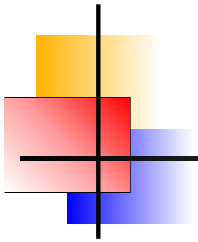


Algoritmos y Estructuras de Datos

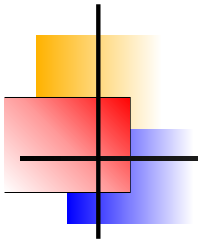
Árboles Generales y Árboles Binarios

Guillermo Román Díez

groman@fi.upm.es

Universidad Politécnica de Madrid

Curso 2015-2016



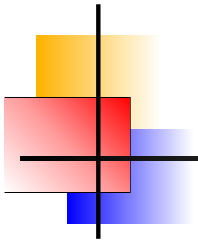
Árboles generales

- El objetivo de los árboles es organizar los datos de forma jerárquica
- Se suelen utilizar en las implementaciones de otros TADs (colas con prioridad, maps ordenados, . . .)

Árbol General

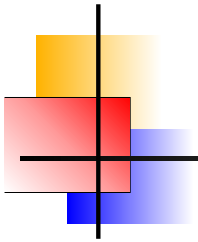
" Un árbol general es o bien vacío o bien tiene dos componentes: (1) un nodo raíz que contiene un elemento, y (2) un conjunto de cero o más (sub)árboles hijos."

- Un árbol está formado por nodos
- Un nodo tiene un elemento y un conjunto de nodos que son la raíz de los subárboles hijos



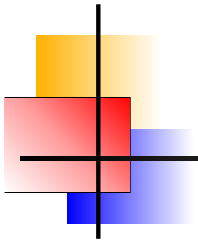
Terminología

- **Raíz** ("root"): nodo sin padre
- **Nodo interno** ("internal node"): nodo con al menos un hijo
- **Nodo externo** ("external node"): nodo sin hijos
- **Nodo hoja** ("leaf node"): sinónimo de nodo externo, usaremos estos dos nombres indistintamente
- **Subárbol** ("subtree"): árbol formado por el nodo considerado como raíz junto con todos sus descendientes
- **Ancestro** de un nodo: un nodo 'w' es ancestro de 'v' si y sólo si 'w' es 'v' o 'w' es el padre de 'v' o 'w' es ancestro del padre de 'v'
- **Descendiente** de un nodo (la inversa de ancestro): 'v' es descendiente de 'w' si y sólo si 'w' es ancestro de 'v'



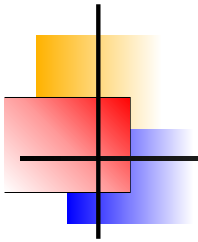
Terminología

- **Hermano** ("sibling") de un nodo: nodo con el mismo padre
- **Arista** ("edge") de un árbol: par de nodos en relación padre-hijo o hijo-padre
- **Grado** ("degree") de un *nodo*: el número de hijos del nodo
- **Grado** ("degree") de un *árbol*: el máximo de los grados de todos los nodos
- **Camino** ("path") de un árbol: secuencia de nodos tal que cada nodo consecutivo forma una arista. La **longitud del camino** es el número de aristas
- **Árbol ordenado** ("ordered tree"): existe un orden lineal (total) definido para los hijos de cada nodo: primer hijo, segundo hijo, etc...



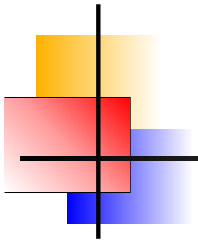
Terminología

- La **profundidad de un nodo** ("depth") es la longitud del camino que va desde ese nodo hasta la raíz (o viceversa)
- La **altura de un nodo** ("height") es la longitud del mayor de todos los caminos que van desde el nodo hasta una hoja
- **Altura de un árbol no vacío**: la altura de la raíz
- **Nivel** ('level'): conjunto de nodos con la misma profundidad. Así, tenemos desde el nivel 0 hasta el nivel 'h' donde 'h' es la altura del árbol



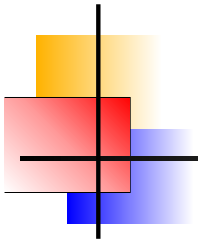
Interfaz Tree

```
public interface Tree<E> {
  public int size();

  public boolean isEmpty();

  public Iterator<E> iterator();

  public Iterable<Position<E>> positions();

  public E replace(Position<E> v, E e)
    throws InvalidPositionException;

  public Position<E> root() throws EmptyTreeException;
  ...
}
```



Interfaz Tree<E>

```
public interface Tree<E> {
  ...
  public Position<E> parent(Position<E> v)
    throws InvalidPositionException,
        BoundaryViolationException;

  public Iterable<Position<E>> children(Position<E> v)
    throws InvalidPositionException;

  public boolean isInternal(Position<E> v)
    throws InvalidPositionException;

  public boolean isExternal(Position<E> v)
    throws InvalidPositionException;

  public boolean isRoot(Position<E> v)
    throws InvalidPositionException;
}
```

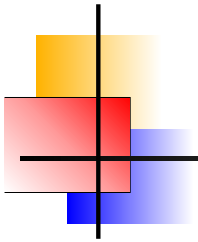


Interfaz Tree<E>

- `Tree<E>` es un interfaz pensado para trabajar directamente con posiciones
- Los métodos trabajan con posiciones y no con árboles (no son recursivos)
- Tenemos dos métodos que devuelven un `Iterable`
  - `children` devuelve un `Iterable` para recorrer los hijos de un nodo
  - `positions()` devuelve un `Iterable` que es un *snapshot* de los nodos del árbol
- El interfaz sólo se dispone de métodos observadores
  - Sólo está pensado para hacer recorridos de árboles
  - Los métodos modificadores se definirán en las clases que implementan el interfaz

Pregunta

¿qué os parece esto?

