Systems Programming

Trees

Julio Villena Román (LECTURER)
<jvillena@it.uc3m.es>

CONTENTS ARE MOSTLY BASED ON THE WORK BY:
Carlos Delgado Kloos, M.Carmen Fernández Panadero and Raquel M.Crespo García

Universidad Carlos III de Madrid

Contents

- ***Concept***
  - Non recursive definition
  - Recursive definition
  - *Examples*
- ***Terminology***
  - Key concepts
  - Properties
- ***Implementation***
  - Sequence-based
  - Linked structure
    - Basic operations
    - Traversals
- ***Particular cases***
  - Binary search trees
  - Binary heaps

Quote

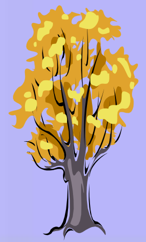
*"The structure of concepts is formally called a **hierarchy** and since ancient times has been a basic structure for all Western knowledge. Kingdoms, empires, churches, armies have all been structured into hierarchies. Tables of contents of reference material are so structured, mechanical assemblies, computer software, all scientific and technical knowledge is so structured..."*

Robert M. Pirsig:

Zen and the Art of Motorcycle Maintenance

Tress

Concept and characteristics

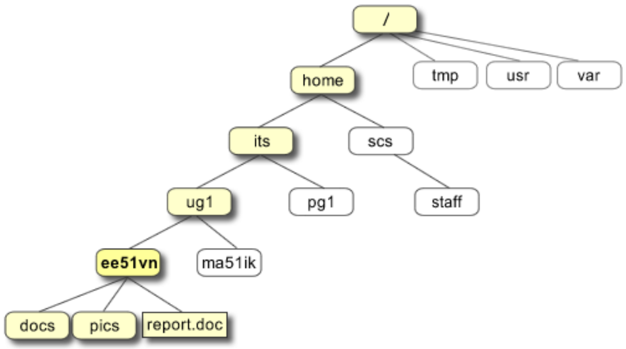


A **tree** is a non-linear data structure that stores the elements **hyerarchically**

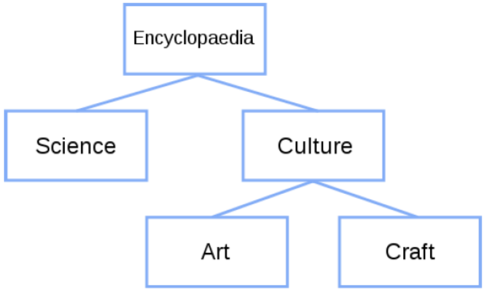
(Generalization of lists)

Examples

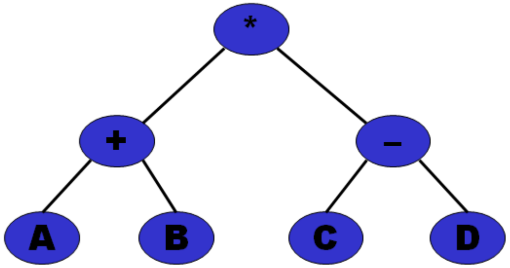
- File system



- Structure of a book or a document



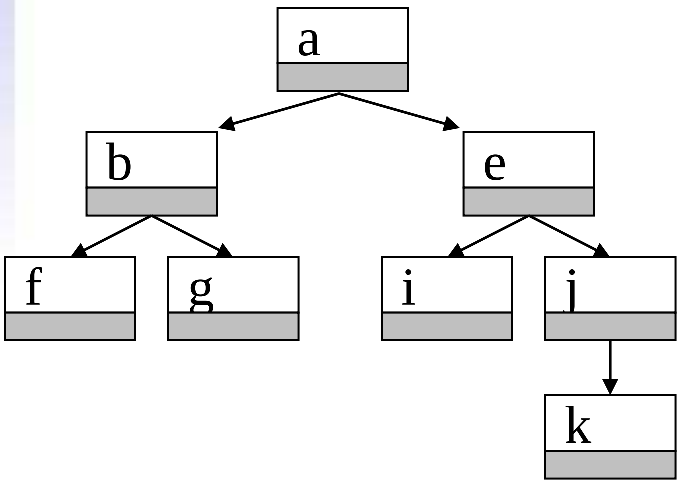
- Arithmetic expressions



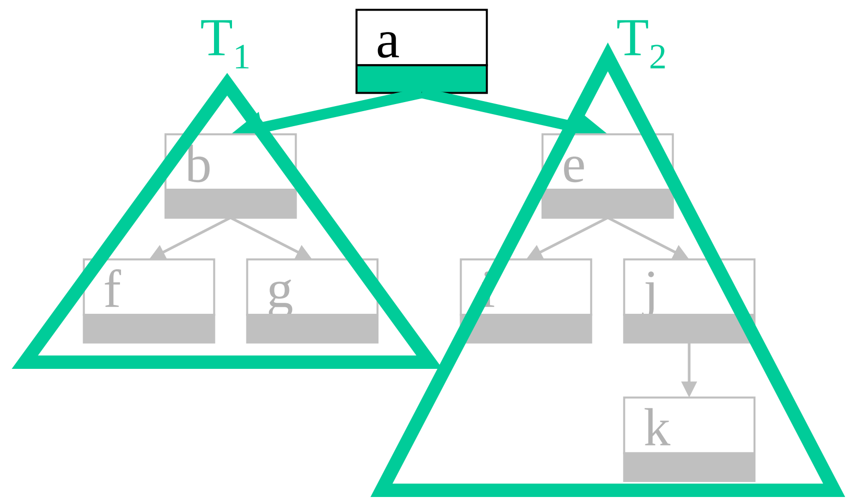
Tress

Concept and characteristics

- Trees can be defined in two ways:
  - Non-recursive definition
  - Recursive definition

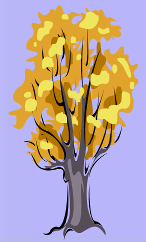


Non-recursive definition



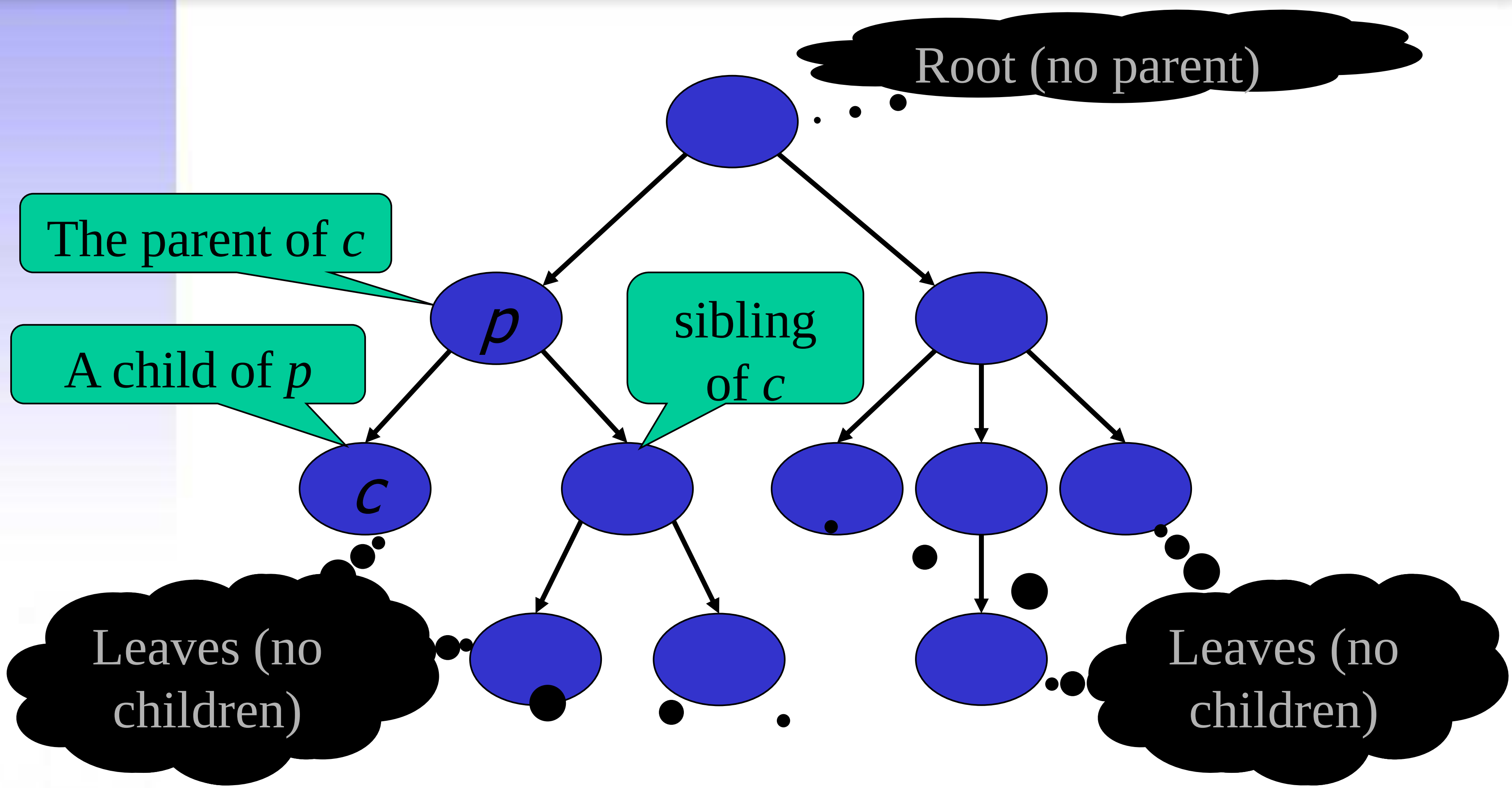
Recursive definition

Non-recursive definition



- A tree consists of a set of **nodes** and a set of **edges**, such that:
  - There is a special node called **root**
  - For each node *c*, except for the root, there is one edge from another node *p* (*p* is **parent** of *c*, *c* is one of the children of *p*)
  - For each node there is a **unique path** (sequence of edges) from the root
  - Nodes without children are called **leaves**

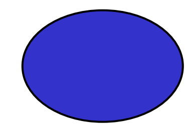
Example



Root (no parent)
The parent of c
A child of p
p
sibling of c
c
Leaves (no children)
Leaves (no children)

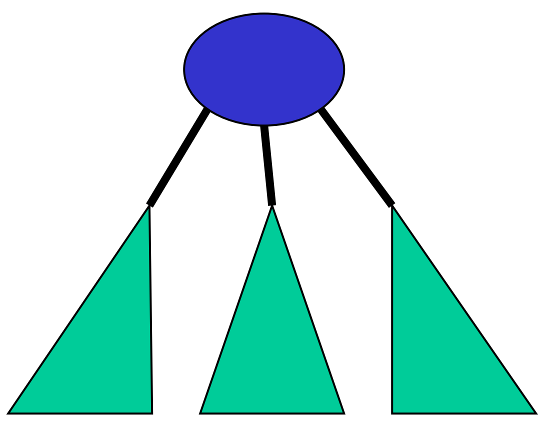
Recursive definition (1)

- A tree is
  - A node

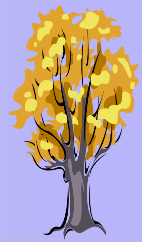


  - or a node and subtrees connected to the node by means of an edge to its root

Doesn't include the empty tree



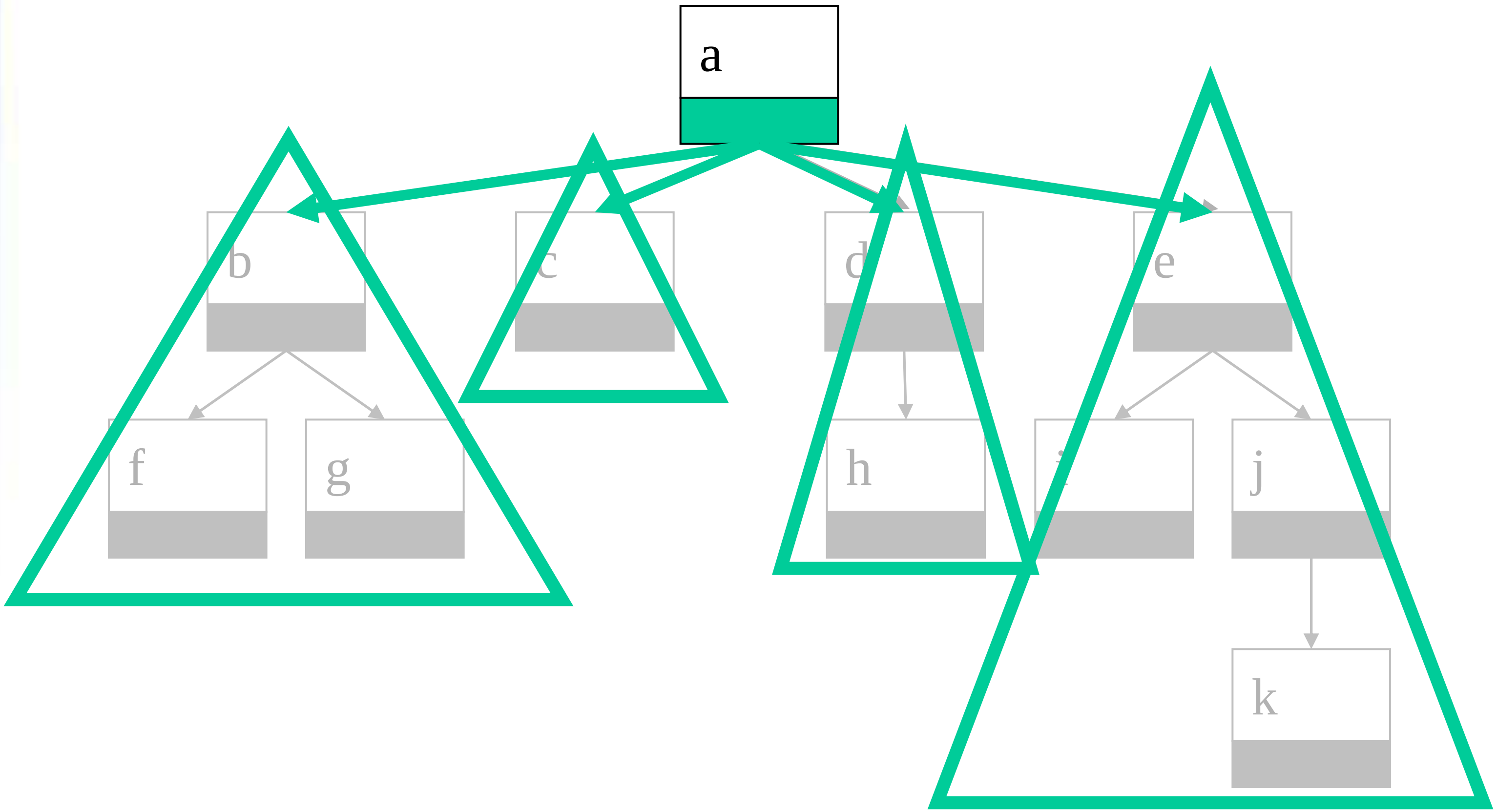
Recursive definition (2)



