Topic 4

Tree, Binary Tree and Binary Search Tree

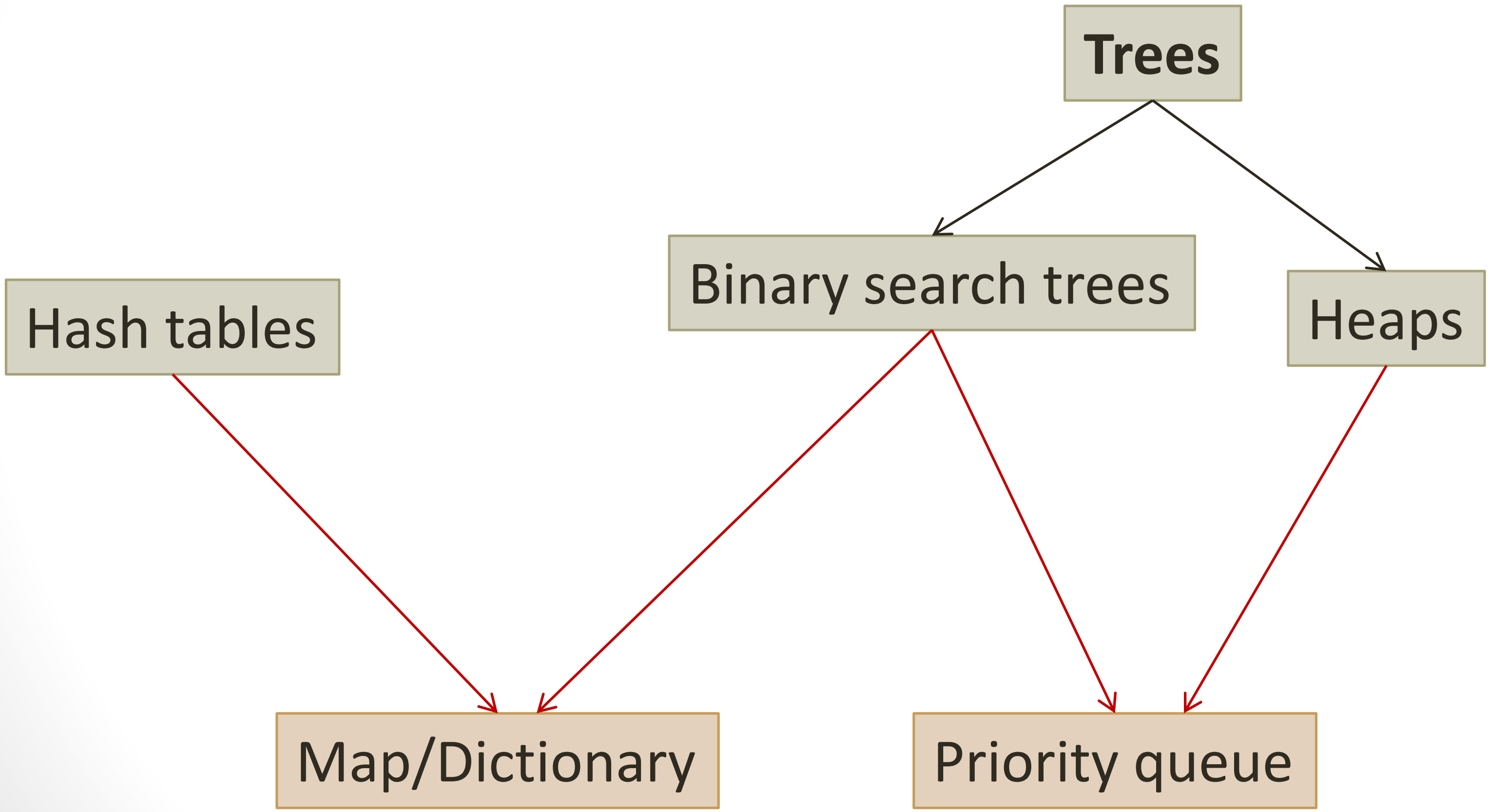
Aim

- To learn the basic concepts of trees, binary trees, binary search trees.
- To learn the concept of traversal in trees and the basic operations with binary search trees.
- To know the balanced trees.

Contents

1. Concepts of trees

Relationship between models and implementations



1. Concepts of trees

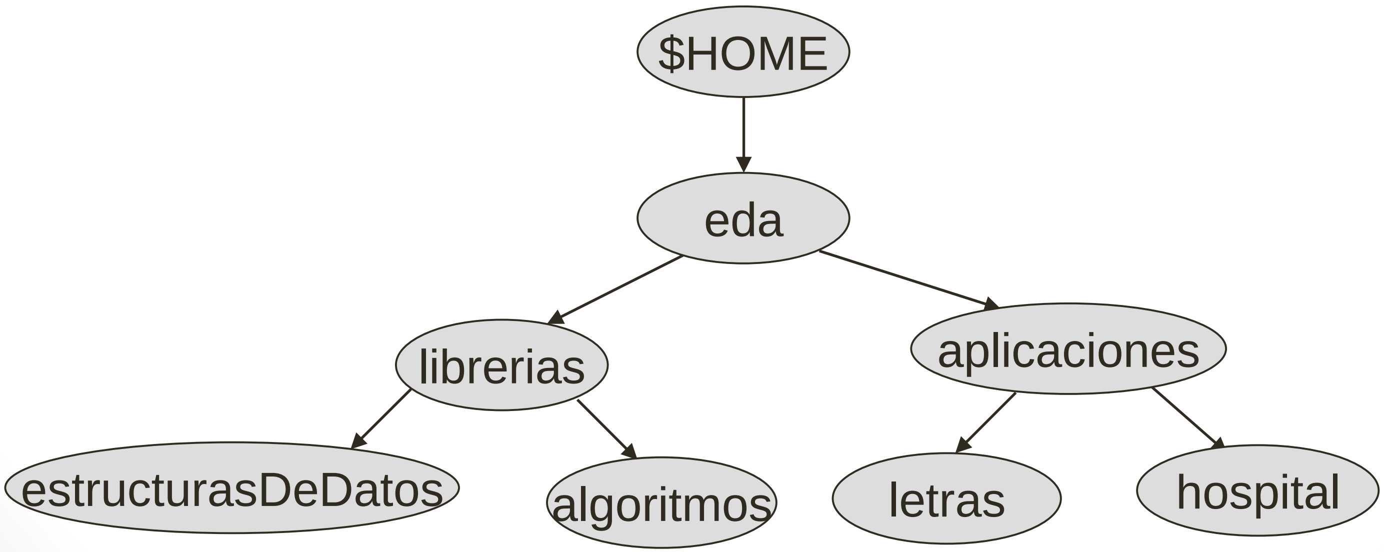
Models: linear vs. hierarchical

- Linear data structures allow to describe sets of data that allow relationships of successor (or of predecessor).
  - Example: list of clients of an enterprise, jobs in the print queue, etc.
- Trees allow to represent hierarchical structures among data sets.
  - Example: structure of directories, genealogic tree, arithmetic expressions, etc.

