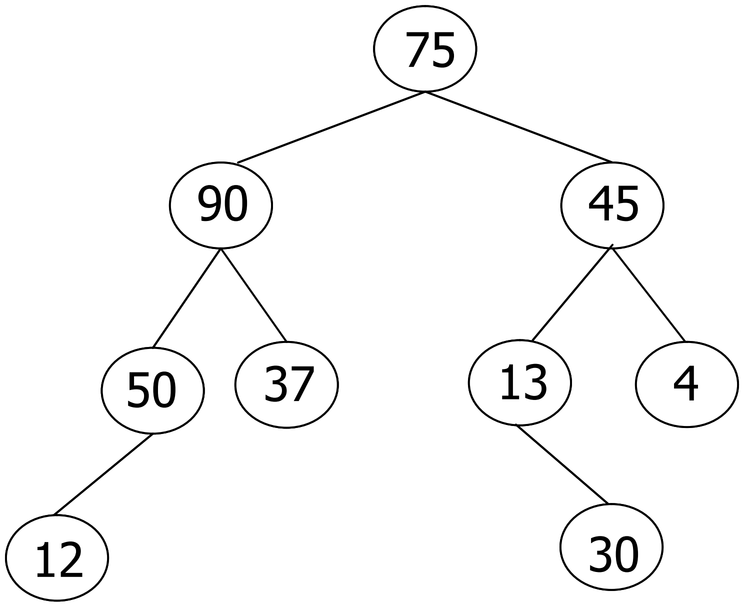
Breadth First Traversal

In a breadth first traversal the nodes of the tree are visited level by level

```
Algorithm void breadthfirst (){
   Initialize queue Q to contain root()
   while (Q not empty) {
      p = Q.dequeue()
      visit(p)
      for (each child c in children(p))
         Q.enqueue(c)
   }
}
```



Search an Element



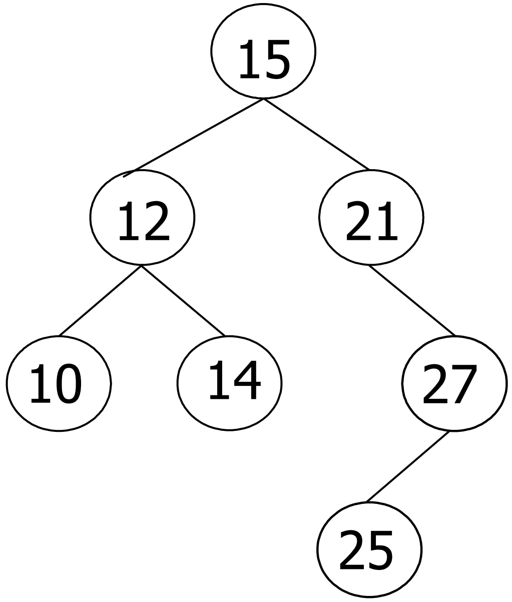
Time Complexity: O(?)

Binary Search Trees

Binary Search Tree (BST)

Is a binary tree where every node value is:

- **Greater than** all its left descendants
- **Less than** to all its right descendants



The elements in the BST must be comparable

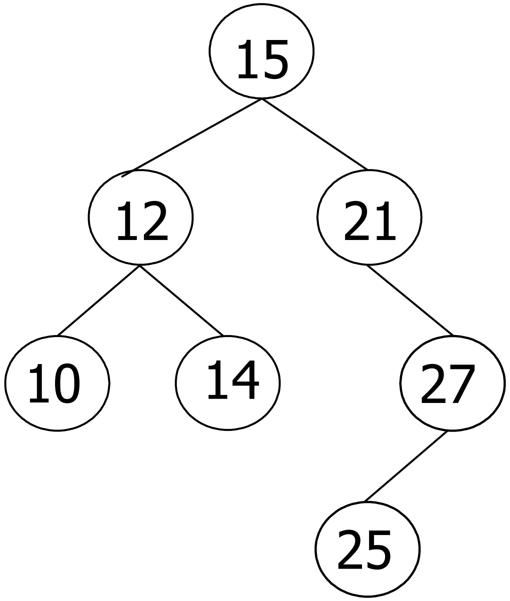
Duplicates are not allowed

Each subtree of a BST is also a BST

Applications:

- Symbol tables in compilers, "assemblers“
- Used in implementing efficient priority-queues (heaps)
- ….