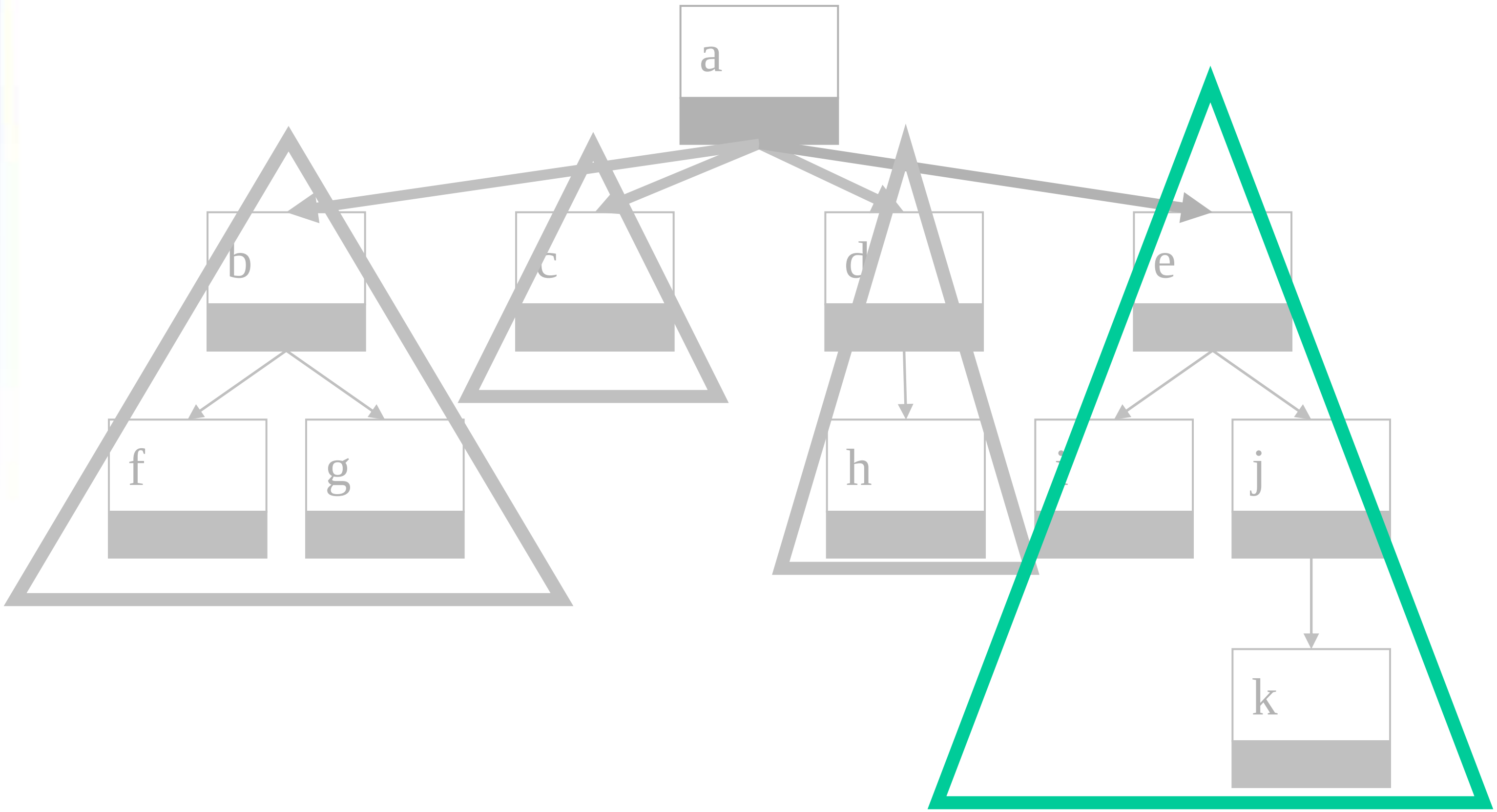
- A tree is
  - empty
  - or a node and zero or more non-empty subtrees connected to the node by means of an edge to its root

Trees

Recursive definition

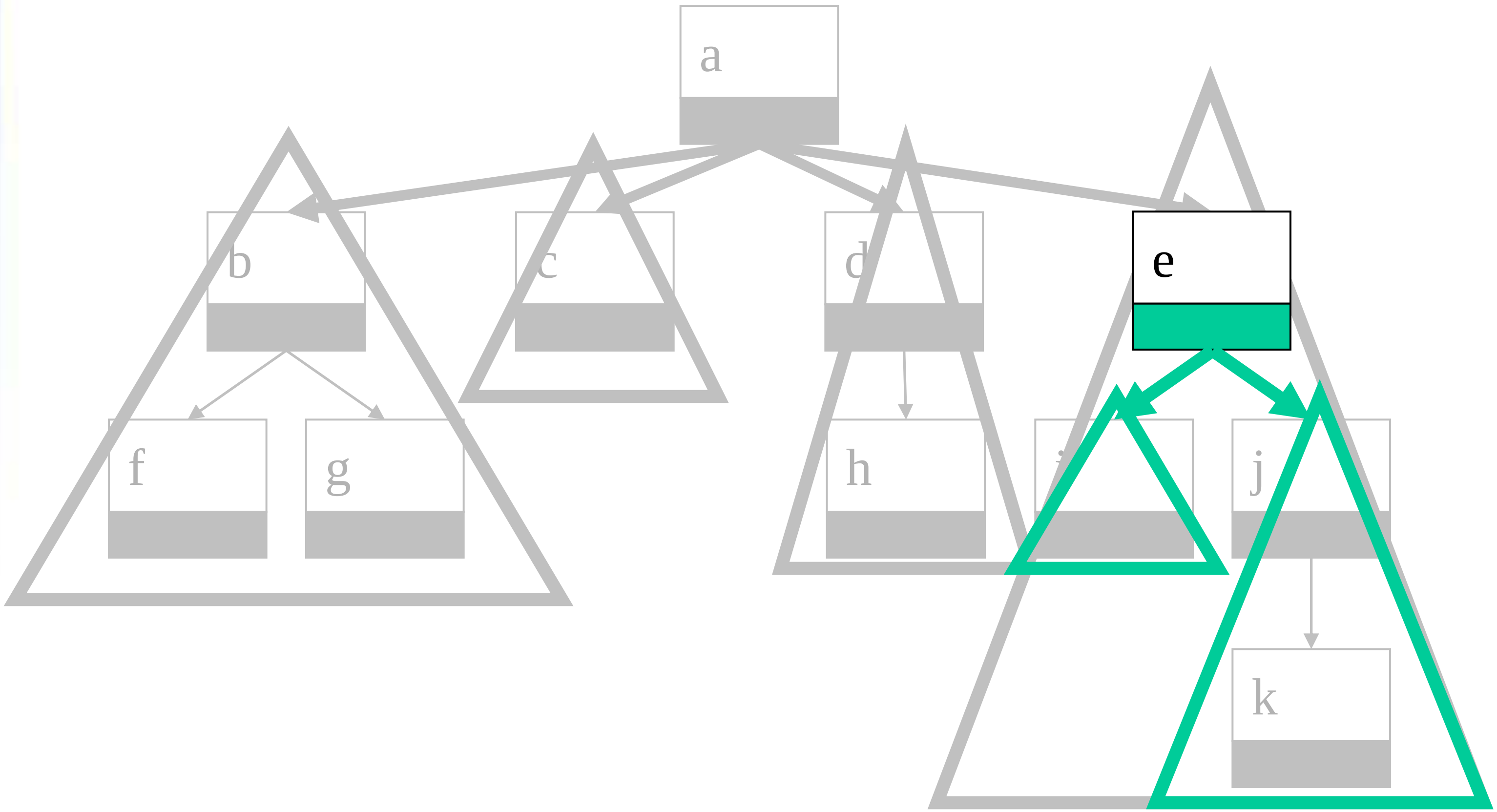


Trees
Recursive definition



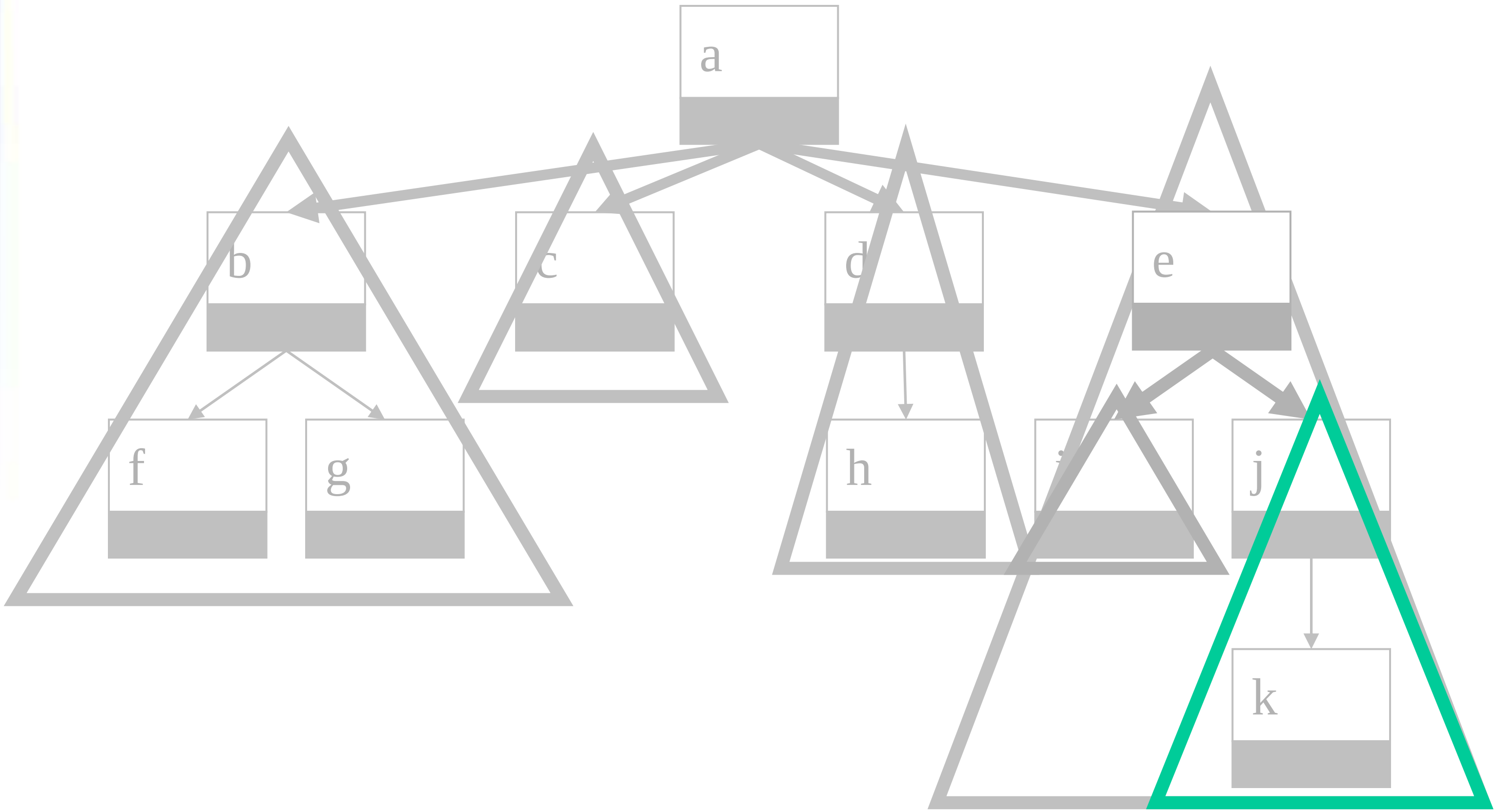
Trees

Recursive definition



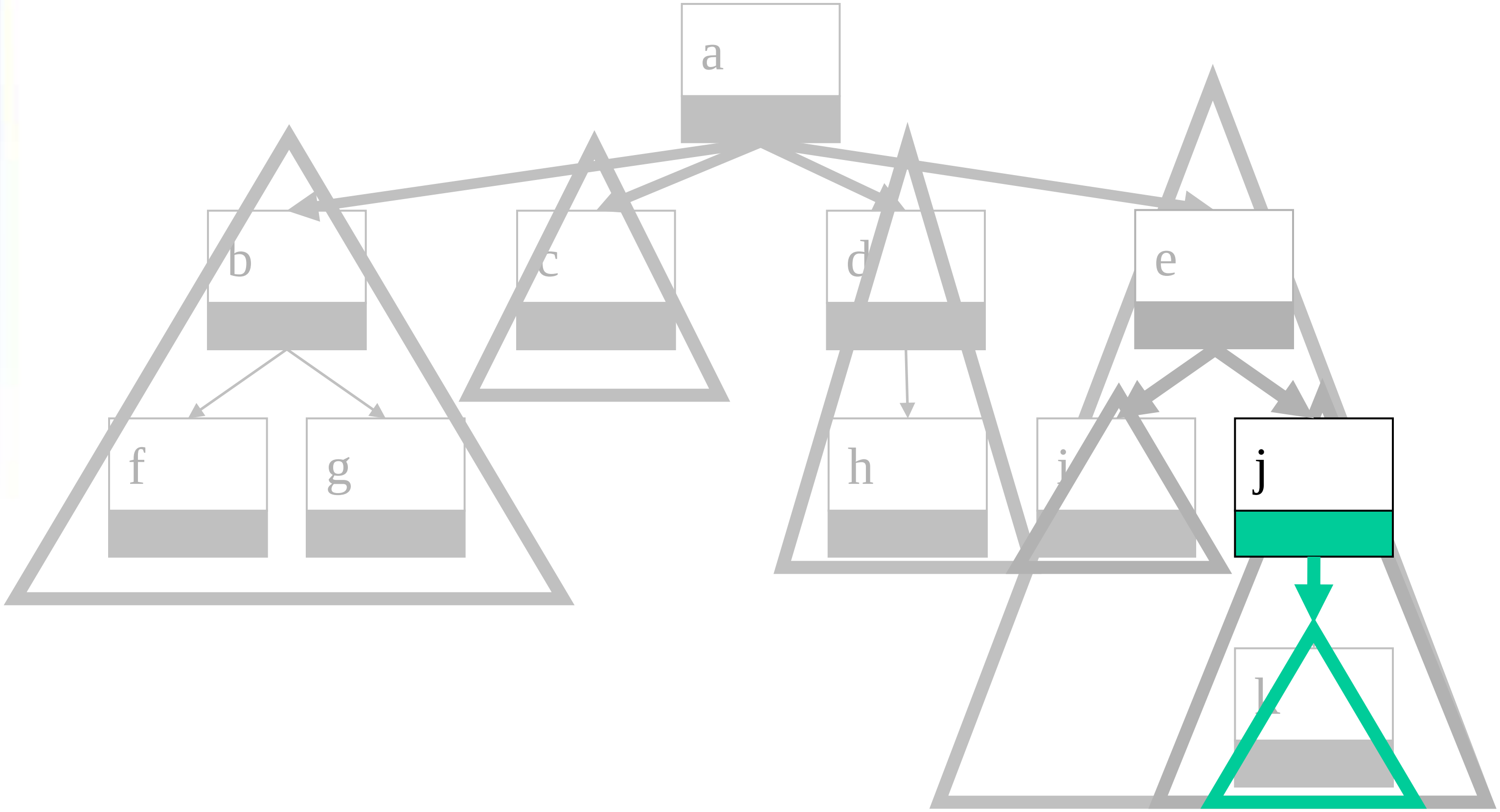
Trees

Recursive definition



Trees

Recursive definition



Terminology

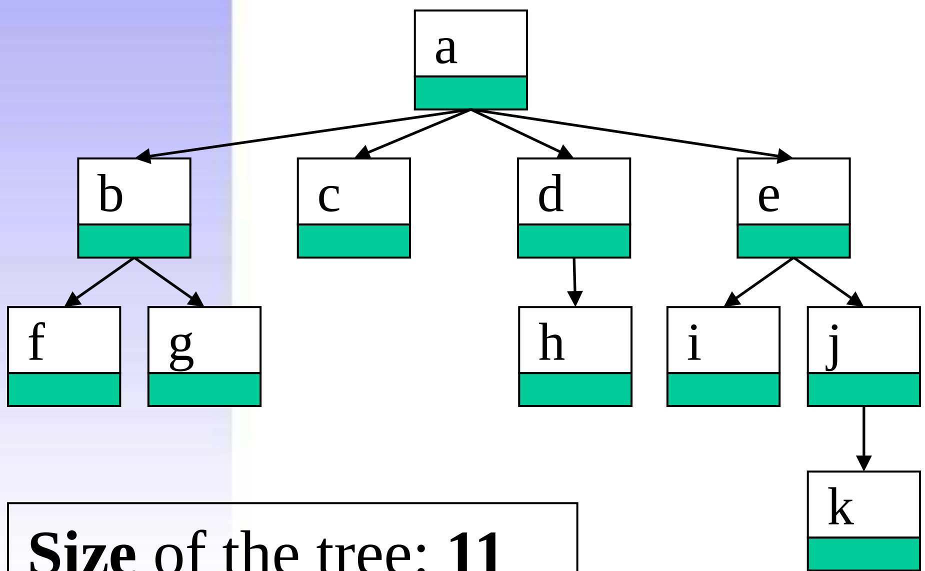
- A node is **external**, if it doesn't have children (it is a ***leaf***)
- A node is **internal**, if it has one or more children
- A node is **ancestor** of another one, if it is its parent or an ancestor of its parent
- A node is **descendent** of another one, if the latter is ancestor of the former
- The descendents of a node determine a **subtree** where this node acts as the root

Terminology

- A **path** from one node to another one, is a sequence of consecutive edges between the nodes.
  - Its *length* is the number of edges it is composed of.
- The **depth** of a node is the length of the path from the root to this node.
- The **height** of a tree is the depth of the deepest node.
- The **size** of a tree is the number of nodes.