1. Concepts of trees

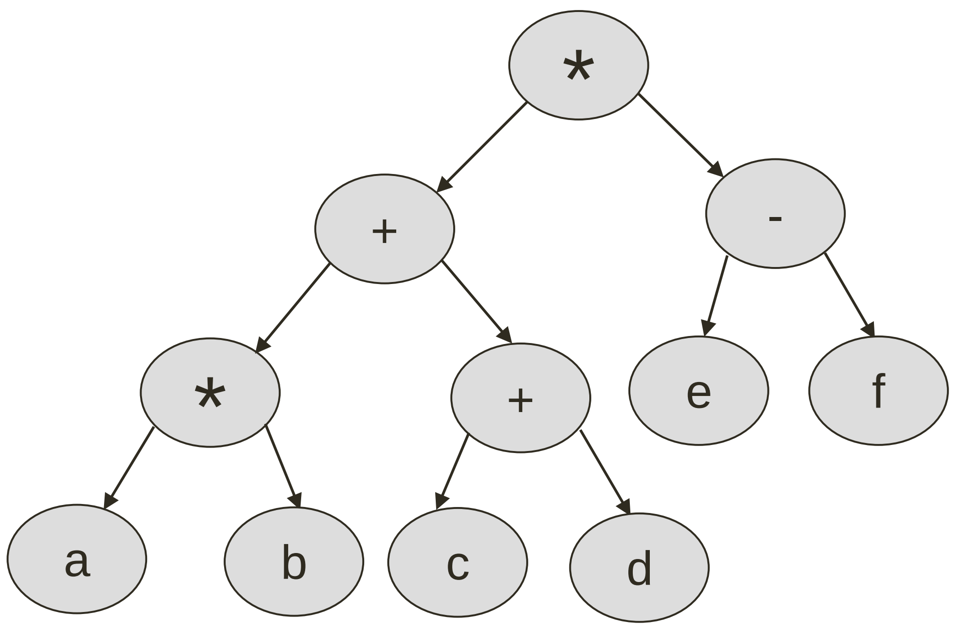
Hierarchical structures

- Sometimes data of a collection have hierarchical relationships that is not possible to model with a linear representation.
  - Example 1: collection of directories for works in the lab of EDA



1. Concepts of trees
Hierarchical structures

- Example 2: the following tree represents the arithmetic expression (((a*b)+(c+d))*(e-f)):



- Trees are basic structures for search and optimisation problems (chess, draughts, sudoku, etc.)

1. Concepts of trees

Basic concepts

- A ***tree*** is a hierarchical structure that is possible to define through a set of **nodes** (one is the **root** of the tree) and a set of **edges** such that:
  - Each node *H*, with the exception of the root, is linked to a unique node *P* via an edge. *P* is the node **father** and *H* is the **child**
  - A node without children is a **leaf**
  - A node that is not a leaf is an **inner node**
  - The **degree** is the number of its children

1. Concepts of trees
Example

<u>Root:</u>

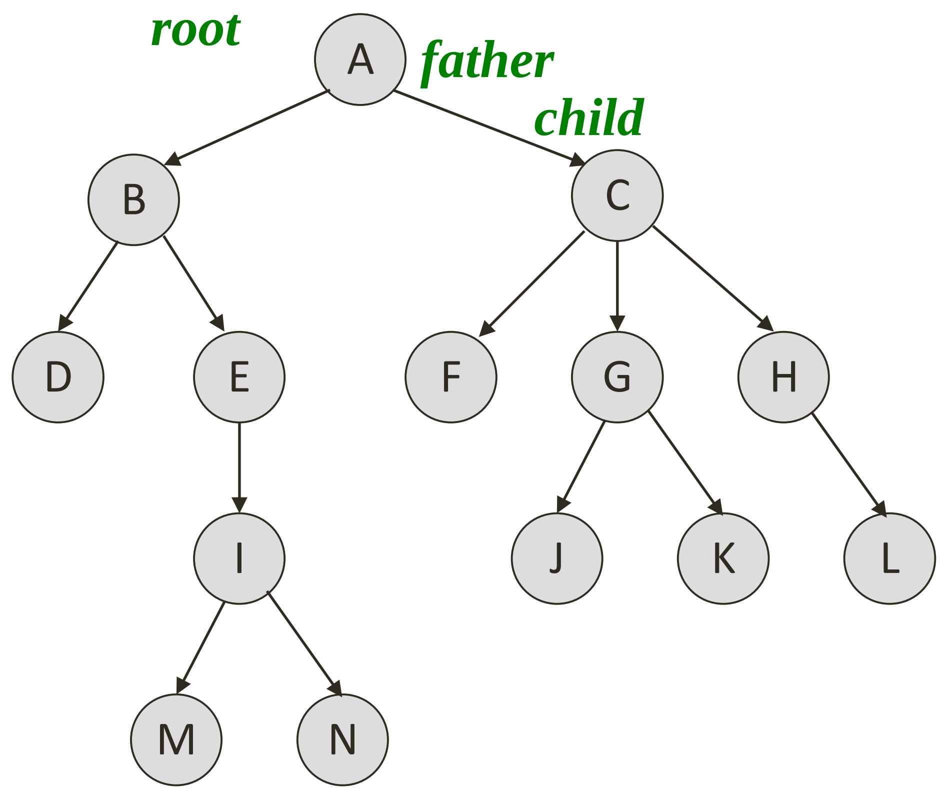
A

<u>Leaves:</u>

{D, M, N, F, J, K, L}

<u>Inner nodes:</u>

{A, B, E, I, C, G, H }



1. Concepts of trees
Length, depth and height

- In a tree there is a unique **path** from the root to each node
- The number of edges of a path gives its **length**
- **Depth** of a node: length of the path from the root to the node
  - The depth of the root is 0
  - All the nodes at the same depth belong to the same **level**
- **Height** of a node: length of the path from the node to its deepest leaf
  - Height of a tree = Height of its root

1. Concepts of trees
Exercise 1

a) How many edges does a tree with *N* nodes have?

b) Length of the path A-D?

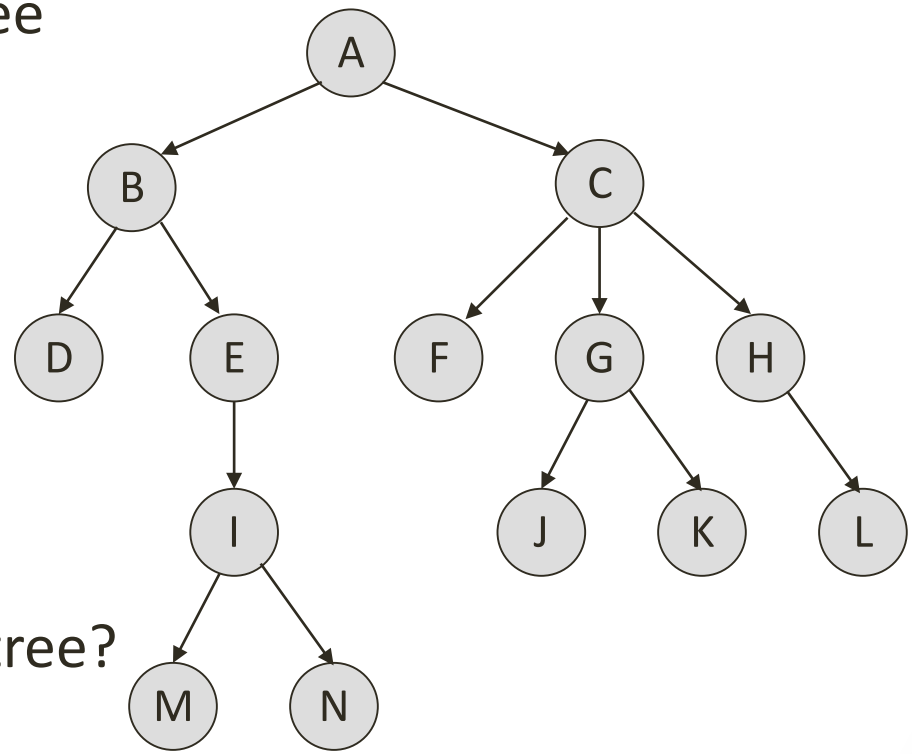
c) Length of C-K?

d) Length of B-N?

e) Length of B-B?

f) Depth of A, B, C and F?

