Universidad Carlos III de Madrid

Algorithms and Data Structures (ADS)
Bachelor in Informatics Engineering
Computer Science Department

General Trees.

Authors: Isabel Segura Bedmar

April 2011
Departamento de Informtica,
Laboratorio de Bases de Datos Avanzadas (LaBDA)
http://labda.inf.uc3m.es/

Trees.

Most of the information has been sourced from the books [1, 2].

General Trees

Trees are a natural organization for data and support algorithms much faster than the linear data structures. They are widely used in computing to represent file systems, web sites, databases, graphical user interfaces, etc.

A tree is an abstract data type to store elements that have hierarchical relationships between them. Figure 1 shows the tudor family tree whose root is the node 'Eduardo III'. This node has three children 'Eduardo','Juan' and 'Edmundo' (they are siblings because they are children of the same parent). The parent (direct ancestor) of the node 'Enrique VI' is the node 'Enrique V'. The organigram of a company, an arithmetic expression or the rules of a grammar can be also represented by trees (see Figure 2, 3 and 4).

Properties

A tree is a set of nodes (elements) that maintain a hierarchical (*parent-child*) relationship between them. The top element of a tree is called *root*. The following properties must be satisfied:

- Every node has zero or more children. Nodes without children are called *leaves* or *externals*, while nodes with children are *internals*.
- Every node, except the root, has an unique parent node.

An *empty tree* is a tree that does not contain any nodes. We can formally define *ancestor* and *descendent* terms:

- u is *ancestor* of v (v is *descendent* of u) $\leftrightarrow$ u=v or u is ancestor(parent(v)).
- u is *descendent* of v if v is *ancestor* of u.

Figure 1 shows that 'Eduardo IV' and 'Ricardo III' are descedent of 'Edmundo'. You can also see that 'Enrique VII' is ancestor of 'Isabel I'.

A tree is *ordered* if there is a linear order among siblings. For example, Figure 5 shows an ordered tree of integers.

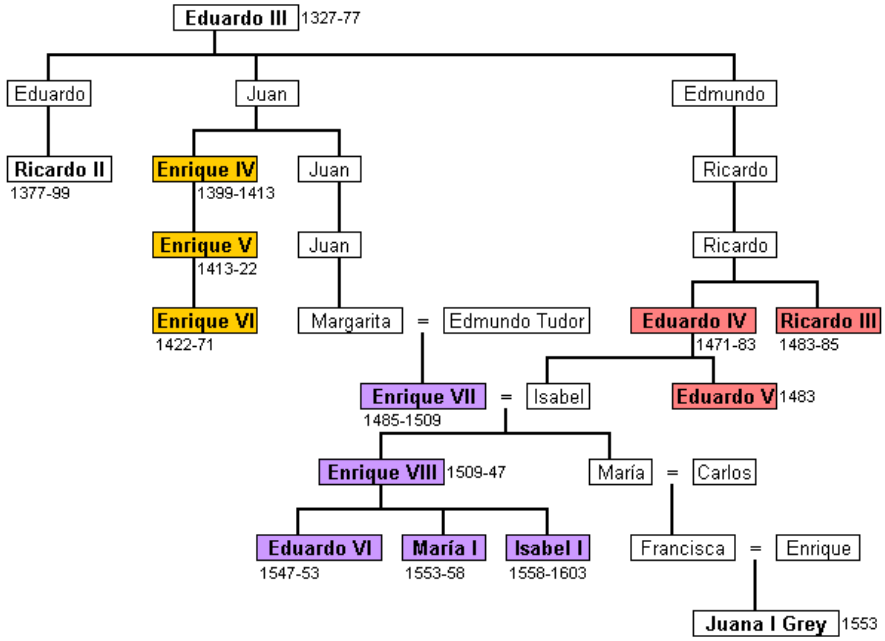


Figure 1: Tudor family tree.

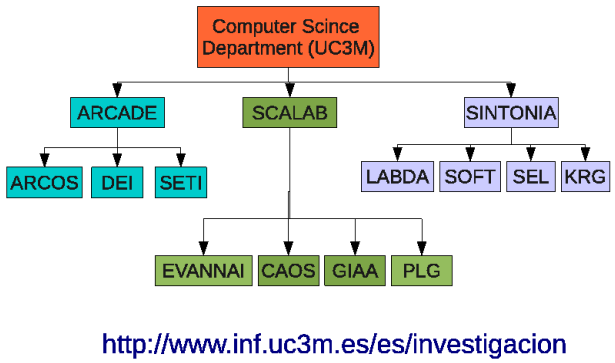


Figure 2: UC3M Computer Science Department's organigram.