Example

Terminology and properties



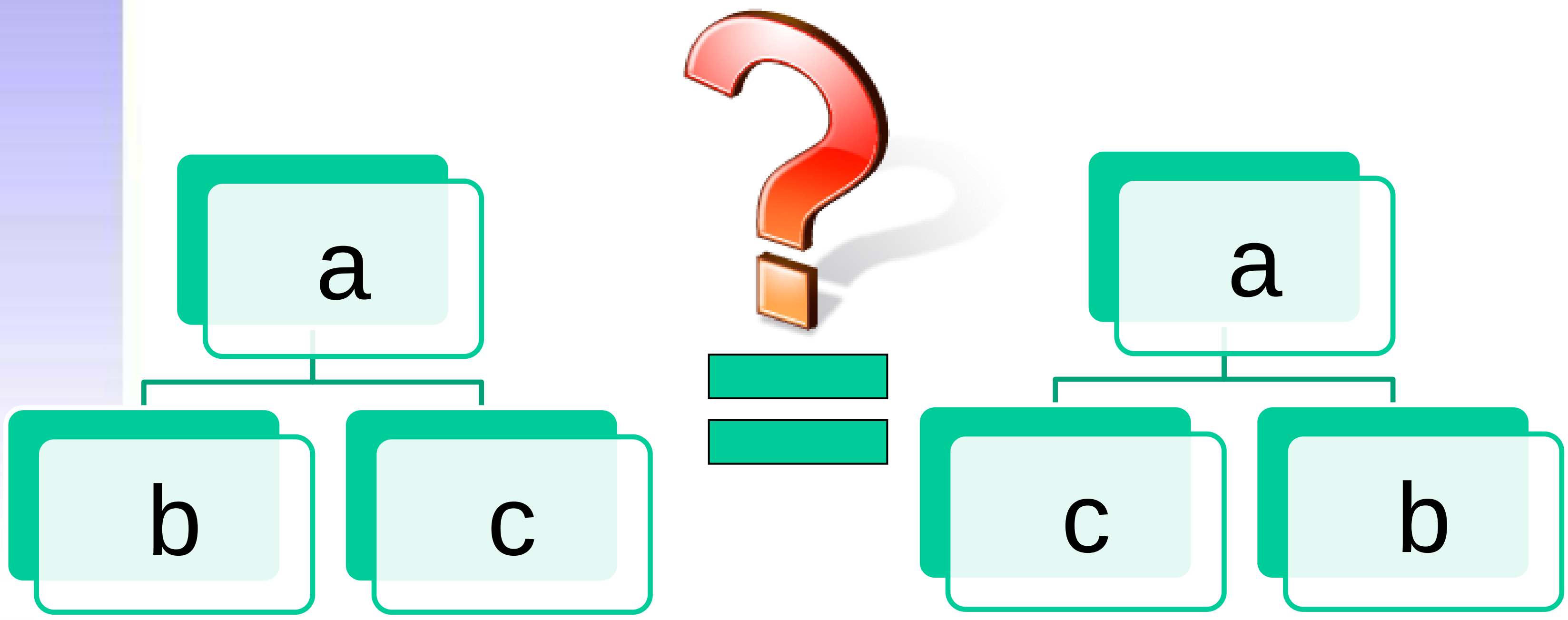
Size of the tree: **11**

Height of the tree: **3**

Node	Height	Depth	Size	Internal / External
a	**3**	0	**11**	Internal
b				
c	0	1	1	External
d	1	1	2	Internal
e				
f	0	2	1	External
g	0	2	1	External
h				
i				
j				
k				

Terminology

Ordered tree



- A tree is **ordered**, if for each node there exists a linear ordering for its children.

Terminology

Binary tree

- A **binary** tree is an ordered tree, where each node has 0, 1 or 2 children (the left and the right child).
  - Full binary tree: each node is either a leaf or possesses exactly 2 children
  - Complete binary tree: all levels except possibly the last are full, and the last level has all its nodes to the left side

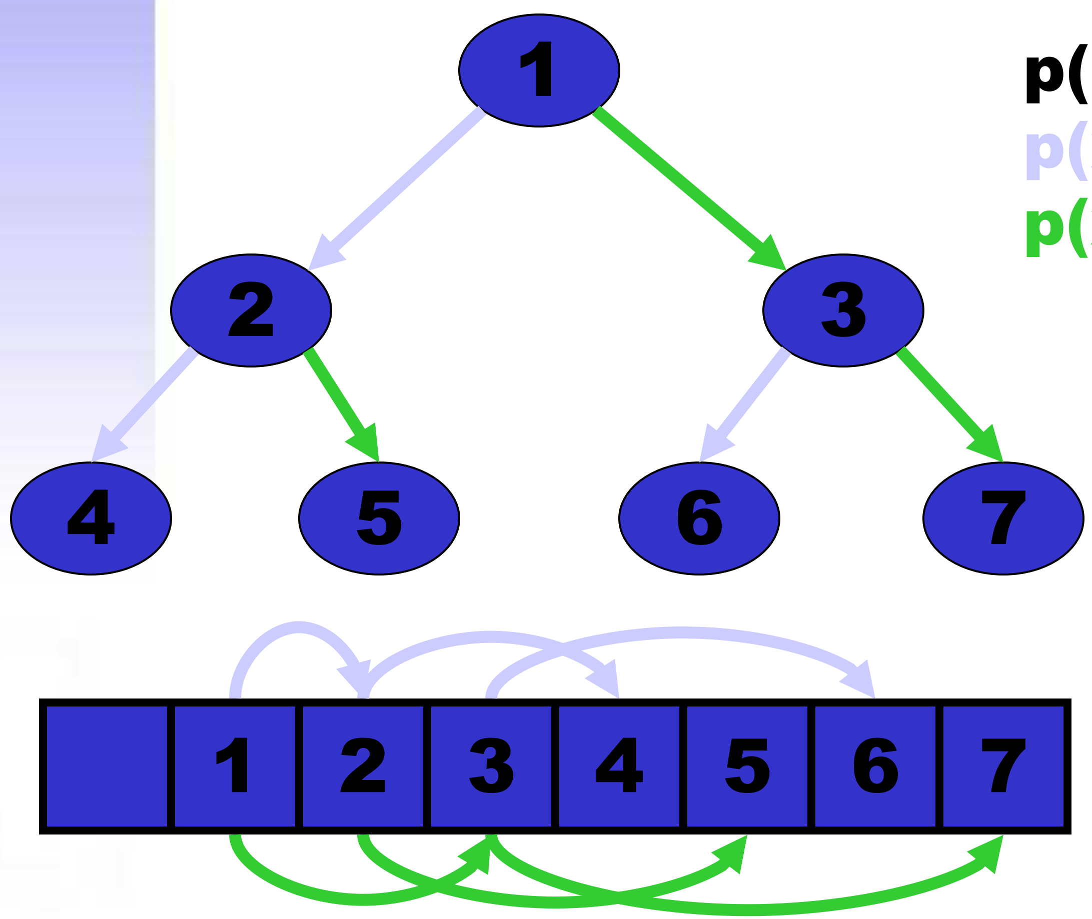
BTree interface

```
public interface BTree<E> {
  static final int LEFT = 0;
  static final int RIGHT = 1;

  boolean isEmpty();
  E getInfo() throws BTreeException;
  BTree<E> getLeft() throws BTreeException;
  BTree<E> getRight() throws BTreeException;

  void insert(BTree<E> tree, int side) throws BTreeException;
  BTree<E> extract(int side) throws BTreeException;

  String toStringPreOrder();
  String toStringInOrder();
  String toStringPostOrder();
  String toString(); // preorder

  int size();
  int height();

  boolean equals(BTree<E> tree);
  boolean find(BTree<E> tree);
}
```

Implementations

- Array-based implementation
- Based on a linked structure

Array-based implementation



p(root)=1

p(x.left)=2*p(x)

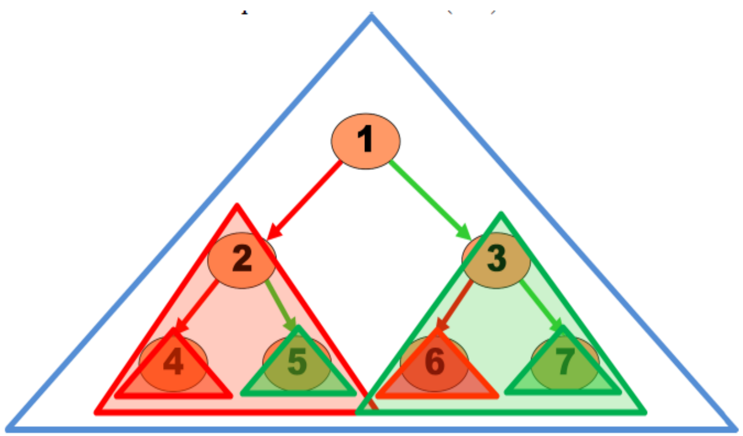
p(x.right)=2*p(x)+1

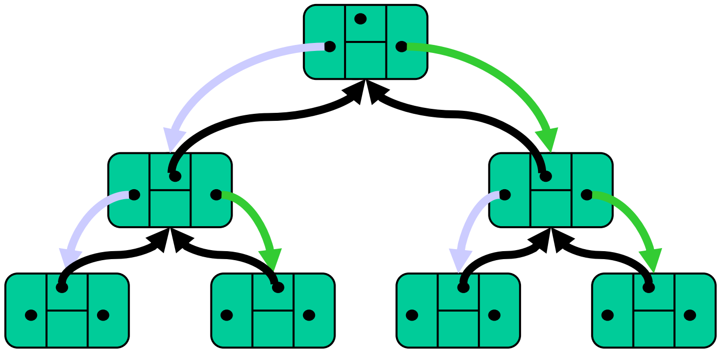
Implementation based on a linked structure

Linked Binary Node (`LBNode`)

Linked Binary Tree (`LBTree`)

- Each tree (`LBTree`) has a root node (`LBNode` attribute)
- Each root node `LBNode` contains two trees (`LBTree` attributes), which can be empty (null)

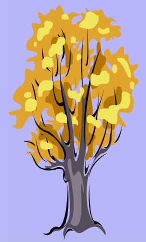




LBNode class (recursive tree)

```
public class LBNode<E> {
  private E info;
  private BTree<E> left;
  private BTree<E> right;

  public LBNode(E info, BTree<E> left, BTree<E> right) {…}

  public E getInfo() {…}
  public void setInfo(E info) {…}

  public BTree<E> getLeft() {…}
  public void setLeft(BTree<E> left) {…}

  public BTree<E> getRight() {…}
  public void setRight(BTree<E> right){…}
}
```

LBTree class... (recursive tree)



```
public class LBTree<E> implements BTree<E> {

  private LBNode<E> root;

  public LBTree() {
    root = null;
  }

  public LBTree(E info) {
    root = new LBNode<E>(info, new LBTree<E>, new
 LBTree<E>);
  }

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

… LBTree class... (recursive tree)

```
public E getInfo() throws BTreeException {
  if (isEmpty()) {
   throw new BTreeException("empty trees do not have info");
  }
  return root.getInfo();
}
public BTree<E> getLeft() throws BTreeException {
  if (isEmpty()) {
   throw new BTreeException("empty trees do not have a left child");
  }
  return root.getLeft();
}
public BTree<E> getRight() throws BTreeException {
  if (isEmpty()) {
   throw new BTreeException("empty trees do not have a right child");
  }
  return root.getRight();
}
```

Binary node... (non-recursive)

```
public class LBNode<E> {
  private E info;
   private LBNode<E> left;
   private LBNode<E> right;

   public LBNode() {
     this(null);
   }
  public LBNode(E info) {
     this(info,null,null);
   }
   public LBNode(E info, LBNode<E> l, LBNode<E> r) {
     this.info = info;
     left = l;
     right = r;
   }
}
```

Basic algorithms

- Size (number of nodes)
- Depth of a node
- Height
- Traversals
  - Euler
  - Pre-, in- and post-order

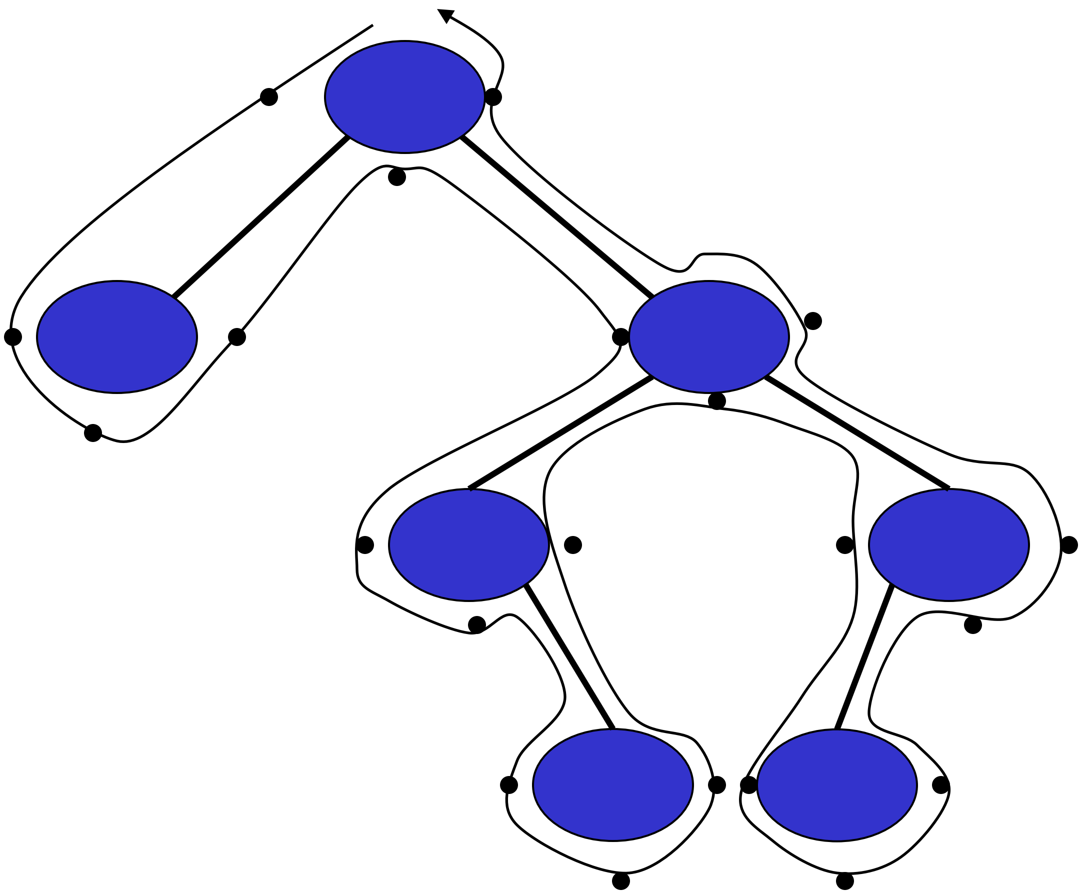
(To simplify, we assume binary trees)

… LBTree class… (recursive tree)