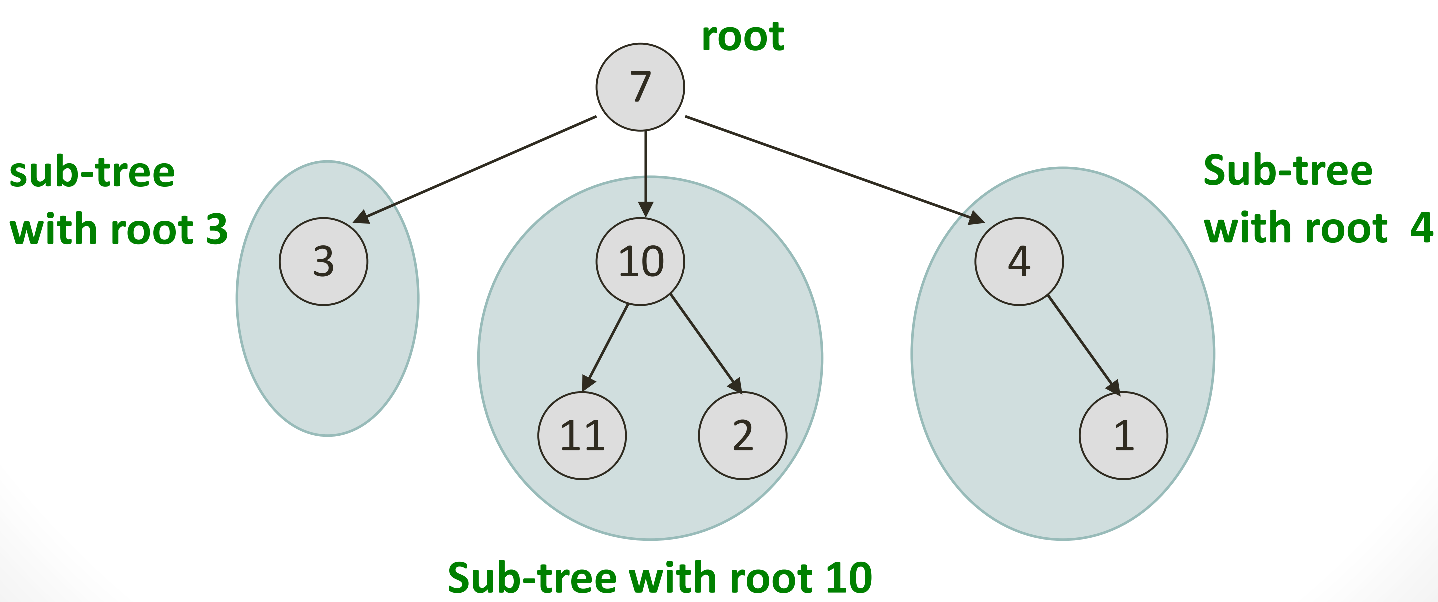
g) Height of B, C, I, F and the tree?



1. Concepts of trees

Recursive definition of tree

- A tree is:
  - An empty set (without nodes and edges) , or
  - A root and zero or more not empty sub-trees where each of its roots is linked via an edge to the root



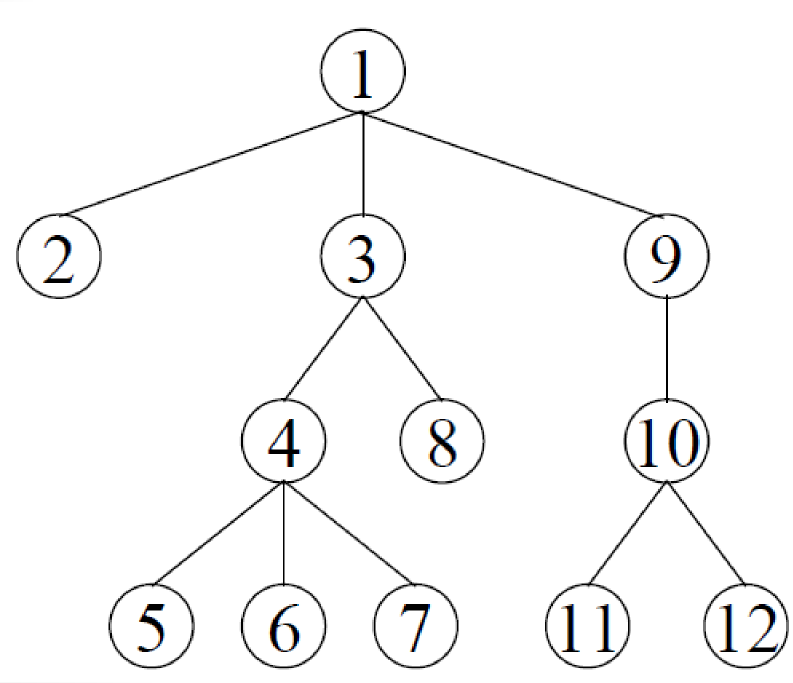
2. Generic trees
Representation

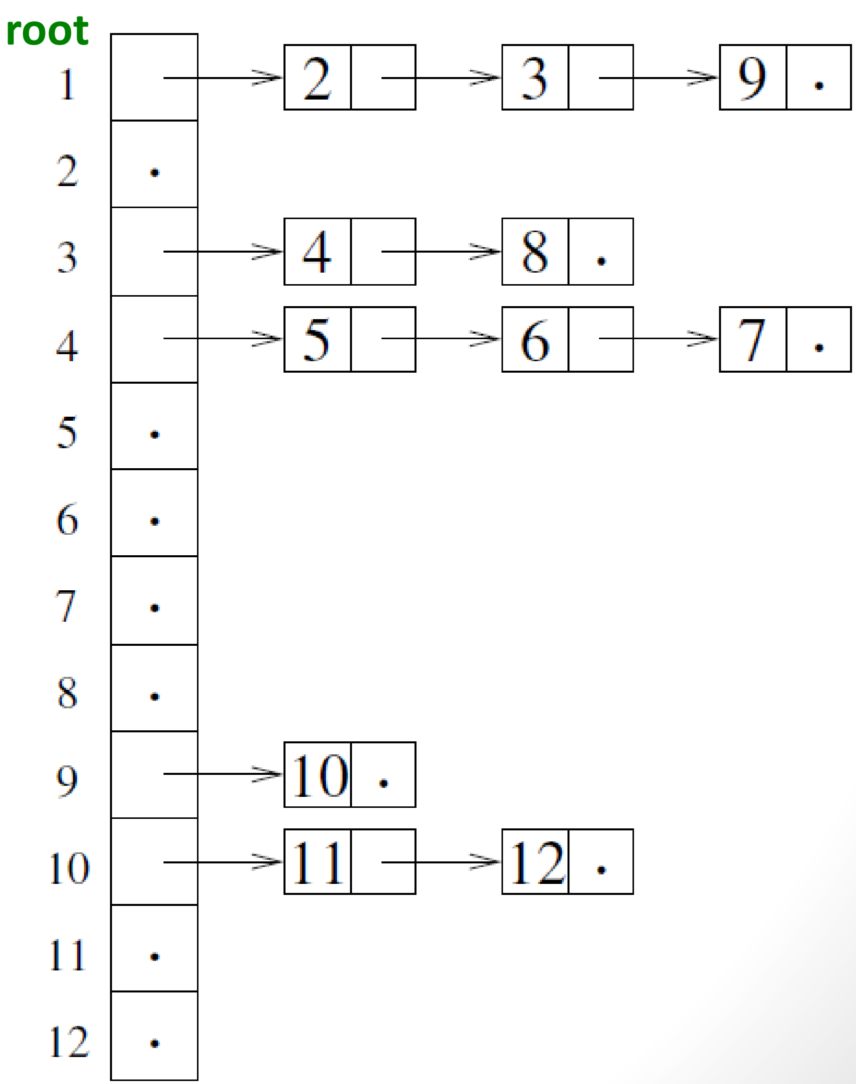
- Representation of generic trees (with no upper bound for number of children) :
  - Lists (sorted) of children
  - Leftmost child – right brother
  - With arrays and references to the father (*mf-sets*)
  - Others…