The *depth* of a node is the number of its ancestors. The depth of the root is 0. For example, in the tree of Figure 5, the node storing the value *12* has depth 3, the node storing *6* has depth 2 and the node storing *4* has depth 1.

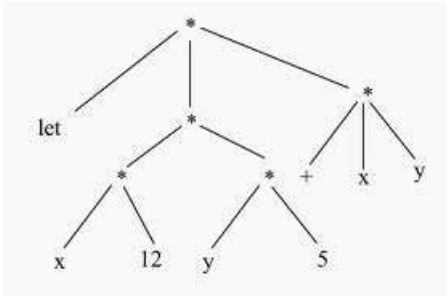


Figure 3: An arithmetic expression.

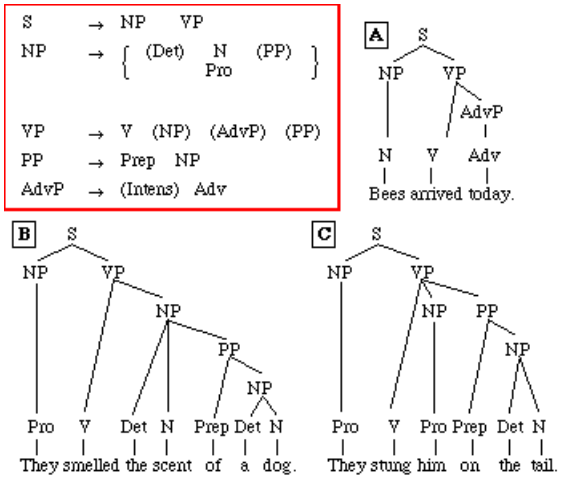


Figure 4: A sintax grammar.

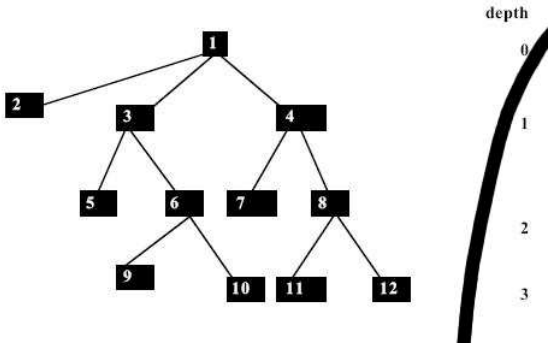


Figure 5: An ordered tree of integers.

Tree Abstract Data Type

A tree should support the following methods:

- *isEmpty()* returns true if the tree is empty.
- *isRoot(v)* returns true if the node v is the root of the tree.
- *isLeave(v)* returns true if v is a leave.
- *isInternal(v)* returns true if v is an internal node.
- *root()* returns the root of the tree. If the tree is empty, an error occurs.
- *parent(v)* returns the parent of the node v. If v is the root, an error occurs.
- *children(v)* returns the children of the node v. If the tree is ordered, this methods stores the children in order.
- *size()*: returns the number of nodes in the tree.
- *iterator()*: returns an interator of all the elements stored at nodes of the tree.
- *positions()*: returns an iterable collection of all the nodes of the tree.
- *replace(v,e)*: replaces the node v with the node e.

Implementing a Tree

First of all, we must define a java interface to represent the tree ADT (see Figureẽffig:interfaceTree). You can see that its methods may throw the exception *InvalidTreePositionException* if the node is invalid (null). Also, if the tree is empty, the method root throws the exception *EmptyTreeException*, and the method parent throws the exception BoundaryViolationException if the node is the root of the tree.

A common way to implement a tree is to use a linked structure in which each node n has the following properties (see Figure 7):