```
public int size() {
  if (isEmpty())
    return 0;
  else
    return 1 + root.getLeft().size() + root.getRight().size();
}

public int height() {
  if (isEmpty())
    return -1;
  else {
    int leftHeight = root.getLeft().height();
    int rightHeight = root.getRight().height();
    if (leftHeight > rightHeight)
     return 1 + leftHeight;
    else
     return 1 + rightHeight;
  }
}
```

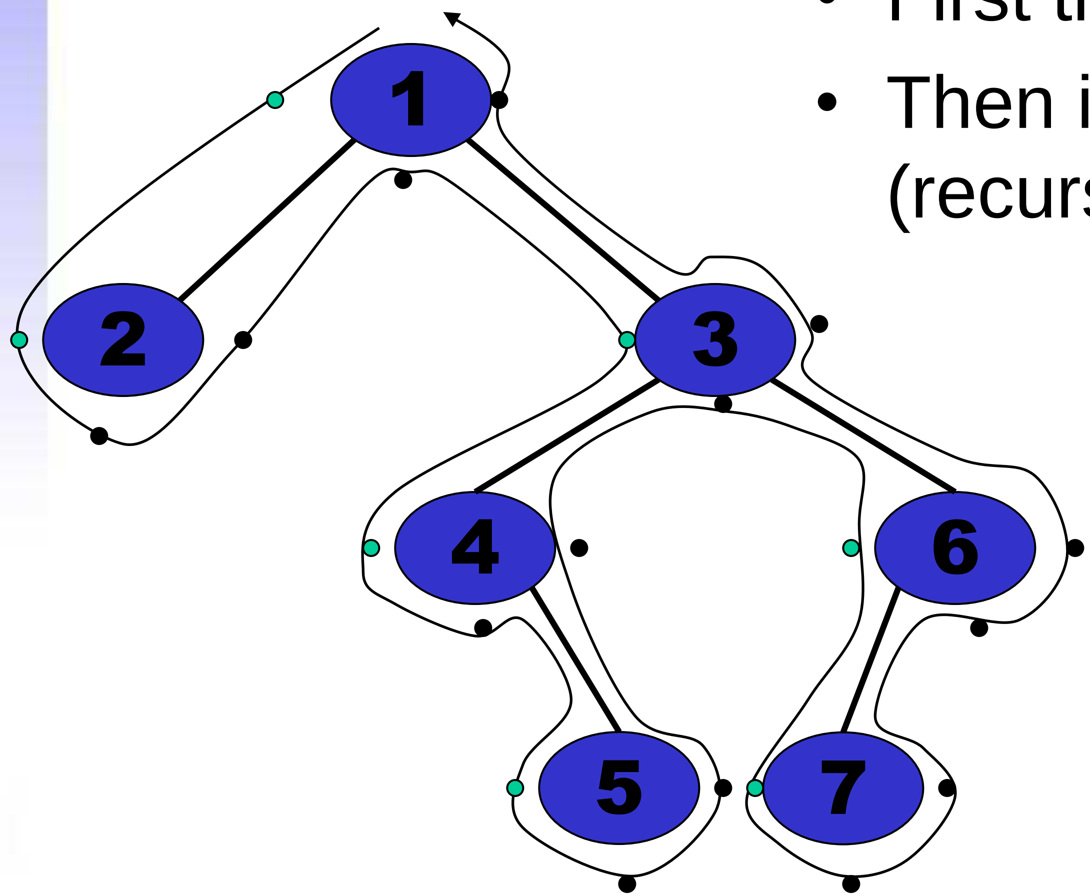
```
1+Math.max(leftHeight, rightHeight);
```

Euler traversal



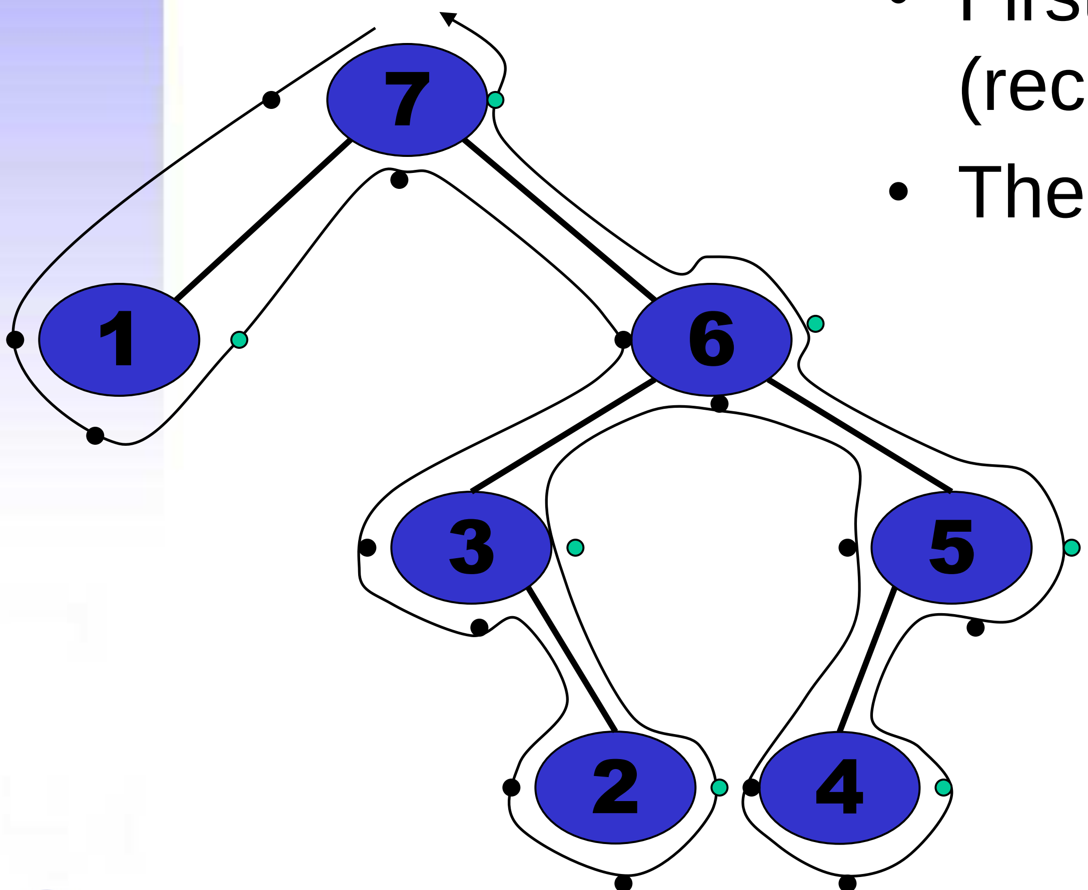
Also applicable
to non-binary trees!

Preorder traversal



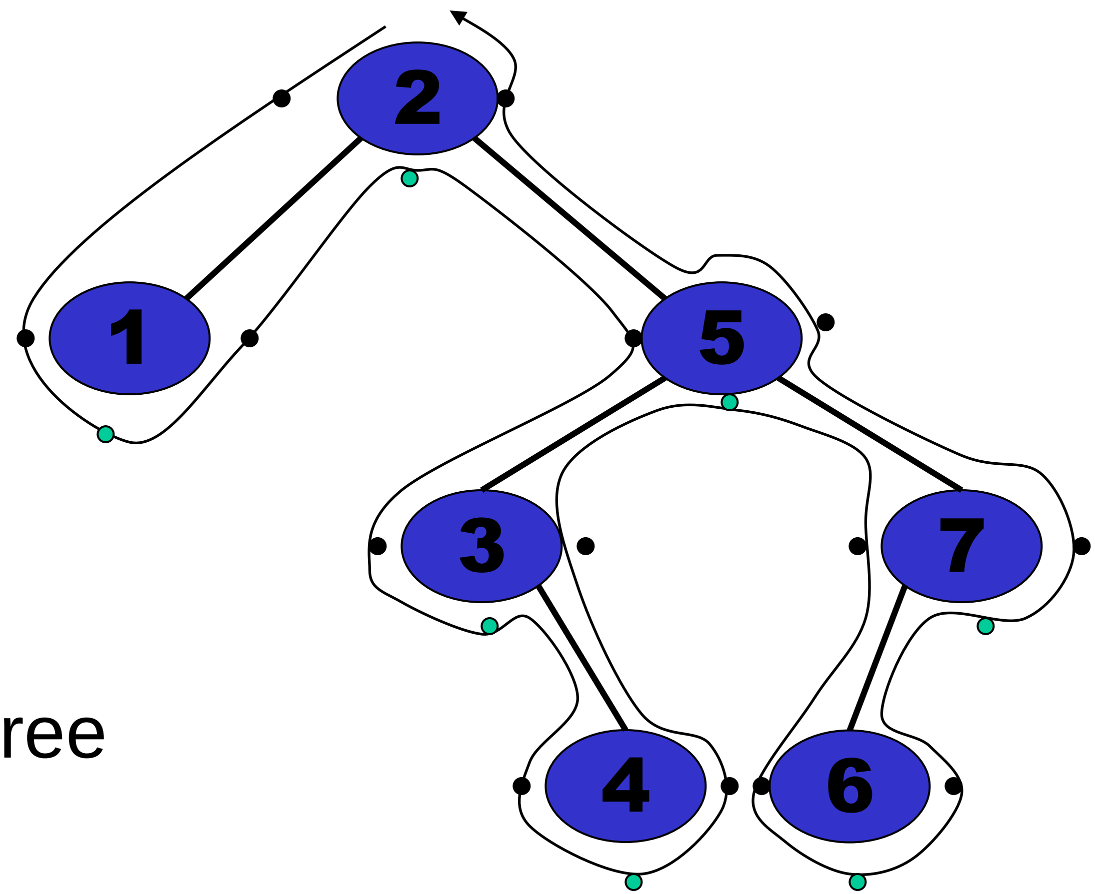
- First the node
- Then its children (recursively)

Postorder traversal



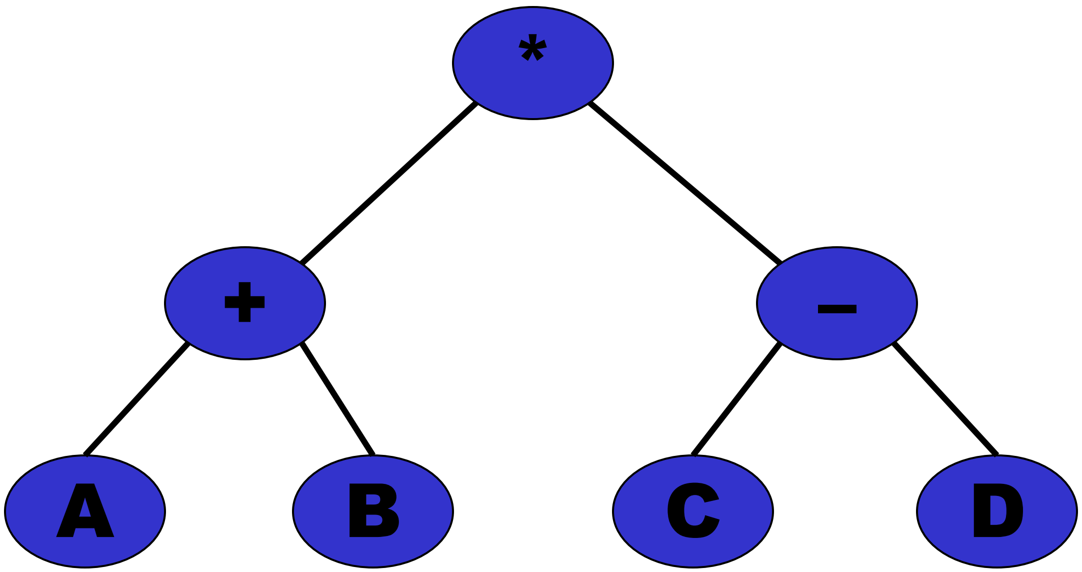
- First the children trees (recursively)
- Then the node

Inorder (symmetric) traversal



- First the left tree (recursively)
- Then the node
- Finally, the right tree (recursively)

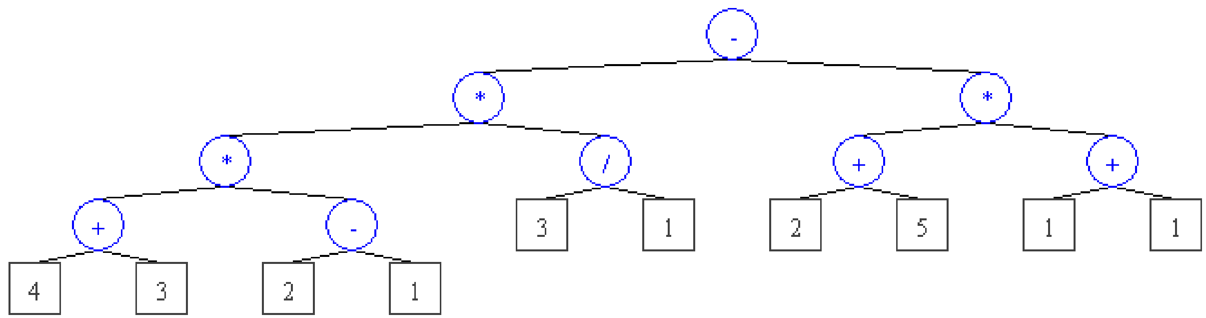
(A+B)*(C–D)



Example

Infix	Prefix	Postfix
A+B	+AB	AB+
A+B–C	–+ABC	AB+C–
(A+B)*(C–D)	*+AB–CD	AB+CD–*

Activity



- Visualize expressions as trees
  http://www.cs.jhu.edu/~goodrich/dsa/05trees/Demo1/

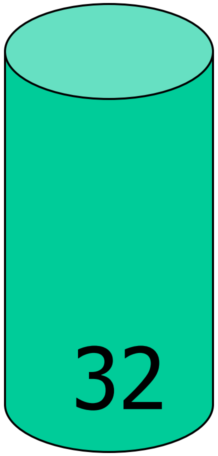
Evaluate

((((4+3)*(2-1))*(3/1))-((2+5)*(1+1)))

Result is 7

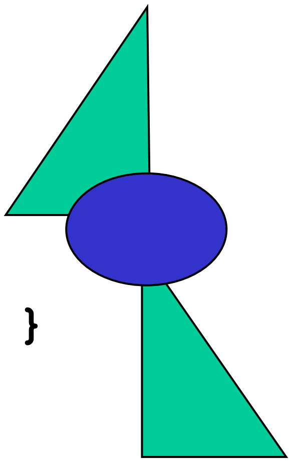
Postfix notation

- HP calculators, RPN=Reverse Polish Notation
- Stack to store objects
- Eg: ~~3~~ ~~5~~ ~~+~~ ~~6~~ ~~2~~ ~~–~~ ~~*~~



LBTree class: inorder (recursive tree)