2. Generic trees

Representation

- <u>Example:</u>
  sorted lists of children

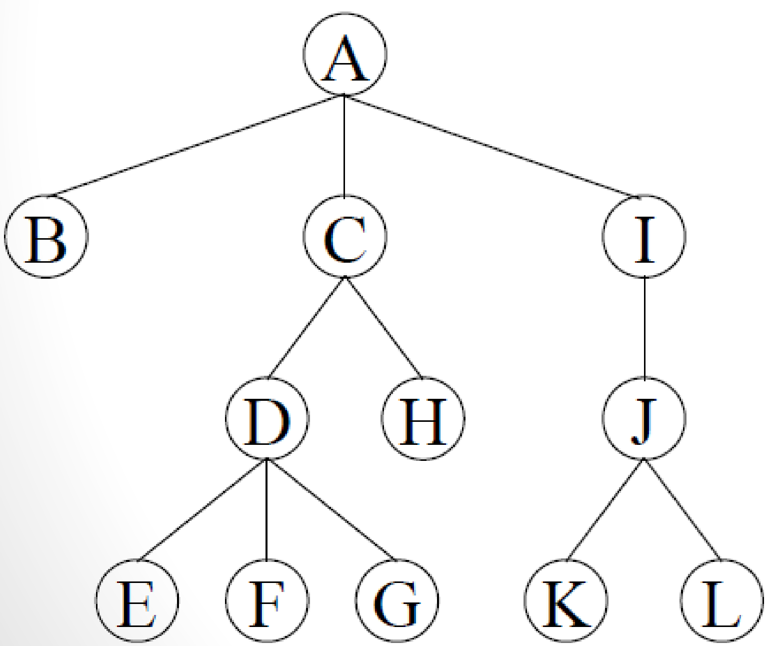




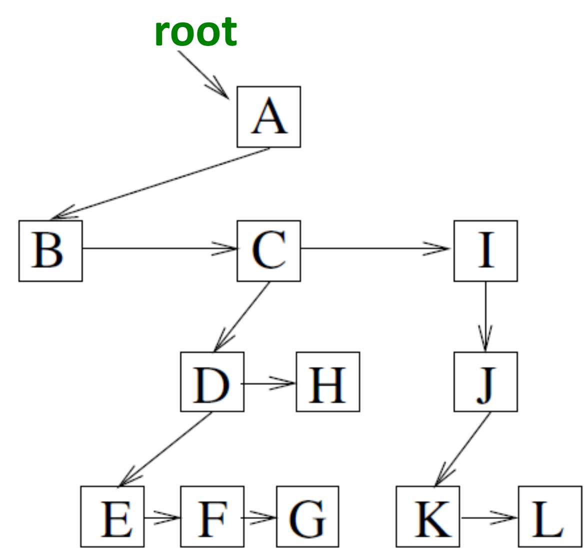
2. Generic trees

Representation

- <u>Example:</u> leftmost child – right brother

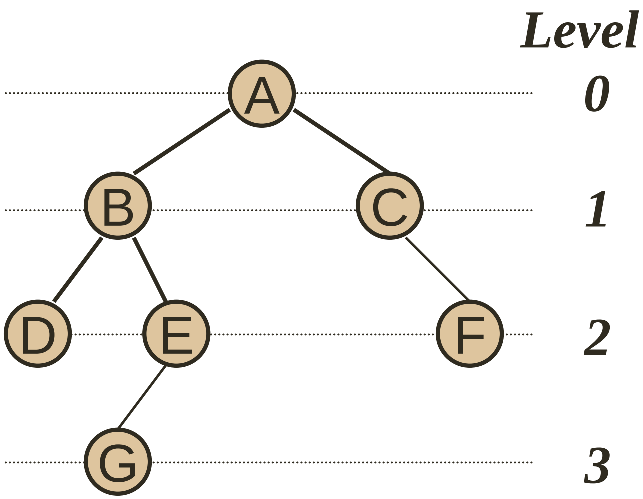


	left child	data	right brother
2	3	C	10
3	17	D	9
root 4	8	A	.
....			
8	.	B	2
9	.	H	.
10	13	I	.
....			
12	.	G	.
13	14	J	.
14	.	K	16
15	.	F	12
16	.	L	7
17	.	E	15
....			



3. Binary trees
Definition and properties

- A **binary tree** is a tree where each node has as maximum two children (left child and right child)
- Properties:
  - The maximum number of nodes at level **i** is $\mathbf{2^i}$
  - In a tree of height **H**, the maximum number of nodes is: $\sum_{i=0..H} 2^i = 2^{H+1} - 1$
  - The maximum number of leaves is: $(2^{H+1} - 1) - (\sum_{\mathbf{i=0..H-1}} 2^{\mathbf{i}}) = 2^H$
  - The maximum number of inner nodes: $(2^{H+1}-1) - (2^H) = 2^H - 1$

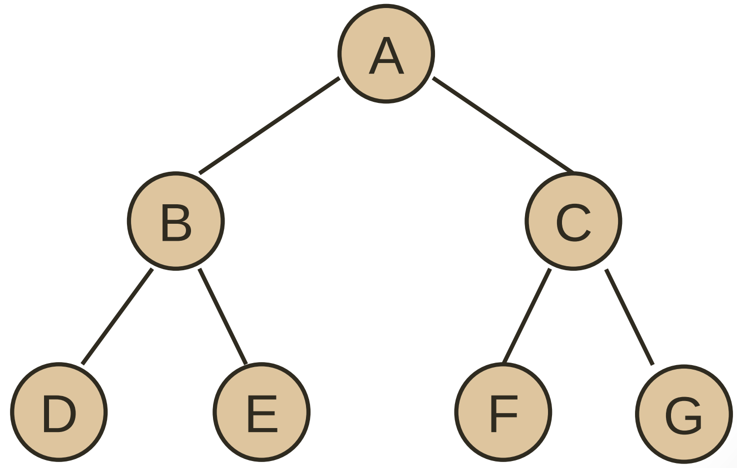


3. Binary trees

Definition and properties

- A **binary tree** is **full** if all its levels are complete
- Properties: be H its height and N its size (number of nodes)
  - $H = \lfloor \log_2 N \rfloor$
  - $N = 2^{H+1} - 1$

N = 7
H = 2

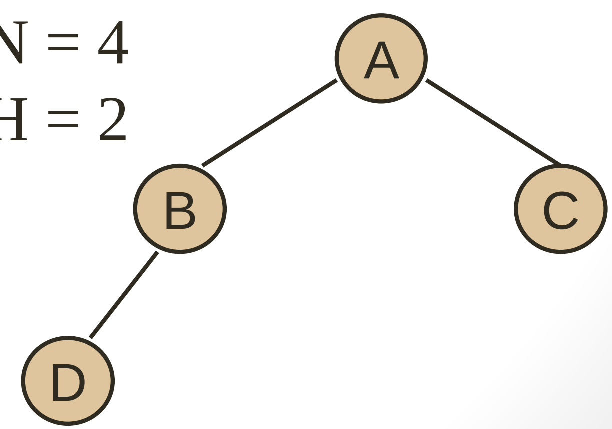


3. Binary trees

Definition and properties

- A ***complete* binary tree** has all its level complete, except maybe the last one in each all the leaves are leftmost as possible
- Properties: be H its height and N its size (number of nodes)
  - $H \leq \lfloor \log_2 N \rfloor \rightarrow$ is a ***balanced*** tree
  - $2^H \leq N \leq 2^{H+1} - 1$