- The element stored at the node.
- A link to its parent. If n is the root, then this field is *null*.
- A collection (for example, an array or a list) containing the links to its children.

```java
package trees;
import java.util.Iterator;
/**An interface for a tree where nodes can have an arbitrary number of children*/
public interface Tree<E> {
    /**Returns the number of nodes of the tree*/
    public int size();
    /**Returns true if the tree is empty, false e.o.c*/
    public boolean isEmpty();
    /**Returns an iterator of the elements stores in the tree*/
    public Iterator<E> iterator();
    /**Returns an iterator of the nodes that stores the elements in the tree*/
    public Iterator<NodeTree<E>> positions();
    /**Returns the root of the tree. If the tree is empty throws an exception*/
    public NodeTree<E> root() throws EmptyTreeException;
    /**Returns the parent of the node v.
     * If v is the root throws the exception BoundaryViolationException.*/
    public NodeTree<E> parent(Position<E> v)
        throws InvalidTreePositionException, BoundaryViolationException;
    /**Returns an iterable collection of the children of a given node*/
    public Iterable<Position<E>> children(Position<E> v)
        throws InvalidTreePositionException;
    /**Returns true if the node v has children.*/
    public boolean isInternal(Position<E> v) throws InvalidTreePositionException;
    /**Returns true if the node v is a leave.*/
    public boolean isLeave(Position<E> v) throws InvalidTreePositionException;
    /**Returns true if the node v is the root of the tree.*/
    public boolean isRoot(Position<E> v) throws InvalidTreePositionException;
    /**Replaces the element stored in the node v by the element e*/
    public E replace(Position<E> v, E e)  throws InvalidTreePositionException;
}
```

Figure 6: An interface for a tree.

Tree Traversal Algorithms

This section presents algorithms for traversing a tree. Before we should define two concepts like **depth** and **height**:

- The **depth** of a node v is the number of ancestors of v. For example, the depth of the node '1' in the tree shown in figure 5 is 0 because this node is the root of the tree. However, for node '10' its depth is 3, because this node has three ancestors. Its implementation is shown in Figure 9

Its running time is $O(d_v)$ where d_v is the depth of the node v, because the algorithm only performs a constant-time recursive step for each ancestor of v.

The height of a node v in a tree can be defined recursively as follows:

- if v is a leave node then its height is 0.
- if v is an internal node then its height is 1 plus the maximum height of a child of v.

Therefore, the height of a Tree is the height of its root. For example, the height of the tree shown in Figure 5 is 4 because its root has depth equals to 4. The height of the node '6' is 1 since all its nodes ('9' and '10') are already leave nodes. Its implementation is shown in Figures 10 and **??**.

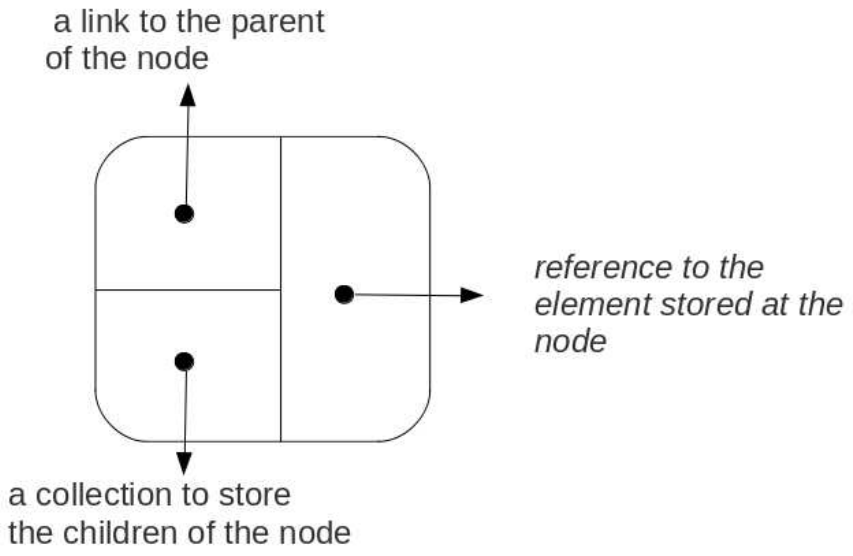


Figure 7: Each node of a tree is represented as a node with the fields: its value, link to its parent and a collection of its children.