```
public String toStringInOrder() {
  if (isEmpty()) {
    return "";
  } else {
    return root.getLeft().toStringInOrder() +
           root.getInfo().toString() + " " +
           root.getRight().toStringInOrder(); }
 }
```



Properties of binary trees

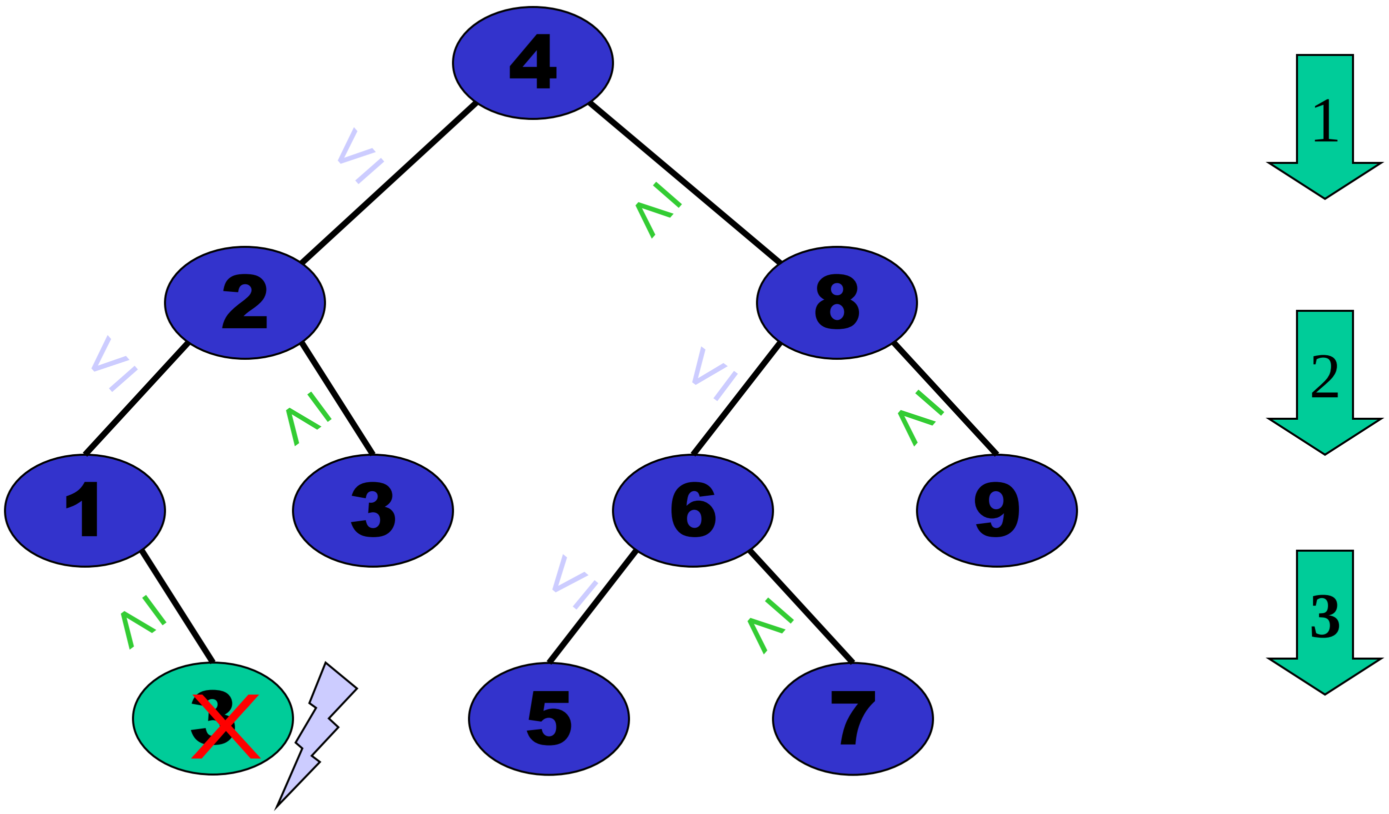
- Let
  - E=Number of external nodes
  - I=Number of internal nodes
  - N=Size=E+I
  - H=Height
- then
  - $E=I+1$
  - $H+1 \leq E \leq 2^H$ $\quad H \leq I \leq 2^H-1$ $\quad 2*H+1 \leq N \leq 2^{H+1}-1$
  - $\log_2(N+1)-1 \leq H \leq (N-1)/2$

Universidad Carlos III de Madrid

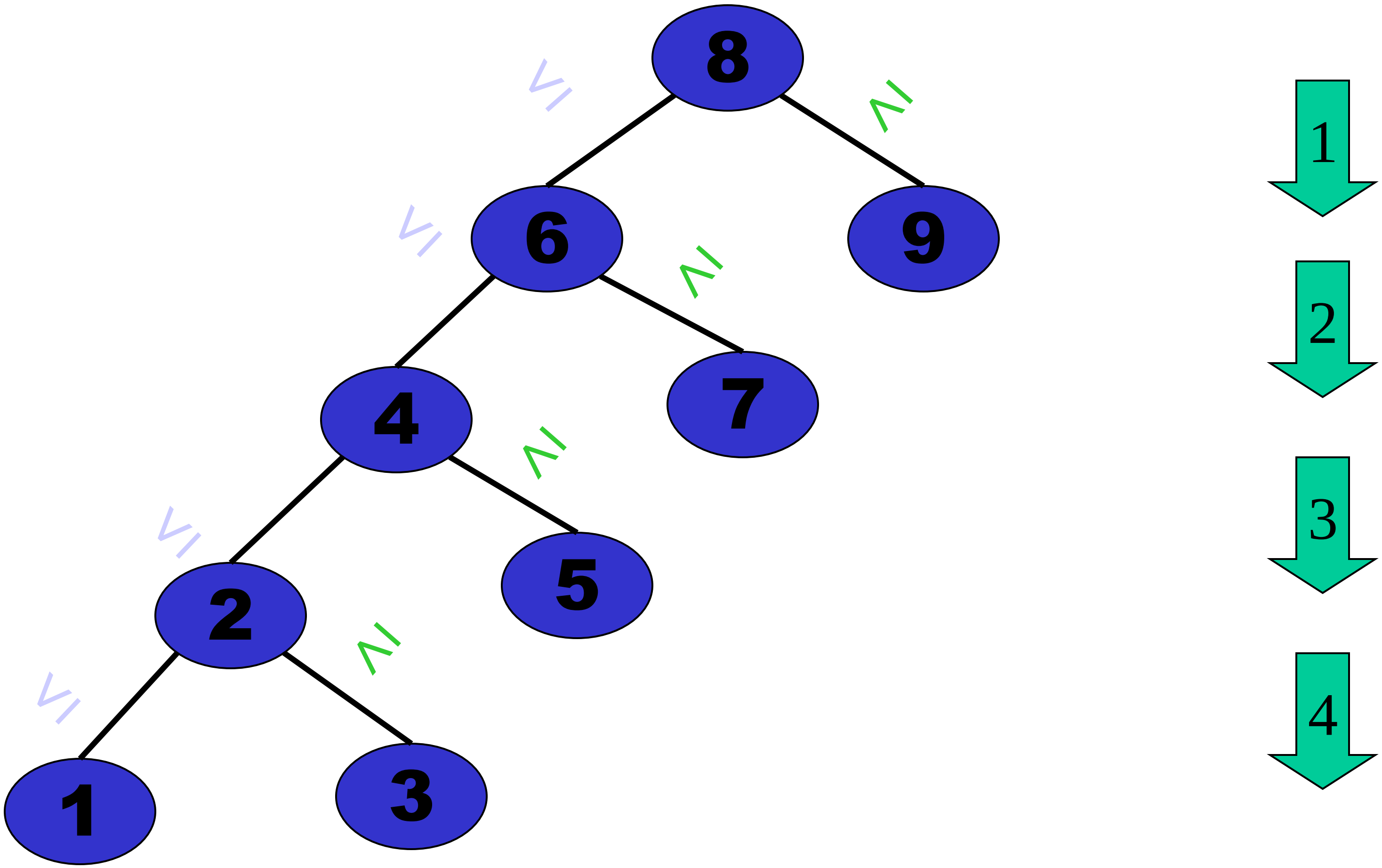
Binary search trees

- A **binary search tree** is a binary tree where for each node *n*,
  - all the keys in the left subtree are **smaller** (or equal) than the key of *n*