N = 4
H = 2

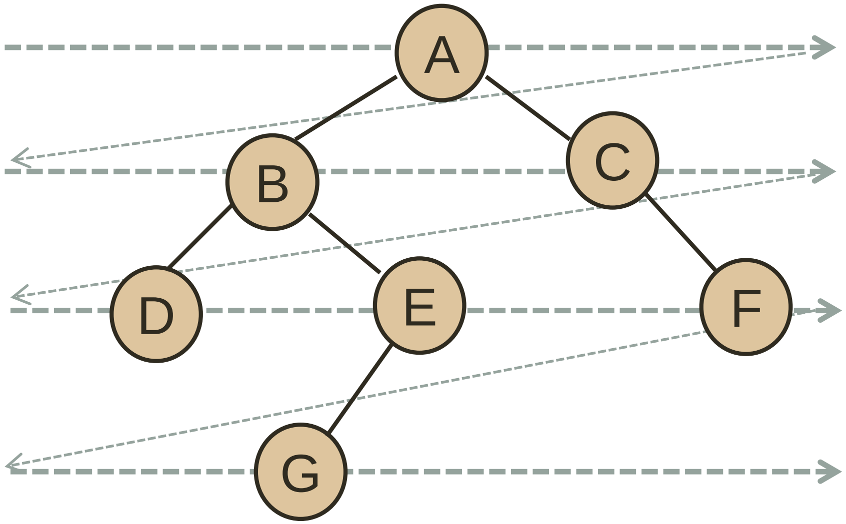


Note: a tree is *balanced* if the difference between the heights of the left and right sub-trees of each of its nodes is as maximum equal to 1

4. Traversal operations

Traversal by levels

- In a traversal **by levels** of a binary tree the nodes are visited level by level and, in each level, from left to right



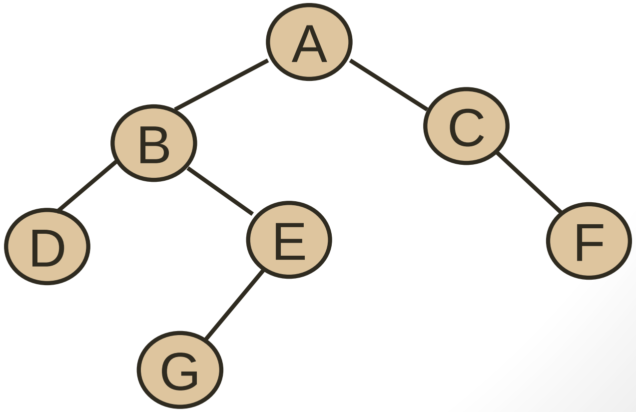
By levels: ABCDEFG

4. Traversal operations

Traversal in depth

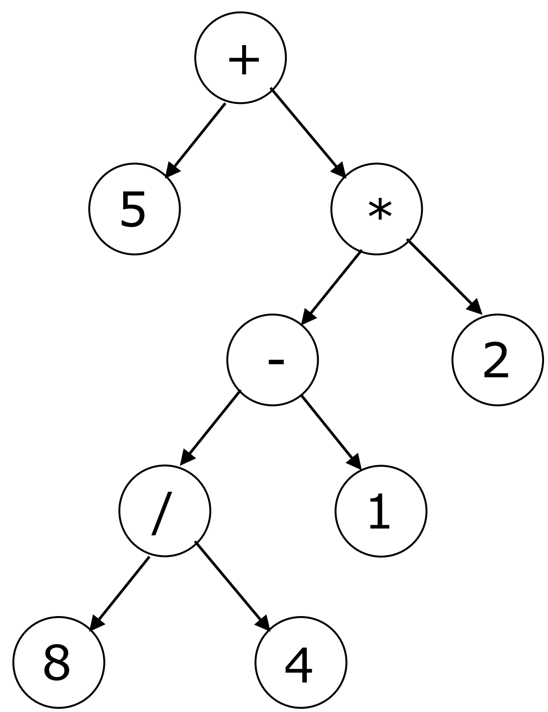
- **In depth**. The nodes can be visited in the following orders:
  - ***Pre-Order***:
    1º) root, 2º) left sub-tree, 3º) right sub-tree
  - ***In-Order***:
    1º) left sub-tree, 2º) root, 3º) right sub-tree
  - ***Post-Order***:
    1º) left sub-tree, 2º) right sub-tree, 3º) root

Pre-Order: ABDEGCF
In-Order: DBGEACF
Post-Order: DGEBFCA



4. Traversal operations
Exercise 2

- Given the following tree, show the results of the pre-order, in-order, post-order and by levels trasversal operations:



5. Binary search trees
Basic concepts

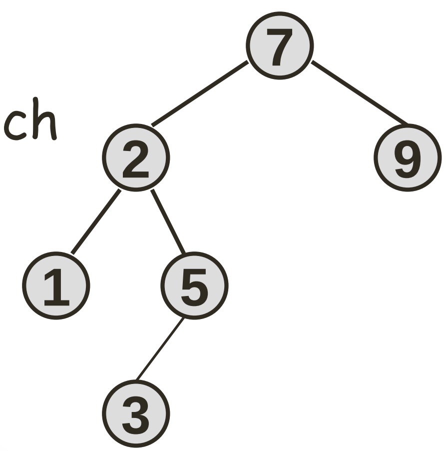
- Data structure that can be used for the implementation of dictionaries and priority queues
- It is a generalisation of the binary search
- It allows for implementing efficiently operations such as search, search min, max, predecessor and successor
- It allows also for an efficient implementation of the insert and delete operations

5. Binary search trees

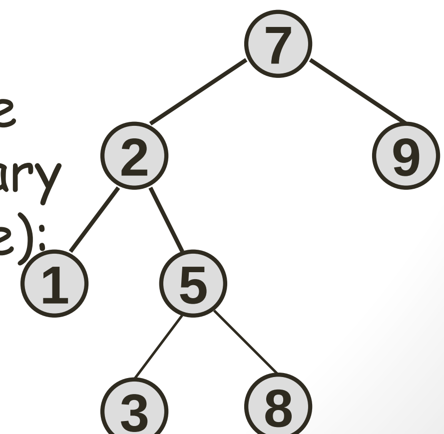
Definition

- A binary tree is a ***binary search tree*** (BST) if:
  - Data of its left sub-tree are smaller than the root
  - Data of its right sub-tree are greater than the root
  - The left and right sub-trees are also binary search trees
- If a binary search tree is visited in-order the result is a sorted sequence of its elements

Binary Search Tree:

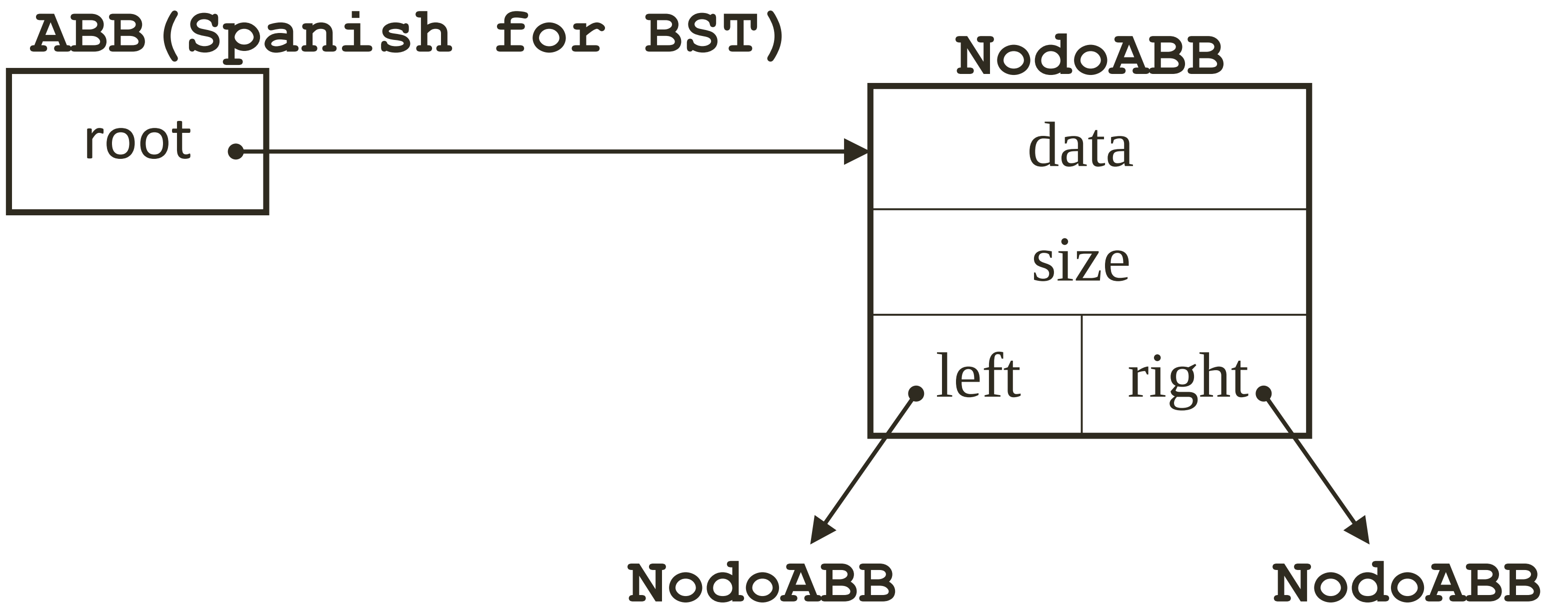


Binary tree (but not Binary Search Tree):



5. Binary search trees

Linked representation



5. Binary search trees

The class NodoABB

```
package librerias.estructurasDeDatos.jerarquicos;
class NodoABB<E> {
   E data;                             // data
   NodoABB<E> left, right;             // children
   int size;                     // size of node (optional)
   // Constructors
   NodoABB(E data, NodoABB<E> l, NodoABB<E> r) {
      this.data = data; size = 1;
      this.left = l; this.right = r;
      if (left != null) size += left.size;
      if (right != null) size += right.size;
   }
   NodoABB(E data) {
      this.data = data; size = 1;
      this.left = this.right = null;
   }
}
```

5. Binary search trees
The class ABB

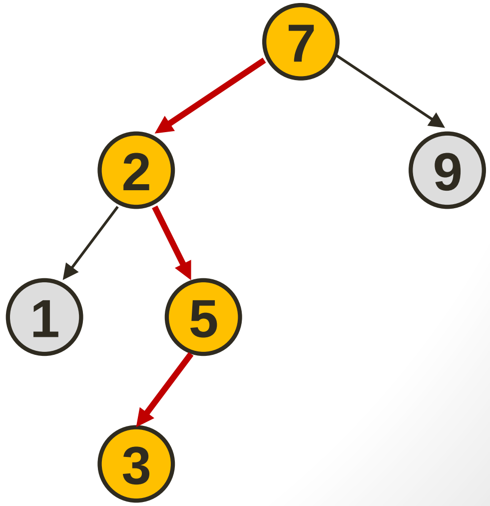
```
package librerias.estructurasDeDatos.jerarquicos;

public class ABB<E extends Comparable<E>> {
  // Attributes
  protected NodoABB<E> root; // Root of ABB


  /** Constructor of an empty ABB **/
  public ABB() {
    root = null;
  }

  …
```

5. Binary search trees

Search in ABB

```java
// Search for x in ABB and returns it.
// Otherwise return null
public E search(E x) {
  NodoABB<E> node = root;
  while (node != null) {
    int resC = x.compareTo(node.data);
    if (resC == 0) return node.data;
    node = resC < 0 ? node.left : node.right;
  }
  return null;
}
```

Example:
search for 3



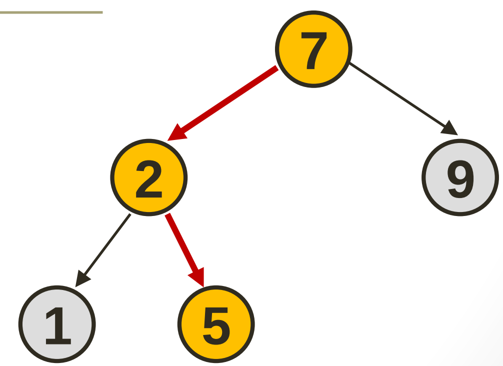
5. Binary search trees

Search in ABB(recursive version)

```
public E search(E x) {
  return search(x, root);
}
protected E search(E x, NodoABB<E> node) { // recursive
  if (node == null) return null;          // not found
  int cmp = x.compareTo(node.data);
  if (cmp < 0) return search(x, node.left);
  else if (cmp > 0) return search(x, node.right);
  else return node.data;               // x found
}
```

<u>Initial invocation</u> (searching for x=5):

```
return search(5, 7);
        cmp < 0
   5    return search(5,2)
                 cmp > 0
          5      return search(5,5)
                        cmp = 0
                 5      return 5
```



Example: searching for 5

5. Binary search trees

Exercise

- Exercise 3: if the number 363 is searched in a BST that contains numbers from 1 to 1000, which among the following sequences of visited numbers cannot be possible?

  a) 2, 252, 401, 398, 330, 344, 397, 363
  b) 924, 220, 911, 244, 898, 258, 362, 363
  c) 925, 202, 911, 240, 912, 245, 363
  d) 2, 399, 387, 219, 266, 382, 381, 278, 363
  e) 935, 278, 347, 621, 299, 392, 358, 363

5. Binary search trees

Size of a ABB

```
// Return the number of elements in ABB
 public int size() {
   return size(root);
}

protected int size(NodoABB<E> node) {
  if (node == null) return 0;
  else return node.size;
}

public boolean isEmpty() {
  return root == null;
}
```

Having the attribute *size* in the nodes, it is not necessary having *size* in *ABB*.

5. Binary search trees

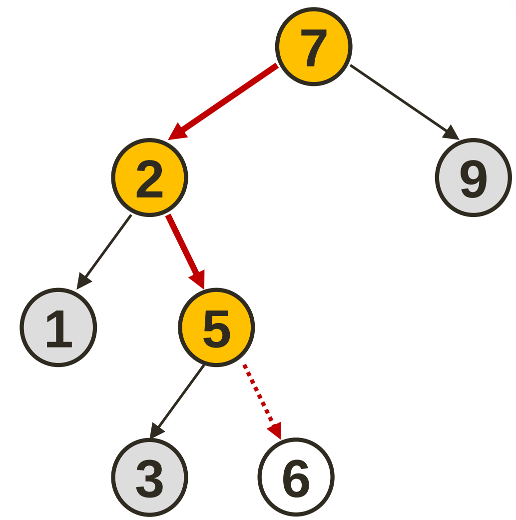
Insert in ABB (recursive version)

```
// Update x in ABB;if x is not in ABB, insert it
public void insert(E x) {
  root = insert(x, root);
}

protected NodoABB<E> insert(E x, NodoABB<E> node) {
  if (node == null) return new NodoABB<E>(x);
  int cmp = x.compareTo(node.data);
  if (cmp < 0) node.left = insert(x, node.left);
  else if (cmp > 0) node.right = insert(x, node.right);
  else node.data = x;
  node.size = 1 + size(node.left) + size(node.right);
  return node;
}
```