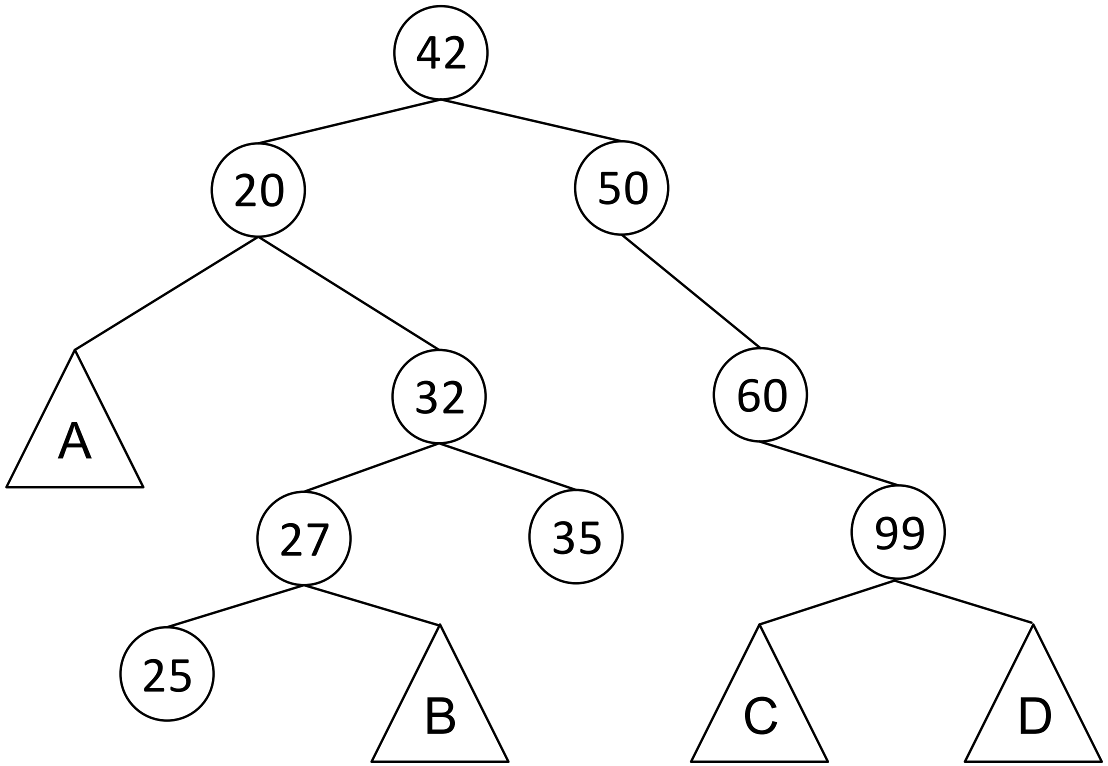
Binary Search Tree



- In-order: 10, 12, 14, 15, 21, 25, 27
- Pre-order: 15, 12, 10, 14, 21, 27, 25
- Pos-order: 10, 14, 12, 25, 27, 21, 15
- Level-order: 15, 12, 21, 10, 14, 27, 25