  - and all those of the right subtree **larger** (or equal)

Example



Example



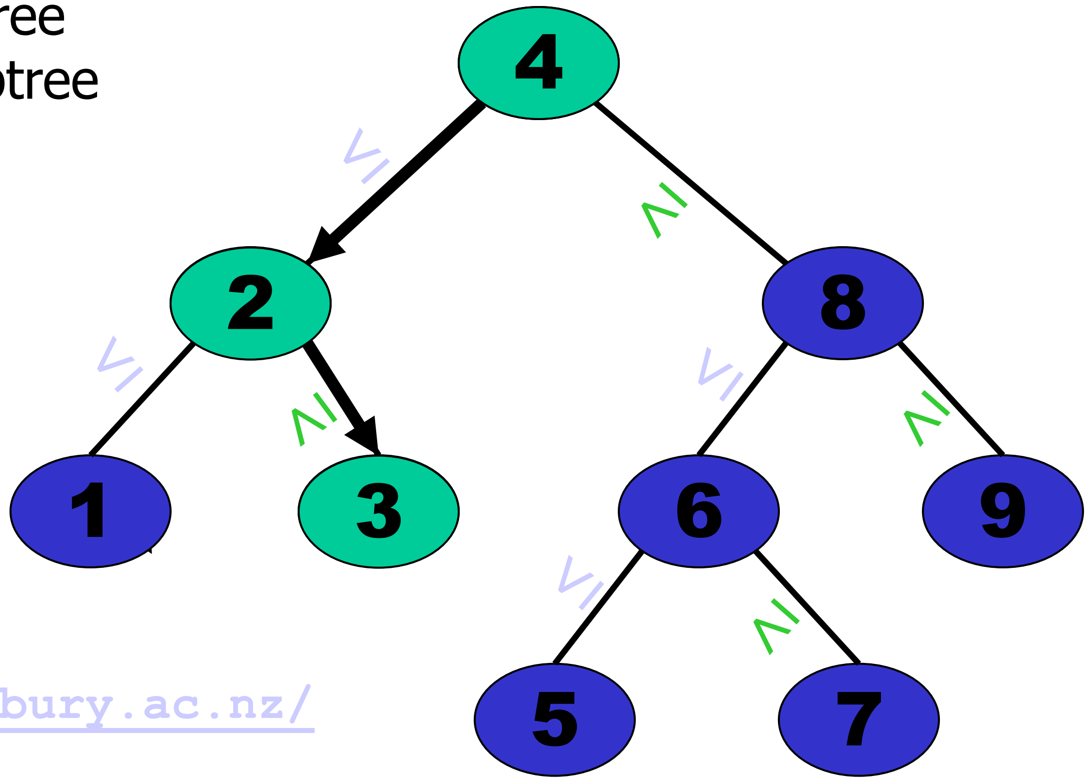
Operations

- Search
- Insertion
- Extraction

Search

Searching “3”:

- 3<4: go to left subtree
- 3>2: go to right subtree
- 3=3: element found

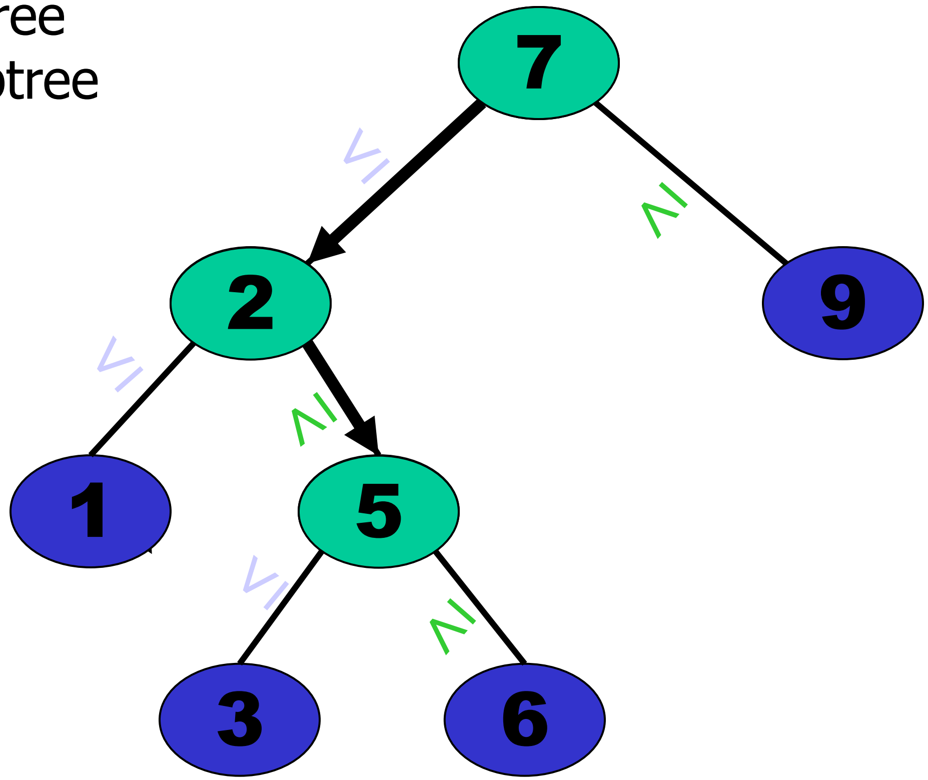


http://www.cosc.canterbury.ac.nz/mukundan/dsal/BST.html

Insertion

Inserting “6”:

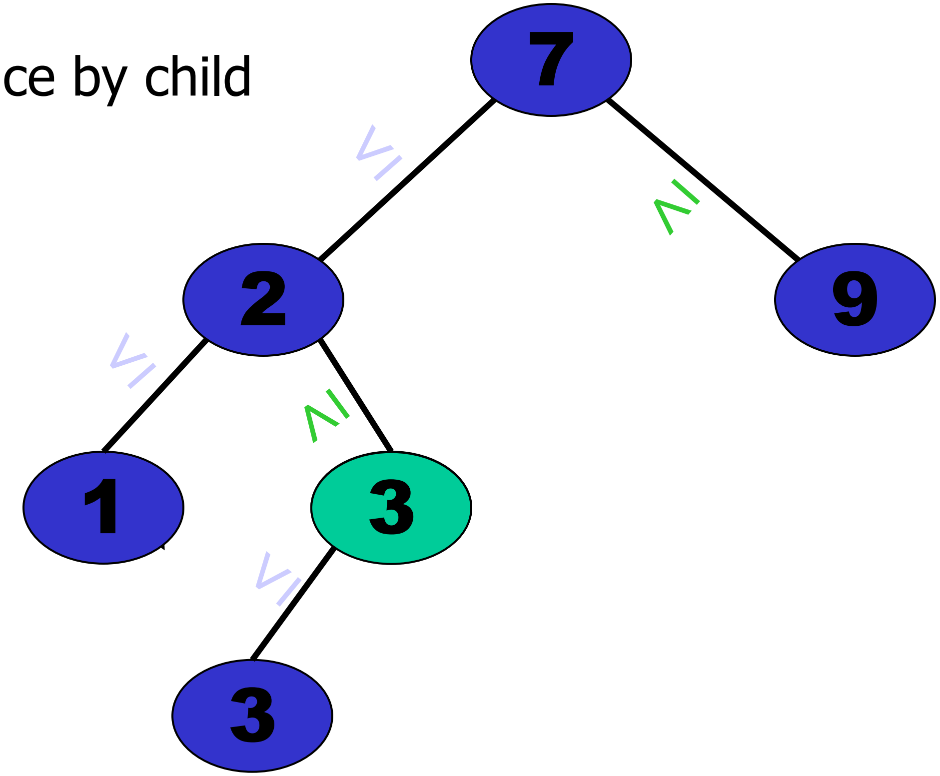
- 6<7: go to left subtree
- 6>2: go to right subtree
- when hole: insert



Extraction (1)

Extracting "5":

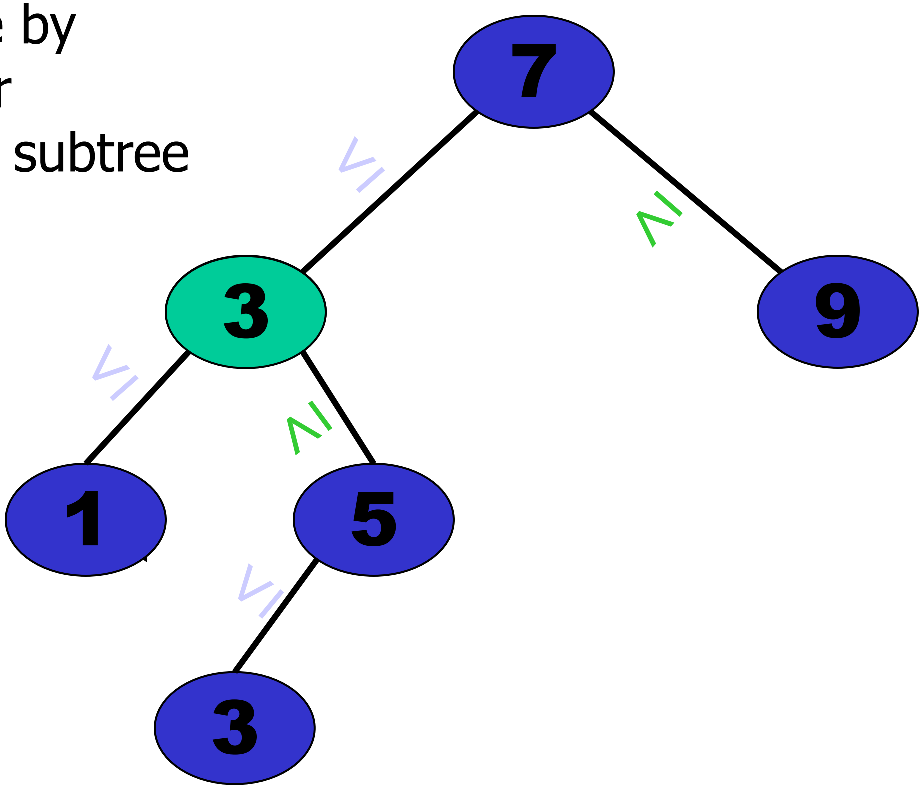
- if leaf: extract
- if degenerate: replace by child



Extraction (2)

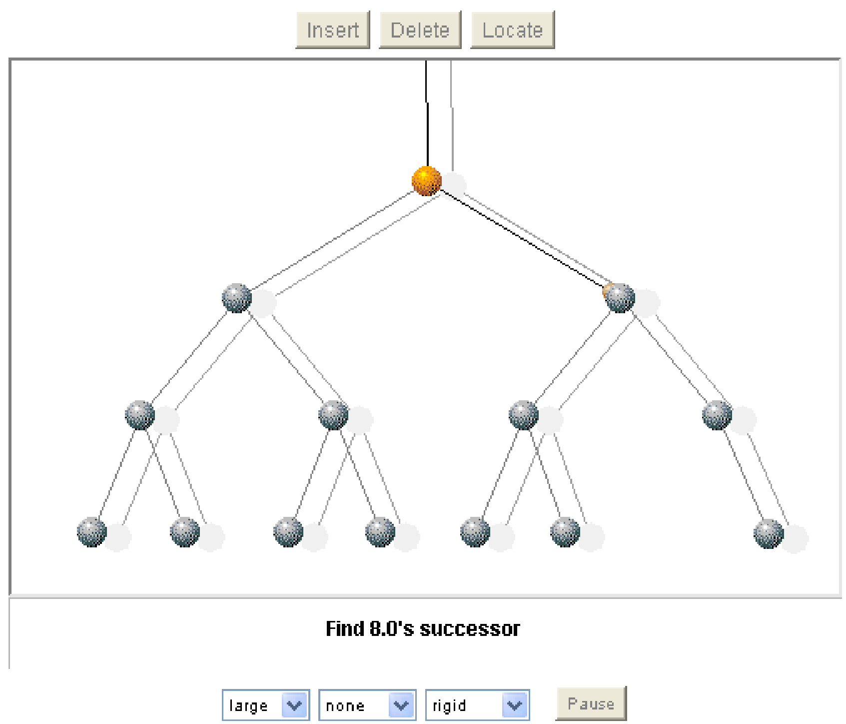
Extracting “2”:

- if 2 children: replace by
  - largest at left, or
  - smallest at right subtree