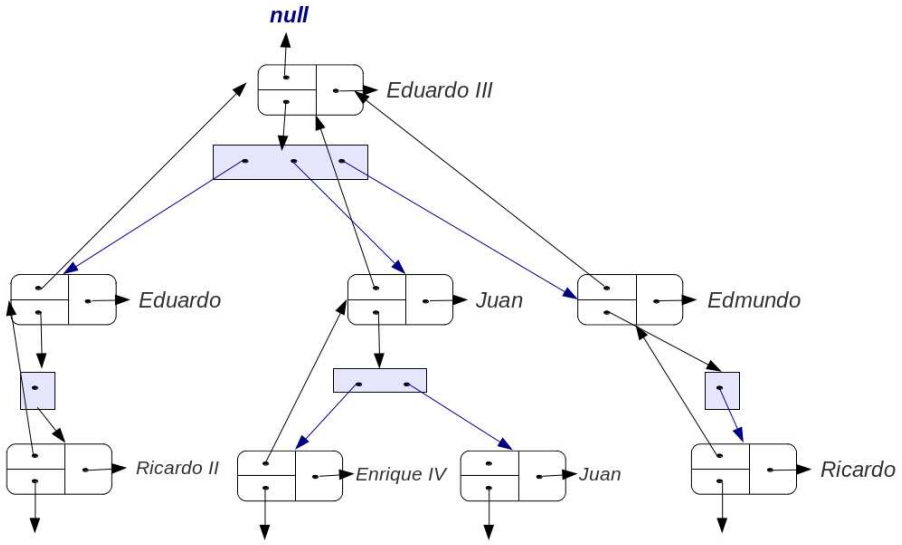


Figure 8: This figure shows the representation using a linked structure of the general tree shown in Figure 1. It only shows the third first levels of the tree.

Methods	Time
isEmtpy(), size()	O(1)
parent(), root	O(1)
isLeave(), isInternal(), isRoot()	O(1)
positions(), iterator()	O(n)
replace()	O(1)
children(n)	O(num of children of v)

Table 1: Performance of an linked structure-based implementation of a binary tree

```
/**
 * computes the depth of a node v, that is, the number of ancestors of v
 * @param T
 * @param v
 * @return
 */
public static <E> int depth(Tree<E> T, NodeTree<E> v) {
    if (T.isRoot(v)) return 0;
    else return 1 + depth(T, T.parent(v));
}
```

Figure 9: Implementation of the depth method.

```
/**
 * computes the height of a node v in the tree T. If the v is a leave then its height is 0.
 * If the v is an internal node then its height is 1 + maximum height of all its children nodes
 * @param <E>
 * @param T
 * @param v
 * @return
 */
public static <E> int height(Tree<E> T, NodeTree<E> v) {
    if (T.isLeave(v)) return 0;

    int h=0;
    for(NodeTree<E> w:T.children(v)) h=Math.max(h,height(T,w));
    return h;
}
```

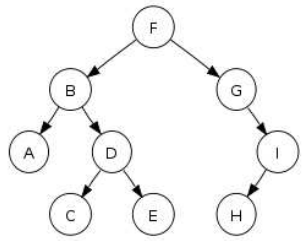
Figure 10: Implementation of the height of a node.

Preorder Traversal

In a preorder traversal, first we must visit the root, then we must visit the subtrees of its children. If the tree is ordered then this output is also ordered. Figure 13 shows the implementation of this preorder traversal.

```java
/**
 * computes the height of a T.
 * @param T
 * @return
 */
public static <E> int height(Tree<E> T) {
    int h=0;
    for(NodeTree<E> w:T.nodes()) h=Math.max(h,height(T,w));
    return h;
}
```

Figure 11: Implementation of the height of a tree.



In this binary search tree

- Preorder traversal sequence: F, B, A, D, C, E, G, I, H (root, left, right)
- Inorder traversal sequence: A, B, C, D, E, F, G, H, I (left, root, right); note how this produces a sorted sequence
- Postorder traversal sequence: A, C, E, D, B, H, I, G, F (left, right, root)
- Level-order traversal sequence: F, B, G, A, D, I, C, E, H

Figure 12: Preorder Traversal.

```java
/**
 * A public method to visit nodes of T in a preorder way.
 * @param <E>
 * @param T
 * @return
 */
public static <E> String toStringPreorder(Tree<E> T) {
    return preorder(T,T.root);
}
private static <E> String preorder(Tree<E> T, NodeTree<E> v) {
    String s=v.element.toString();
    for (NodeTree<E> node:T.children(v)) s = s+ ", " + preorder(T,node);
    return s;
}
```

Figure 13: Implementation of the preorder Traversal.

Postorder Traversal

In a postorder traversal, first we must visit the children of the root (in a postorder way), and finally we must visit the root. Figure 14 shows the implemen-

tation of this postorder traversal.

```
public static <E> String toStringPostorder(Tree<E> T) {
    return postorder(T,T.root);
}
private static <E> String postorder(Tree<E> T, NodeTree<E> v) {
    String s="";
    for (NodeTree<E> node:T.children(v)) s = s + preorder(T,node)+ ", ";
    s=s+v.element;
    return s;
}
```

Figure 14: Implementation of the postorder Traversal.

Bibliography

[1] M. Goodrich and R. Tamassia, *Data structures and algorithms in Java*. Wiley-India, 2009.

[2] M. Weiss, *Data structures and problem solving using Java*. Addison Wesley Publishing Company, 2002.