BST in-order traversal returns elements in sorted order

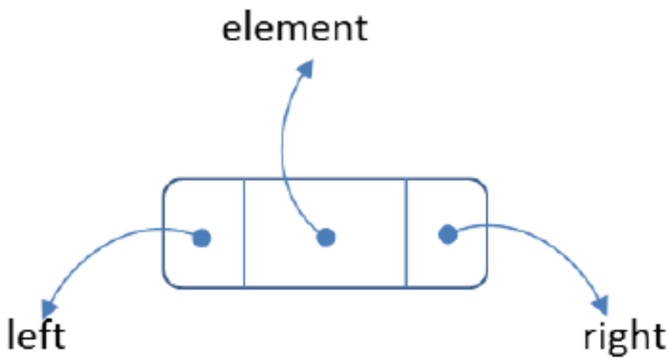
A Binary Search Tree of Integers



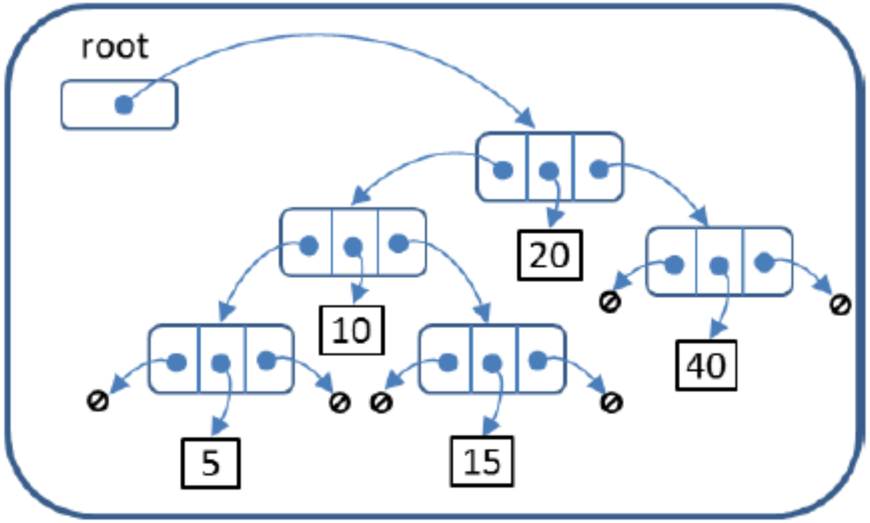
Describe the values which might appear in the subtrees labeled A, B, C, and D

Binary Search Tree ADT

Node

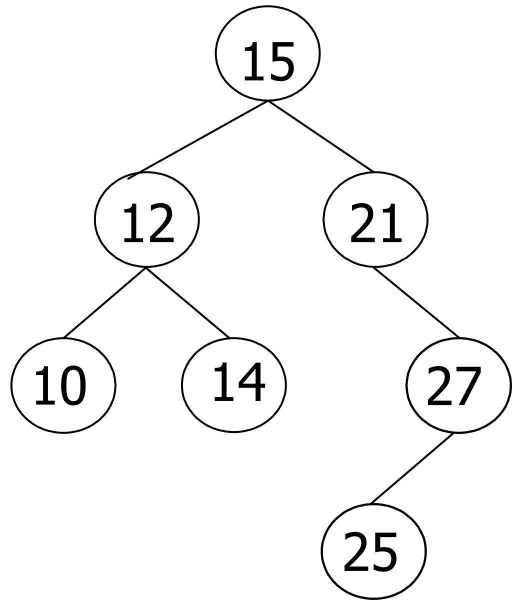


BST




```java
public BST()
public boolean isEmpty()
public int size()
public void insert(E element)
public void remove(E element)
```

Search for an Element



- Start at root
- At each node, compare value to node value:
  - Return true if match
  - If value is less than node value, go to left child (and repeat)
  - If value is greater than node value, go to right child (and repeat)
  - If node is null, return false

Time Complexity

The maximum number of comparisons to conclude whether or not the key is in the tree is the maximum height of tree: h

If the tree is (more or less) balanced, all the leaf nodes with the same depth, the height of the tree can be relate with the total number of elements n

$$n = 2^{(h+1)} - 1$$
$$2^{(h+1)} = n + 1$$
$$h+1 = \log_2 (n+1)$$
$$h = \log_2 (n+1) - 1$$

For all values of $n \geq 1$, there is a constant C, such that:

$$\log_2 (n+1) - 1 \leq C \times \log_2 n$$

$T(n) = O(\log n)$

Search for an Element

```
Algorithm Node<E> search(Node<E> node, E elem){
   if (node == null)
      return null
   if (node.getElement() == elem)
      return node
   if (node.getElement() > elem)
      return search(node.getLeft(),elem)
   else
       return search(node.getRight(),elem)
}
```

Time Complexity: O(?)