Activity

- See animation of binary search trees
  http://www.ibr.cs.tu-bs.de/courses/ss98/audii/applets/BST/BST-Example.html



Heaps

- A binary **heap** is a complete binary tree where every node has a key greater(*) than or equal to the key of its parent.
  - Usually, heaps refer to binary heaps

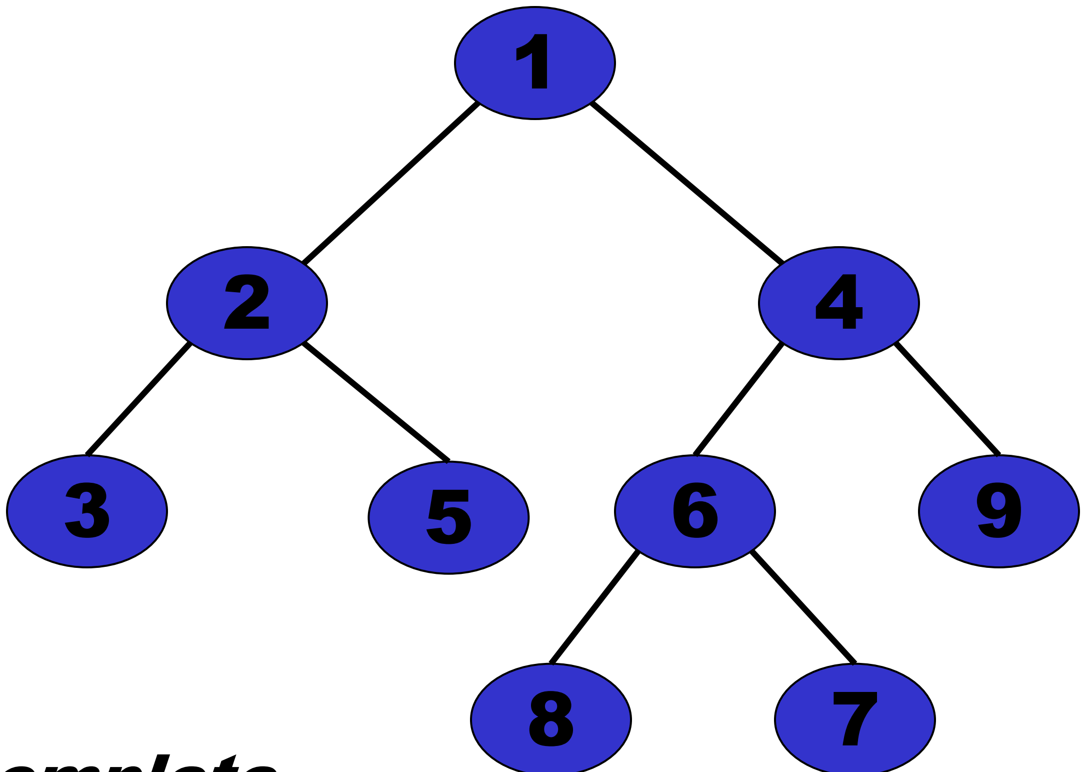
  * It could also be defined as less or equal (the order criteria is arbitrary)

- Utitity
  - Priority queues
  - Sorting algorithm

Heaps properties

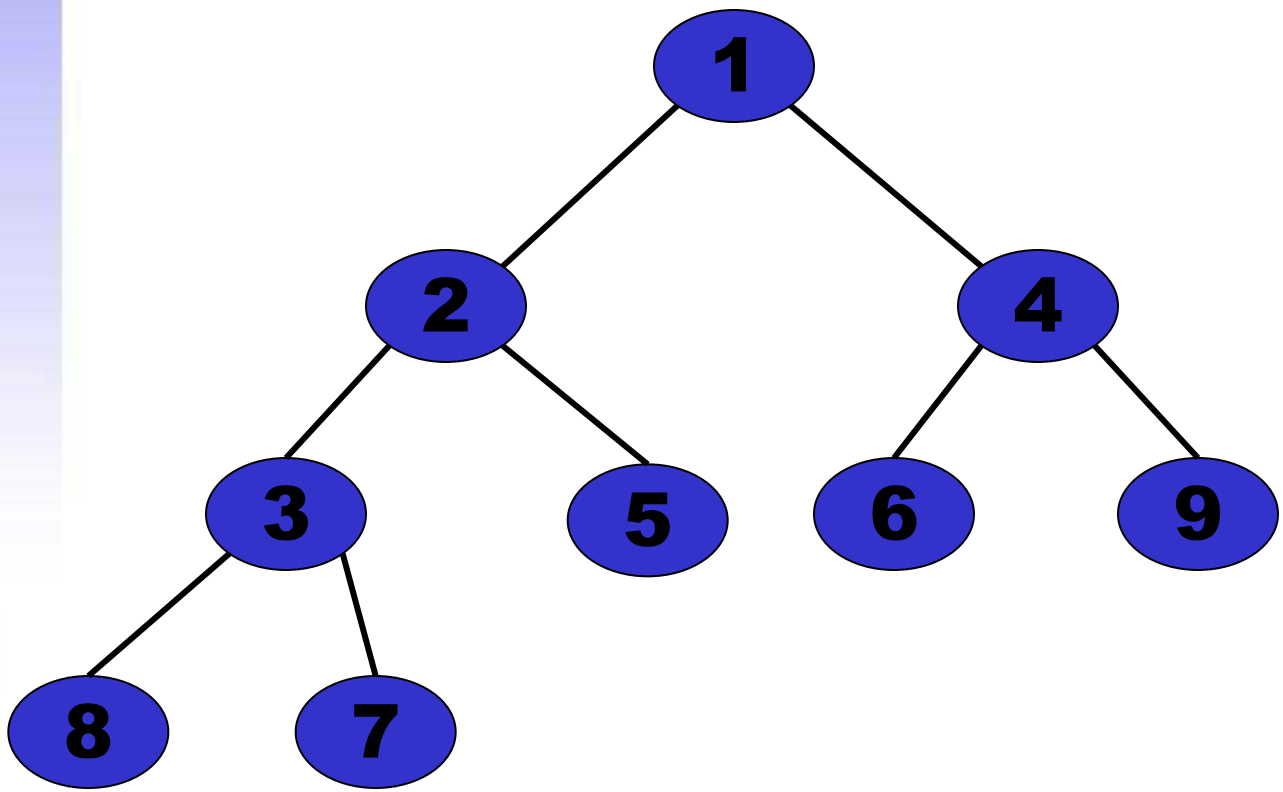
- A heap fulfils two properties:
  – Heap property: for each node *n* (except for the root), its key is larger or equal than the one of its parent.
  – Completeness

Example: heap property

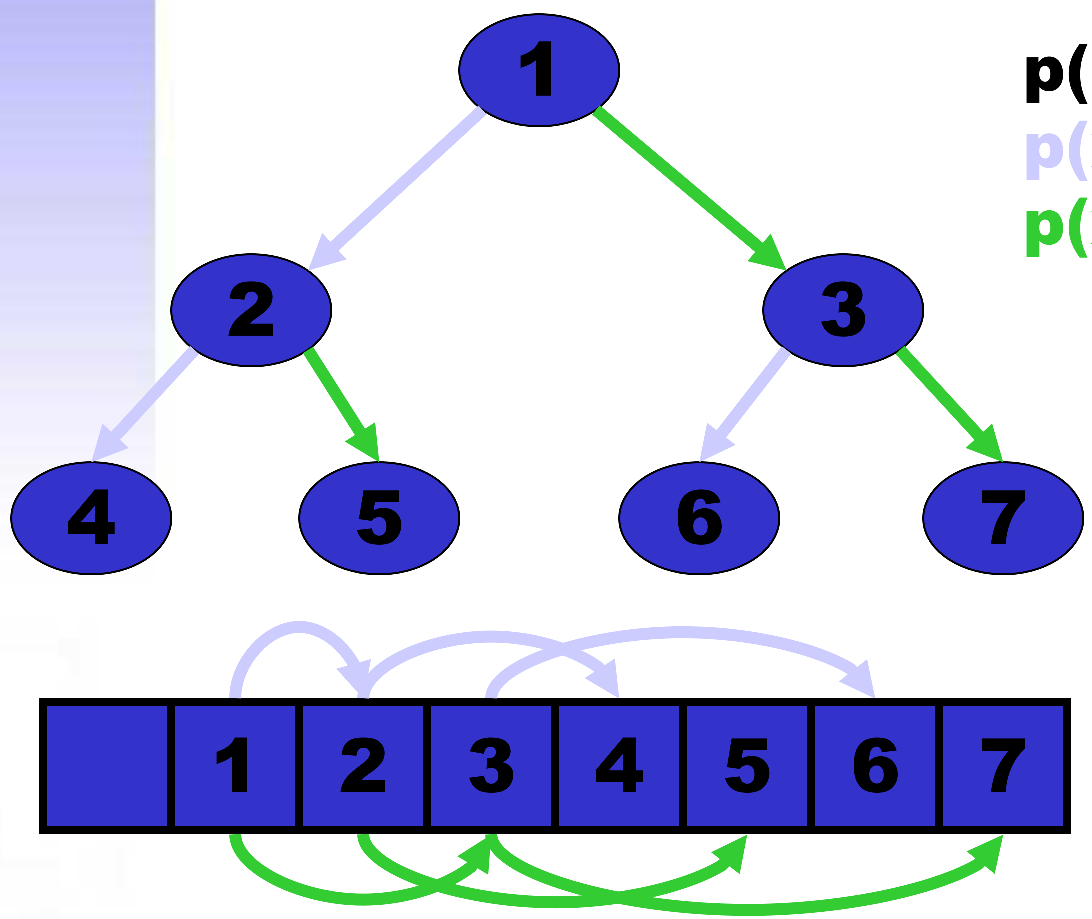


But not complete