5. Binary search trees

Insert in ABB

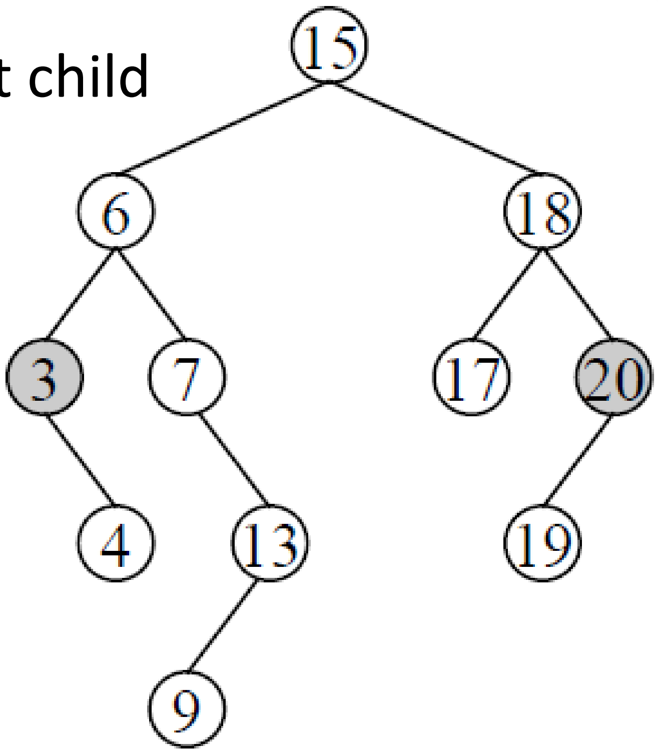
- Example: insert 6




```
root = insert(6,⑦);
  cmp < 0
  ⑦.left = insert(6,②)
      cmp > 0
      ②.right = insert(6,⑤)
          cmp > 0
          ⑤. right = insert(6,null)
              return ⑥
          ⑤.size=3
          return ⑤
      ②.size=5
      return ②
  ⑦.size=7
  return ⑦
```

5. Binary search trees

Smallest and greatest in ABB



- The smallest in ABB does not have left child and it does not belong to any right subtree of any node.
- *The greatest is the symmetric case.*

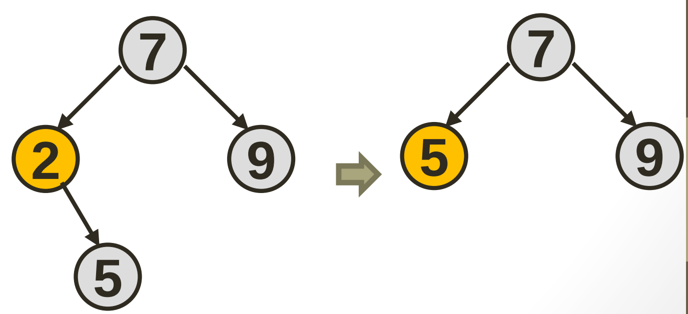
```
// Return the smallest
public E retrieveMin() {
  if (root == null) return null;
  return retriveMin(root).data;
}

protected NodoABB<E> retrieveMin(NodoABB<E> node) {
  if (node.left == null) return node;
  else return retrieveMin(node.left);
}
```

5. Binary search trees

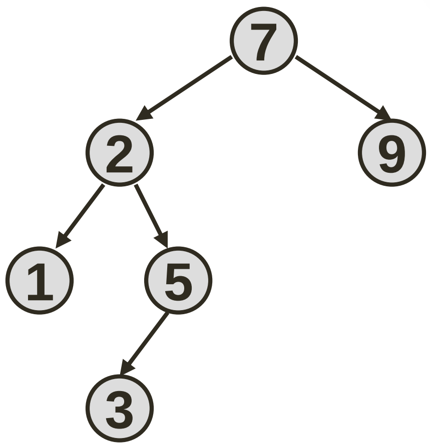
Delete smallest in ABB

```
public E deleteMin() {
   E min = retrieveMin();
   if (min != null) root = deleteMin(root);
   return min;
}

protected NodoABB<E> deleteMin(NodoABB<E> node) {
   if (node.left == null) return node.right;
   node.left = deleteMin(node.left);
   node.size--;
   return node;
}
```

```
root = deleteMin(7);
       7.left = deleteMin(2)
       7.size=3             return 5
       return 7
```



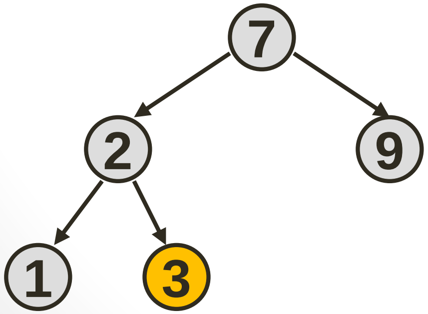
5. Binary search trees
Delete in ABB



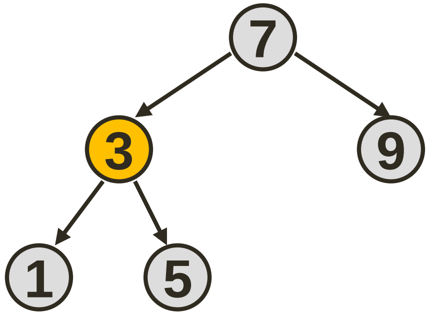
Possible cases:

a) The node to be deleted does not have children
Example: 3

b) The node to be deleted has a child
Example: 5
Its child takes its position:



c) The node to be eliminated has two children
Example: 2
The smallest of its right subtree takes its position:



5. Binary search trees

Delete in ABB

```java
// Delete the node with x
public void delete(E x) {
  root = delete(x, root);
}

protected NodoABB<E> delete(E x, NodoABB<E> node) {
  if (node == null) return node;     // x not found
  int cmp = x.compareTo(node.data);
  if (cmp < 0) node.left  = delete(x, node.left);
  else if (cmp > 0) node.right = delete(x, node.right);
  else {             // x found -> we delete the node
   if (node.right == null) return node.left; // 1 child
   if (node.left == null) return node.right; // 1 child
   node.data = retrieveMin(node.right).data; // 2 children
   node.right = deleteMin(node.right);
  }
  node.size = 1 + size(node.left) + size(node.right);
  return node;
}
```

5. Binary search trees

Traversal in depth

- The natural implementation of the traversal in-depth methods is recursive:

```
public String preOrder() {
  return preOrder(root);
}
private String preOrder(NodoABB<E> actual) {
  if (actual == null) return "";
  return actual.data.toString() + "\n" +
   preOrder(actual.left) + preOrden(actual.right);
}
```

- The methods Post-Order and In-Order are very similar (the order of the instructions changes)

5. Binary search trees

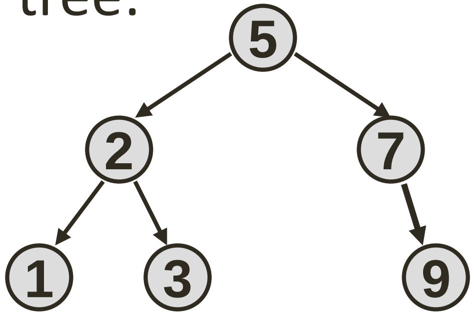
Traversal by levels

- Its design is iterative and it employs a *Queue* as auxiliar structure

```
public String byLevels() {
  if (root == null) return "";
  Cola<NodoABB<E>> q = new ArrayCola<NodoABB<E>>();
  q.enqueue(root);
  String res = "";
  while (!q.isEmpty()) {
   NodoABB<E> actual = q.dequeue();
   res += actual.data.toString() + "\n";
   if (actual.left != null) q.enqueue(actual.left);
   if (actual.right != null) q.enqueue(actual.right);
  }
  return res;
}
```

5. Binary search trees
Successor/predecessor

- If a node has the right sub-tree, the successor of the node is the min of its right sub-tree (the smallest among the elements that are greater)
- Otherwise, the successor is the closest ancestor
- The successor of a node is the next visited node in a in-order trasversal of the tree:



successor(5) = 7
successor(1) = 2
successor(3) = 5
successor(9) = *null*

- Exercise 5: Predecessor (max of the left sub-tree, if any, otherwise the closest ancestor, on the left ?)