- **Best case**: the element is at the root
- **Average Case**: the tree is balanced
- **Worst Case**: the element doesn't exist, the tree degenerates in a list

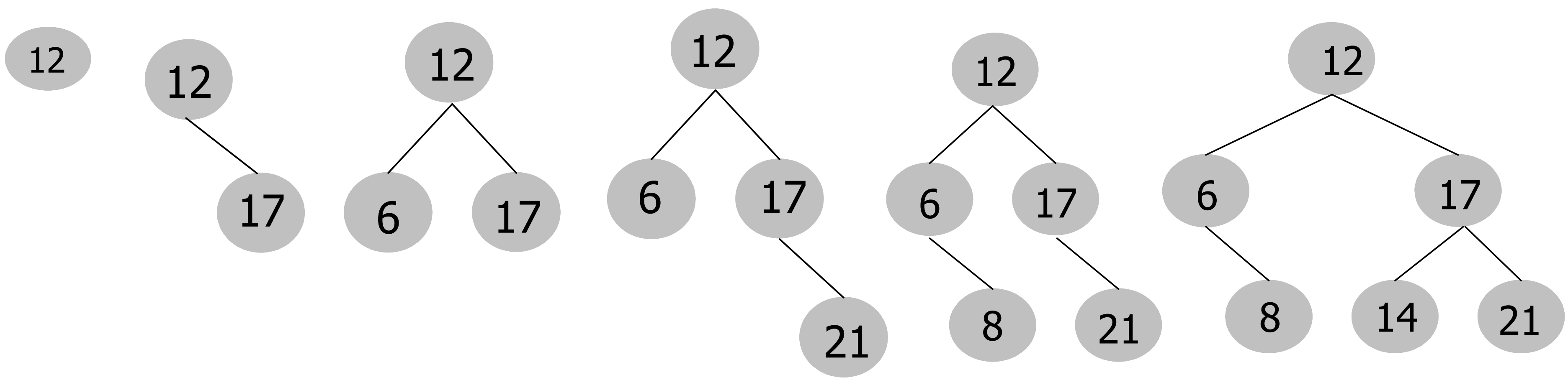
Search for an Element – iterative version

```
Algorithm boolean search(E elem) {
  node = root
  find = false
  while (node != null && !find){
     if (node.getElement() == elem)
        find = true
     if (node.getElement() > elem)
        node = node.getLeft()
     if (node.getElement() < elem)
       node = node.getRight()
  }
  return find
}
```

Insertion

- Start at the root
- successively down the tree from the root choosing the appropriate sub-tree
- Arriving in a leaf, insert in the appropriate side