Example: complete heap



Sequence-based implementation

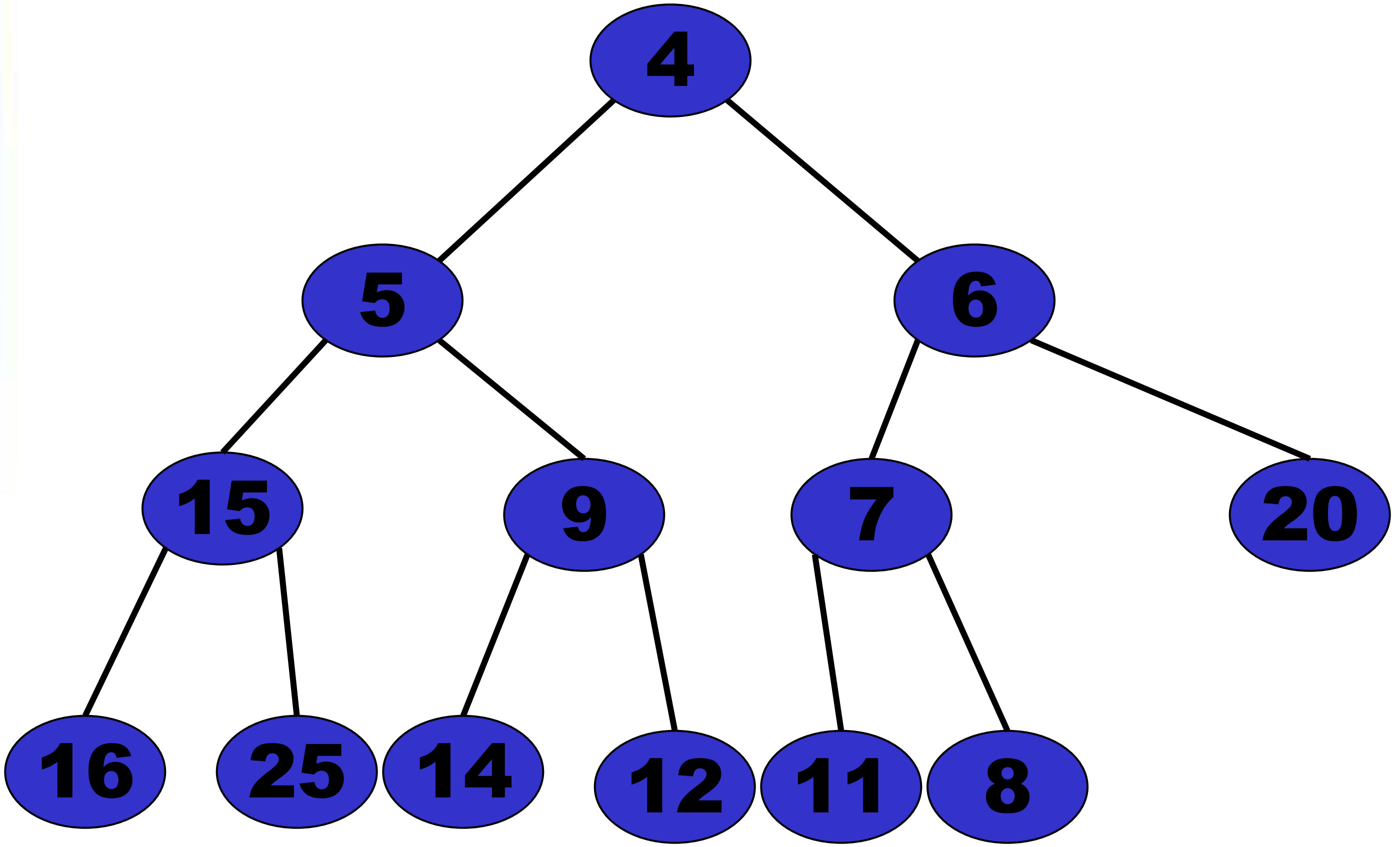


p(root)=1

p(x.left)=2*p(x)

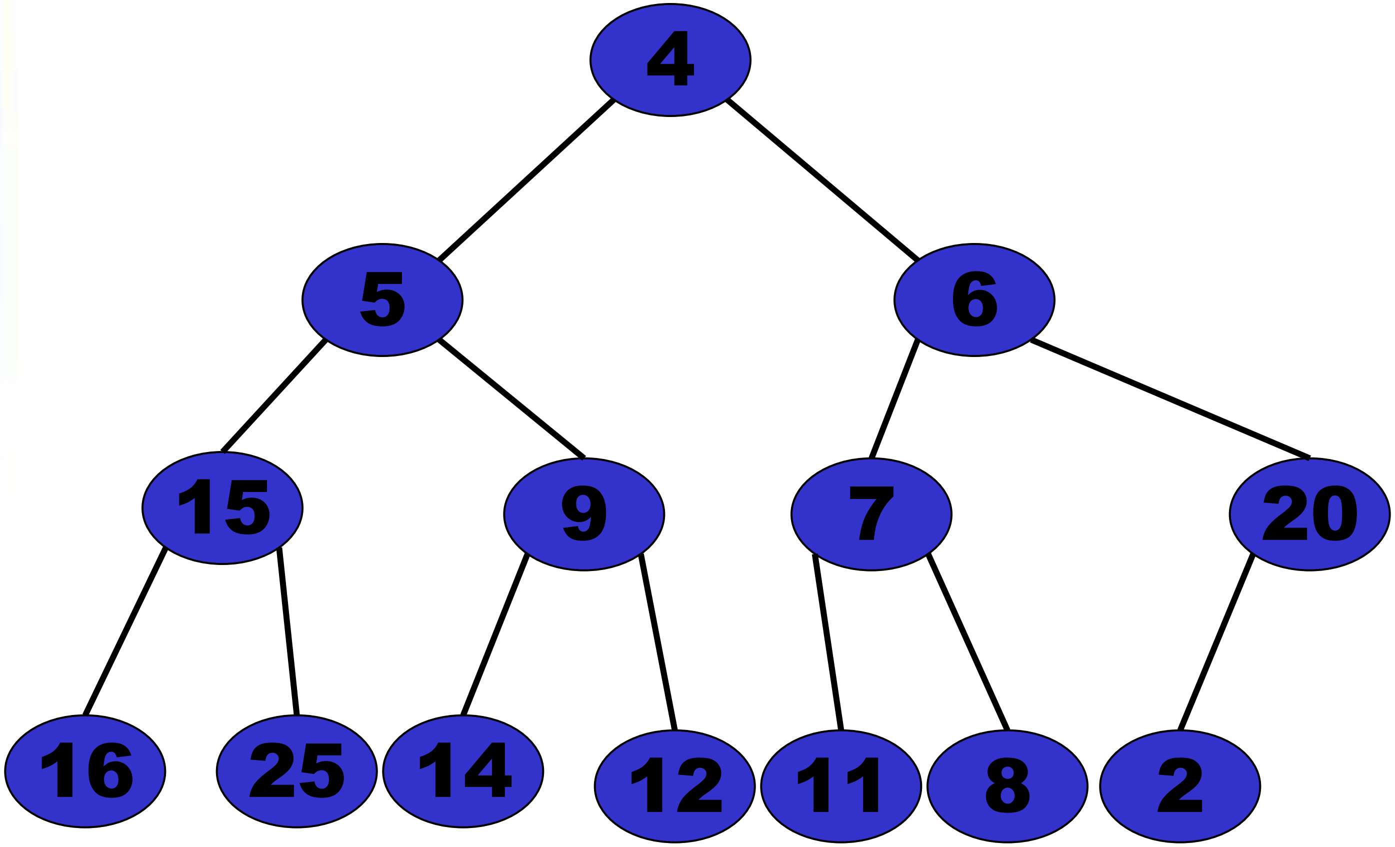
p(x.right)=2*p(x)+1

Insert

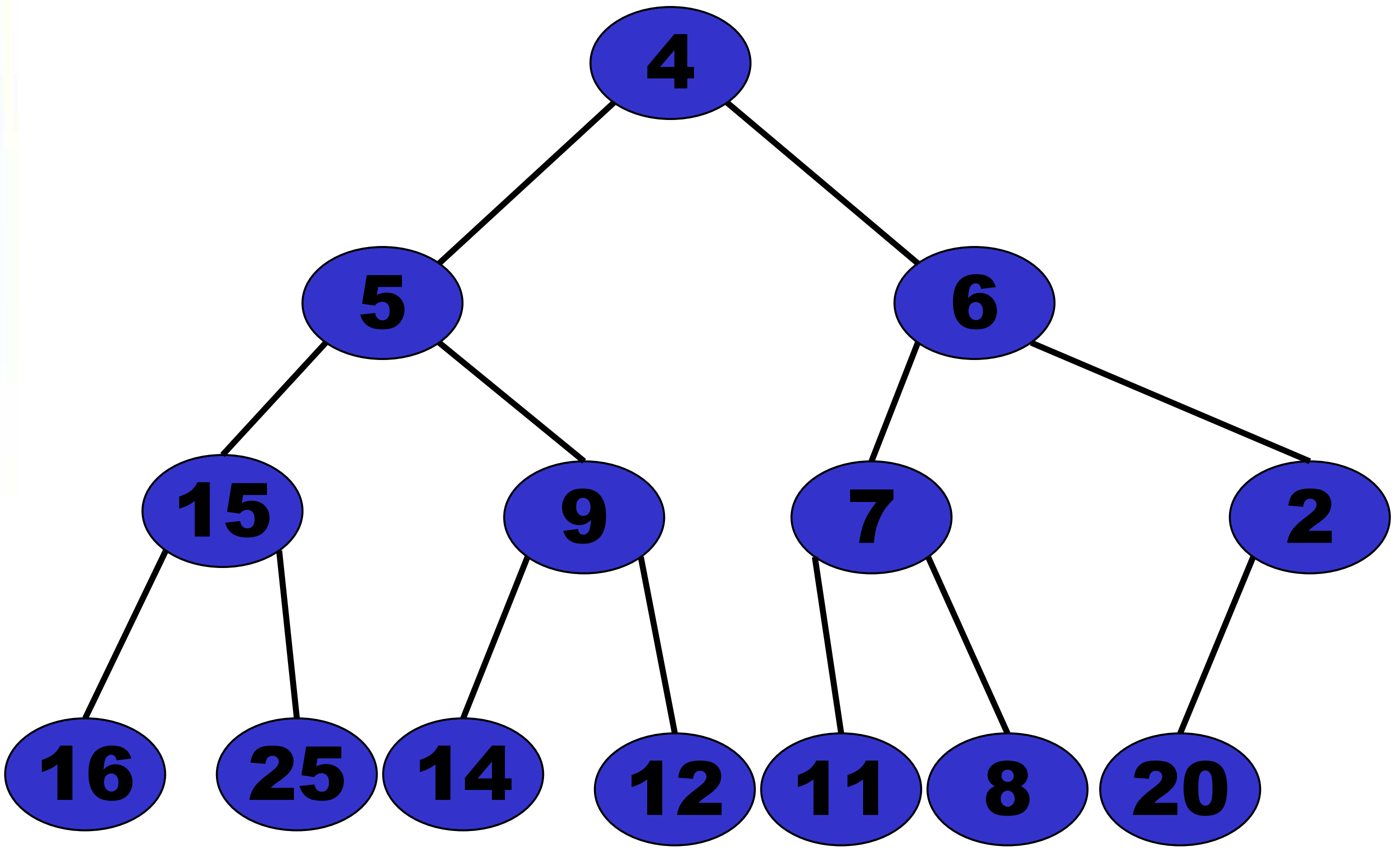


Universidad Carlos III de Madrid

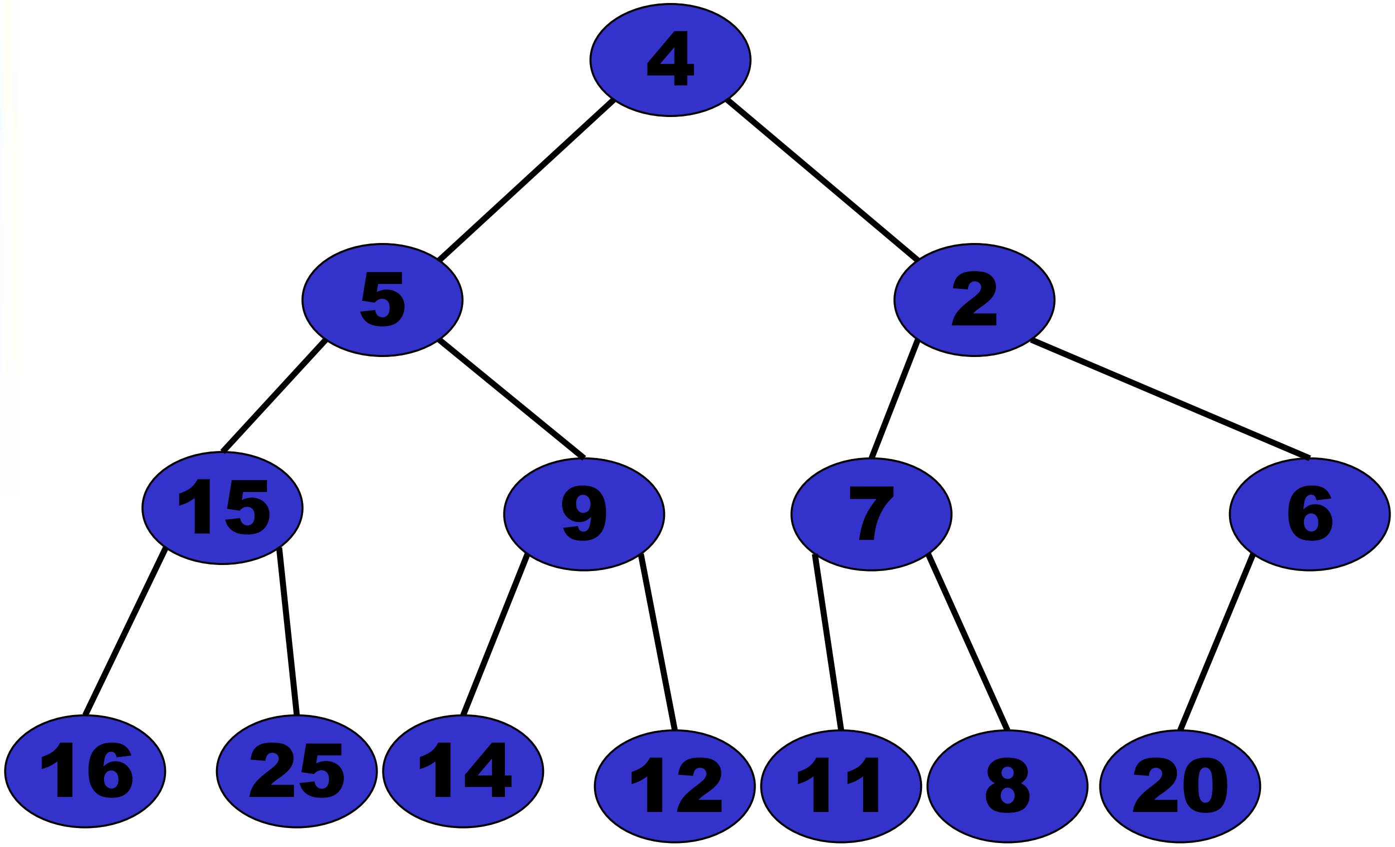
Insert



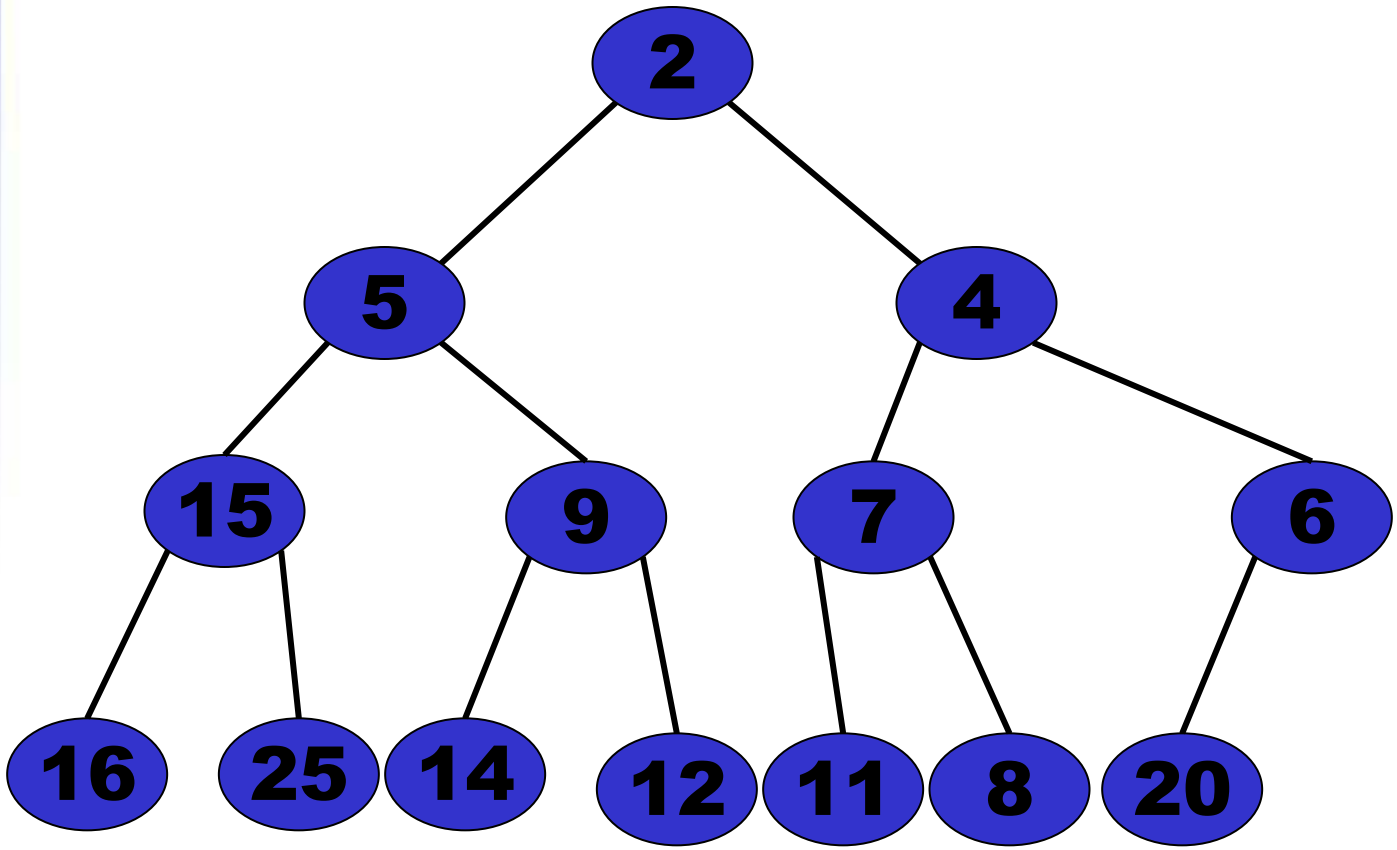
Insert



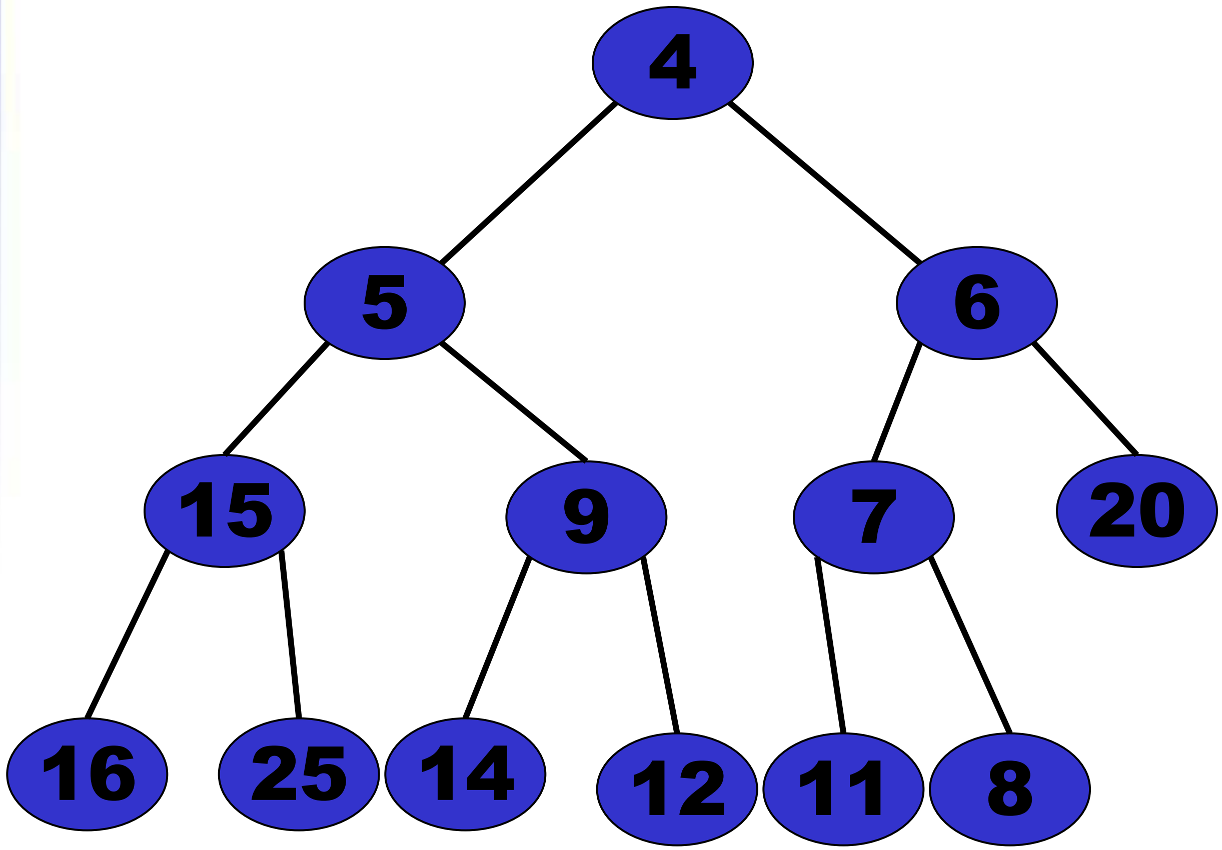
Insert



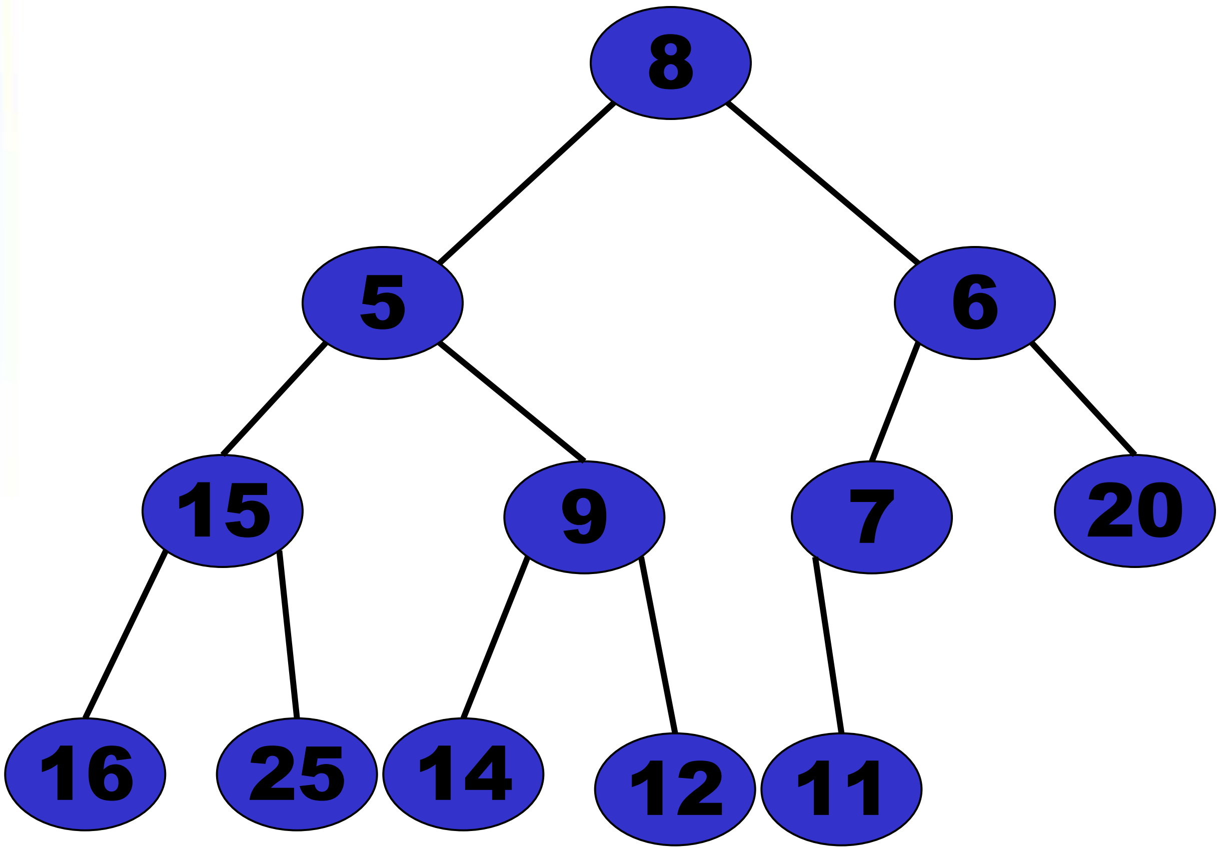
Insert



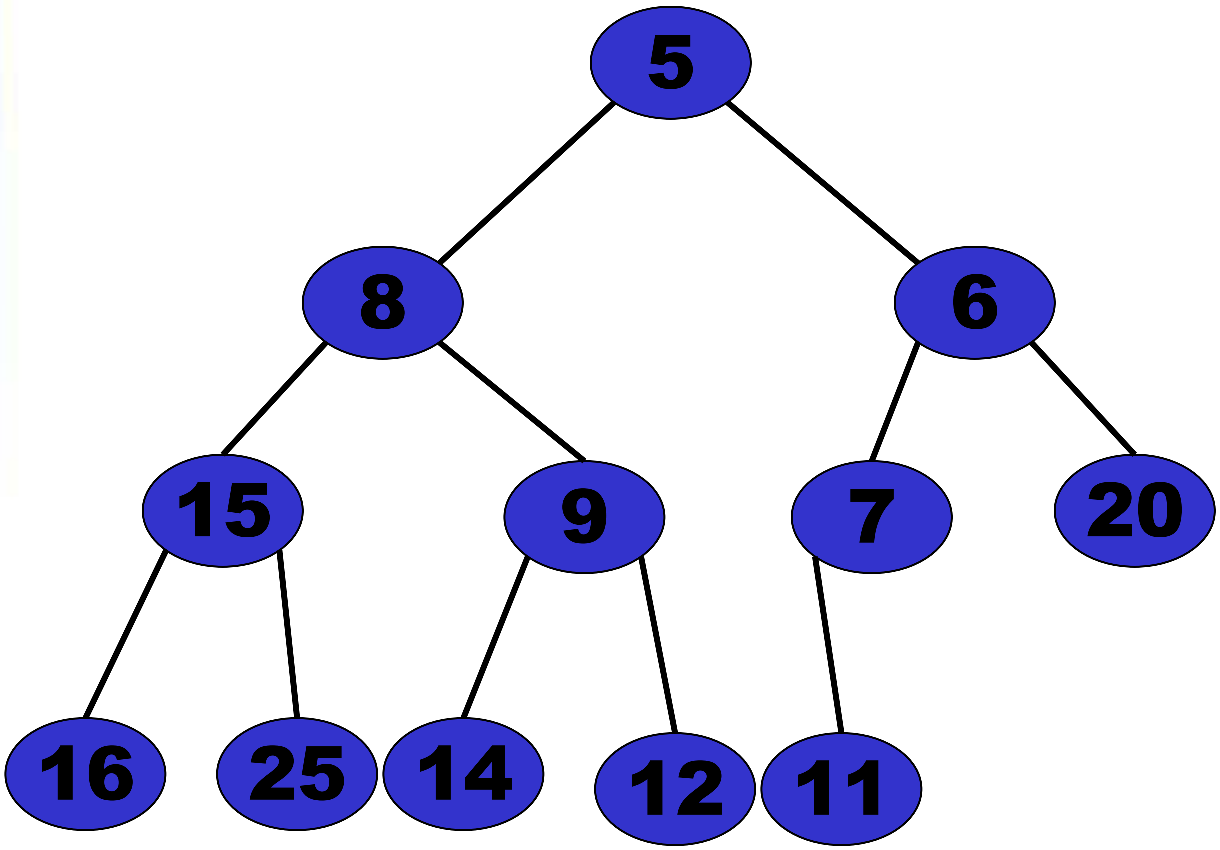
Extract



Extract

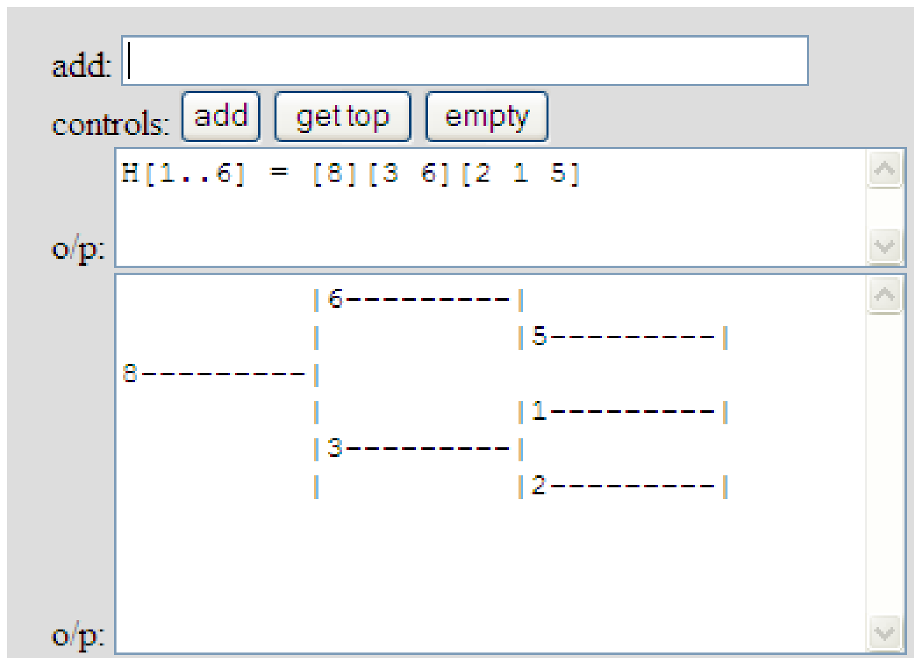


Extract



Activity

- Try out the form in http://www.csse.monash.edu.au/~lloyd/tildeAlgDS/Priority-Q/



Activity

- Try out the applet in http://www.cosc.canterbury.ac.nz/mukundan/dsal/MinHeapAppl.html