5. Binary search trees

Successor/predecessor

```
/* Return the successor of e in ABB, null otherwise */

public E successor(E e) {
  E successor = null;
  NodoABB<E> aux = this.root;
  while (aux != null) {
    int resC = aux.data.compareTo(e);
    if (resC > 0) {
      successor = aux.data;
      aux = aux.left;
    } else aux = aux.right;
  }
  return successor;
}
```

5. Binary search trees

Complexity of operations

Average complexity	search(x)	insert(x)	min ()	deleteMin()
Linked list / Array	Θ(**N**)	Θ(**1**)	Θ(**N**)	Θ(**N**)
LEG (sorted)	Θ(**N**)	Θ(**N**)	Θ(**1**)	Θ(**1**)
Array (sorted)	Θ(**log N**)	Θ(**N**)	Θ(**1**)	Θ(**1**)
ABB	Θ(**log N**)	Θ(**log N**)	Θ(**log N**)	Θ(**log N**)

- The complexity of the operations in ABB depends on the height of the tree (*h*)
- The height *h* is between $\Omega(\log_2 n)$ and O(n)
- In the worst case (ABB imbalanced), the complexity is linear

5. Binary search trees
Exercises

- <u>Exercise 6</u>: Design a method that returns the data of the father of a given element. Study the time complexity of the method.
- <u>Exercise 7</u>: Design a method that returns the level of a node that contains the data x (hp: there are no duplicated data)
- <u>Exercise 8</u>: Design a new constructor for the class ABB that, given an empty ABB, inserts the data of a vector in a way to obtain a balanced ABB.
- <u>Exercise 9</u>: Design a method in ABB to delete all the elements smaller that a given element.

5. Binary search trees

Exercises

- Exercise 10: Design the following methods of ABB that return*:*
  - The number of leaves
  - The data of the nodes of level k
  - The height
- Exercise 11: Design the class ABBInteger as an ABB that works with Integer data, and add the following two methods:
  - To obtain the sum of all elements that are greater (or smaller) than a given int value
  - To change the sign of all the data of the tree. The ABB has to accomplish with the order property of a binary search tree.

6. The class ABBMap

Implementation

- The model *Map* allows searching for **key** obtaining the associated **value** to the given entry
- An entry is a pair (*key*, *value*)
- Two entries with the same key (that is, duplicated elements) are not allowed

⇒In order to implement the interface *Map* with an ABB we need to define the class *EntradaMap*

```
public interface Map<C, V> {
  V insert(C c, V v);
  V delete(C c);
  V retrieve(C c);
  boolean isEmpty();
  int size();
  ListaConPI<C> keys();
}
```

6. The class ABBMap

The class EntradaMap

```
class EntradaMap<C extends Comparable<C>,E>
      implements Comparable<EntradaMap<C,E>> {
  C key;
  E value;
  public EntradaMap(C c, E e) {key = c; value = e;}
  public EntradaMap(C c) { this(c, null); }
  public boolean equals(Object x) {
    return ((EntradaMap<C,E>)x).key.equals(this.key);
  }
  public int compareTo(EntradaMap<C,E> x) {
    return this.key.compareTo(x.key);
  }
  public String toString() {
    return this.key + " => " + this.value;
  }
}
```

6. The class ABBMap

Implementation

```
public class ABBMap<C extends Comparable<C>,V>
       implements Map<C,V> {
  private ABB<EntradaMap<C,V>> abb;
  public ABBMap() { abb = new ABB<EntradaMap<C,V>>(); }
  public V retrieve(C c) {
   EntradaMap<C,V> e;
   e = abb.retrieve(new EntradaMap<C,V>(c));
   return e == null ? null : e.value;
  }
  public V insert(C c, V v) {
   EntradaMap<C,V> newentry = new EntradaMap<C,V>(c,v);
   EntradaMap<C,V> prev = abb.insert(newentry);
   return prev == null ? null : prev.value;
  }
  public V delete(C c) {
   EntradaMap<C,V> prev = abb.delete(new EntradaMap<C,V>(c));
   return prev == null ? null : prev.value;
  }
```

7. The class ABBColaPrioridad

Implementation

- It is possible to inherit the methods *deleteMin* and *retrieveMin* from the class *ABB*
- It is necessary overwriting *insert* to allow the insertion of duplicated elements

```
public interface
  ColaPrioridad<E extends Comparable<E>> {
  void  insert(E e);
  E  deleteMin();
  E  retrieveMin();
  boolean isEmpty();
}
```

7. The class ABBColaPrioridad

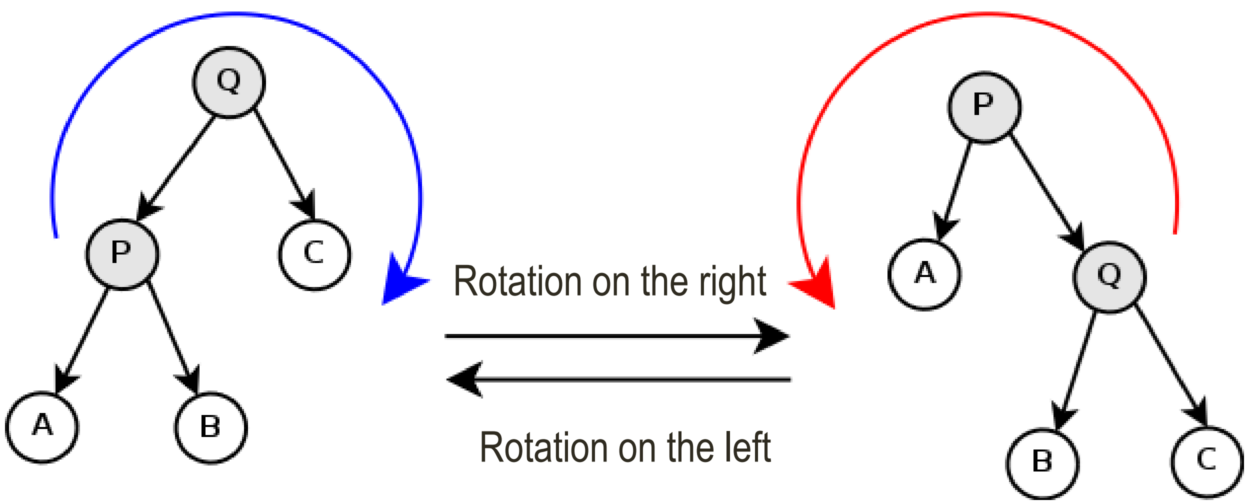
Implementation

```
public class ABBColaPrioridad<E extends Comparable<E>>
       extends ABB<E>
       implements ColaPrioridad<E> {
  public boolean esEmpty() { return super.isEmpty(); }
  public void insert (E x) { // insert with duplicates
    root = insert(x, root);
  }
  protected NodoABB<E> insert(E x, NodoABB<E> node) {
    if (node == null) return new NodoABB<E>(x);
    int cmp = x.compareTo(node.data);
    if (cmp <= 0) node.left = insert(x, node.left);
    else node.right = insert(x, node.right);
    node.size++;
    return node;
  }
}
```

8. Balanced trees

Introduction

- Balanced trees are data structures based on trees that moreover have information and/or methods to balance the trees
- Their behaviour is based on rotations, swapping nodes and sub-trees in a binary search to obtain a (more balanced) equivalent one



8. Balanced trees

The most well-known balanced trees

- AVL trees
  - Store the *balance factor* in each node
  - Are always balanced
- Red-back trees
  - Every node has an attribute that indicates its colour (red or black)
  - Like for AVLs, they are always balanced
- Splay trees
  - The move one level up (through rotations) the element inserted/deleted in the root (and with that they preserve the balance)
- Day–Stout–Warren (DSW)
  - They succeed in balancing the *ABB* in n O(n) (no matter how it was previously)

References

- Data structures, algorithms, and applications in Java, *Sahni* (chapters 12 and 15)

- Data structures in Java, *Weiss* (chapters 17 and 18)

- Data Structures and Algorithms in Java (4th edition), *Goodrich y Tamassia* (chapter 10)