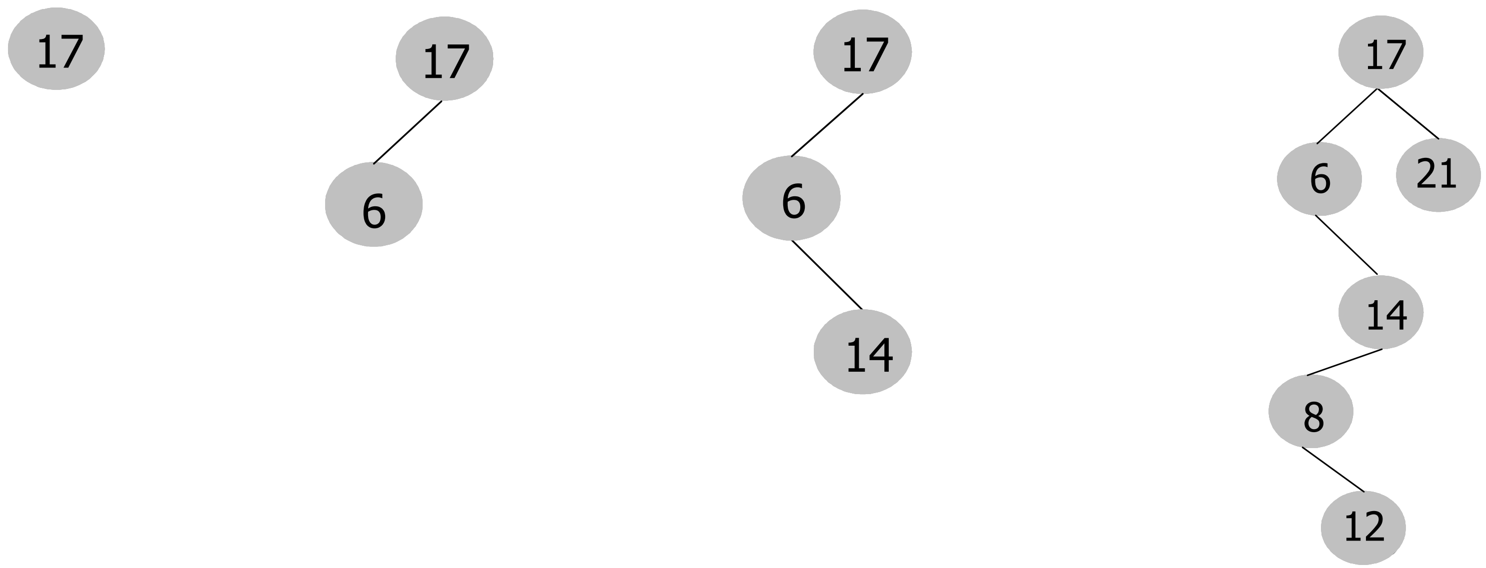
  The shape of the tree depends on the order of elements insertion:
  12, 17, 6, 21, 8, 14



Insertion

The shape of the tree depends on the order of elements insertion:

17, 6, 14, 21, 8, 12



What happens if the elements are inserted into the tree in ascending or descending order?

Insertion

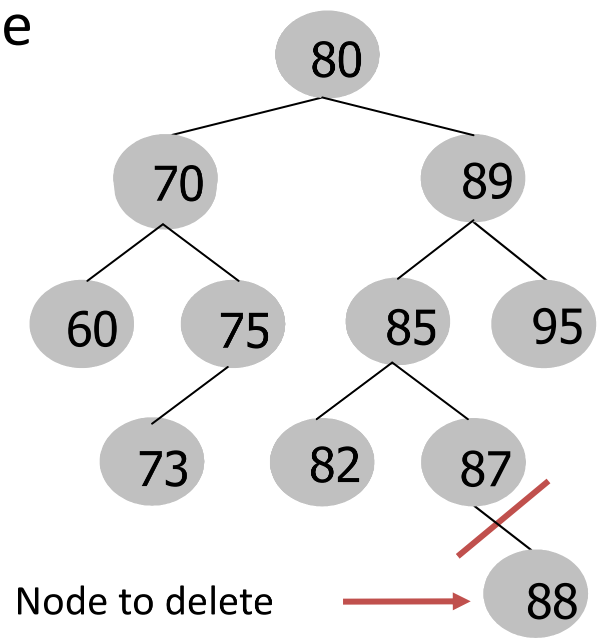
```
Algorithm Node<E> insert(Node<E> node, E elem){

    if (node == null)
       return new Node(elem, null, null)

    if (node.getElement() > elem)
        node.setLeft(insert(node.getLeft(),elem))
    else
        if (node.getElement() < elem)
           node.setRight(insert(node.getRight(),elem))

    return node
}
```

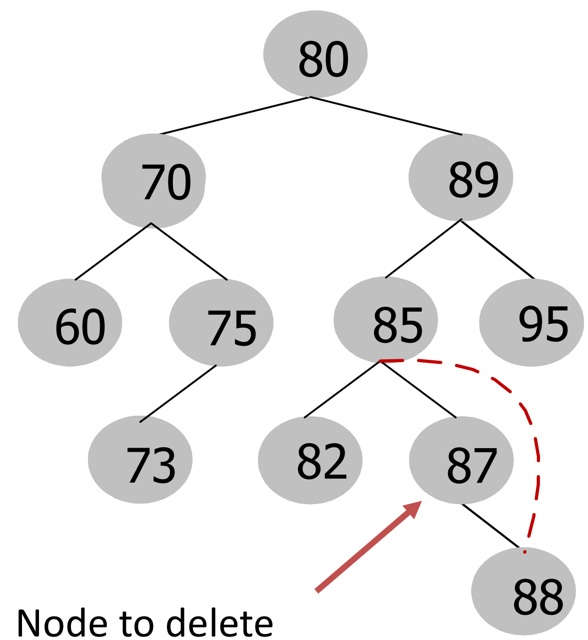
Deletion

When delete a node three cases can happen:

1. the node is a leaf (it hasn't subtrees)
2. the node has only one subtree
3. the node contains two subtrees (left and right)

The first two cases (1 and 2) are solved by adjusting the pointer of the previous node (parent node) that points to the node we want to eliminate





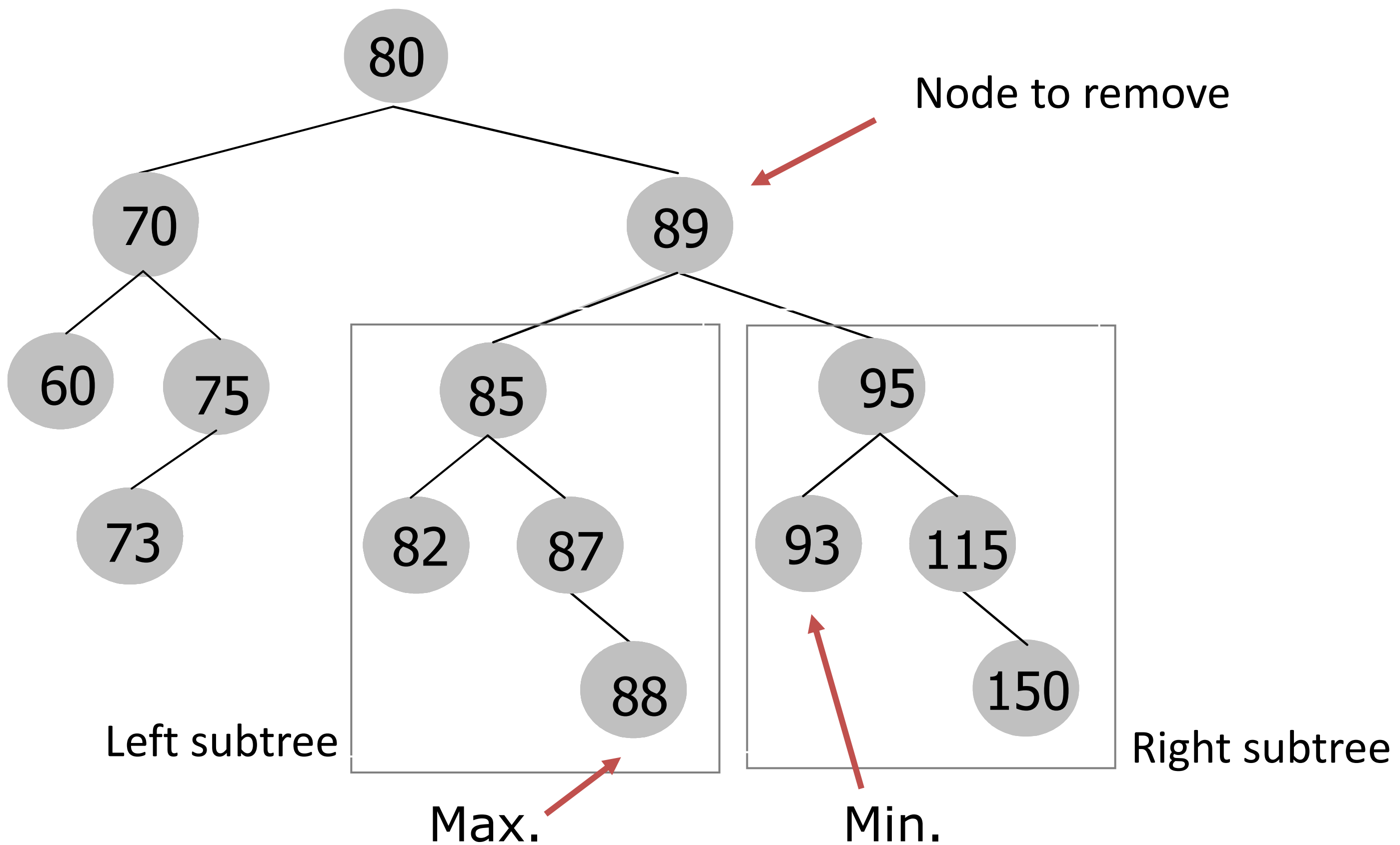
Deletion

Case 3:

- replace the node to eliminate with the greatest node of the left subtree of the node to delete

or

- replace the node to eliminate with the smaller node of the right subtree of the node to eliminate



Deletion

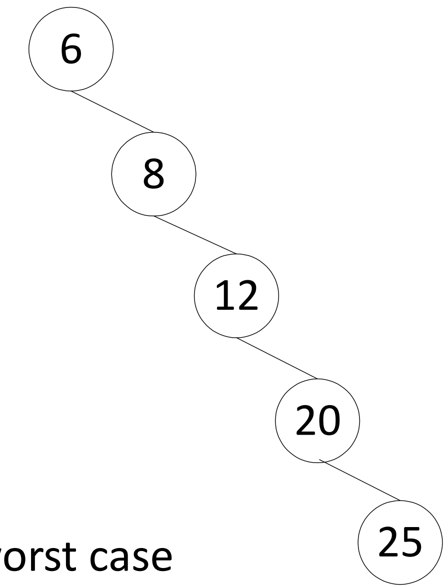
```
Algorithm Node<E> remove(E elem, Node<E> node) {
    if (node == null)
       return null

    if (node.getElement() == elem) {

        if (node.getLeft() == null && node.getRight()== null)
           return null

        if (node.getLeft() == null)
           return node.getRight()

        if (node.getRight() == null)
           return node.getLeft()

        E min = smallestElement(node.getRight())
        node.setElement(min)
        node.setRight(remove(min, node.getRight()))  }
    else if (node.getElement() > elem)
        node.setLeft(remove(elem,node.getLeft()))
    else
        node.setRight(remove(elem,node.getRight()))
    return node }
```

Performance BST methods

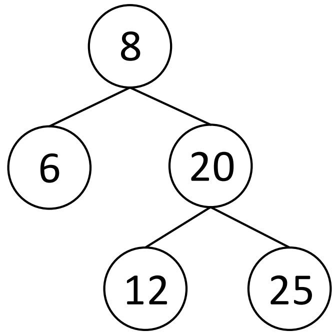
The analysis of search, insert and remove is similar

- In each case, h nodes are visited
- If each node is visited at O(1)
- The methods take O(h) time

The height h is O(n) in the worst case and O(log n) in the best case



worst case



best case

To make sure height h of a tree is always O(log n), the tree must be balanced