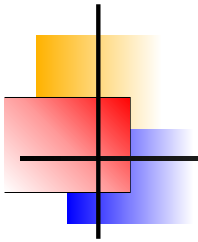


Ejemplos recorrido

Ejemplo

Método que indica si un nodo es ancestro de otro

```
boolean ancestro(Tree<E> t,
                 Position<E> w,
                 Position<E> v)
                     throws InvalidPositionException {
  return w == v || !t.isRoot(v) &&
         (w == t.parent(v)||ancestro(t,w,t.parent(v)));
}
```

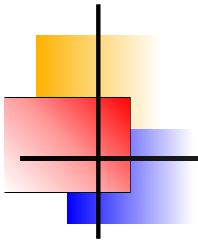


Ejemplos recorrido

Ejemplo

Método que indica si un nodo es hermano de otro

```
boolean isSibling(Tree<E> t,
                  Position<E> w,
                  Position<E> v)
                     throws InvalidPositionException {
 if (w == v || t.isRoot(w) || t.isRoot(v))
   return false;
 else
   return t.parent(w) == t.parent(v);
}
```

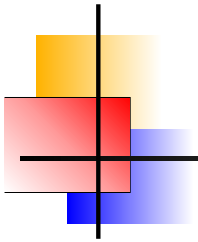


Ejemplos recorrido

Ejemplo

Método que indica si un nodo es hermano de otro (algo retorcido)

```
boolean isSibling(Tree<E> t, Position<E> w,
                  Position<E> v)
    throws InvalidPositionException {
  if (w == v || t.isRoot(w) || t.isRoot(v))
    return false;
  else {
    Iterator<Position<E>> it = t.parent(w).children().
        iterator();
    boolean found = false;
    while (it.hasNext() && !(found = it.next() == v))
      ;
    return found;
  }
}
```

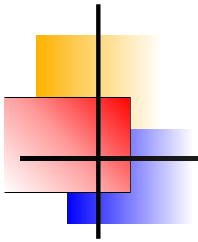


Ejemplos recorrido

Ejemplo

Método que calcula la profundidad de un nodo de un árbol dado (iterativo)

```
int depth(Tree<E> tree, Position<E> v)
                        throws InvalidPositionException {
  int c;
  for (c = 0; !tree.isRoot(v); c++, v=tree.parent(v))
    ;
  return c;
}
```

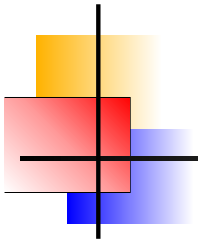


Ejemplos recorrido

Ejemplo

Método que calcula la profundidad de un nodo de un árbol dado (recursivo)

```
int depth(Tree<E> tree, Position<E> v)
                    throws InvalidPositionException {
  if (tree.isRoot(v))
    return 0;
  else
    return 1 + depth(tree,tree.parent(v));
}
```

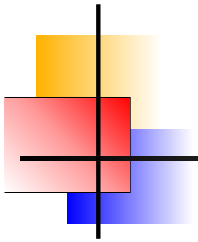


Ejemplos recorrido

Ejemplo

Método que recorre un árbol en **pre-orden**

```
String toStringPreorder(Tree<E> tree, Position<E> v)
      throws InvalidPositionException {

  String s = v.element().toString();

  for (Position<E> w : tree.children(v)) {
    s += ", " + toStringPreorder(tree, w);
  }
  return s;
}
```

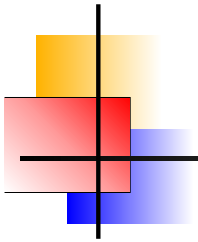


Ejemplos recorrido

Ejemplo

Método que recorre un árbol en **post-orden**

```
String toStringPostorder(Tree<E> tree, Position<E> v)
        throws InvalidPositionException {
  String s = "";
  for (Position<E> w : tree.children(v)) {
    s += toStringPostorder(tree, w) + " ";
  }

  s += v.element();
  return s;
}
```



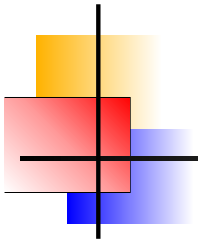
Árboles Binarios

- Es un tipo especial de árbol en el que todo nodo tiene como mucho 2 hijos
  - Es un árbol ordinario con grado 2

Árbol Binario

"Un árbol binario es o bien vacío o bien consiste en (1) un nodo raíz, (2) un (sub)árbol izquierdo, y (3) un (sub)árbol derecho."

- Podemos hablar de varios tipos:
  - **Arbol binario propio** (*proper binary tree*): todo nodo interno tiene 2 hijos
  - **Arbol binario impropio** (*improper binary tree*): árbol binario que no es propio

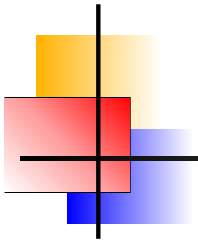


Árboles Binarios

Árbol binario perfecto

"es un árbol binario propio con el máximo número de nodos: $2^{h+1} - 1$"

```
   o          h           = 0
              2^{h+1}-1   = 1

   o          h           = 1
  / \         2^{h+1}-1   = 3
 o   o

   o          h           = 2
  / \         2^{h+1}-1   = 7
 o   o
/ \ / \
o  oo  o
```

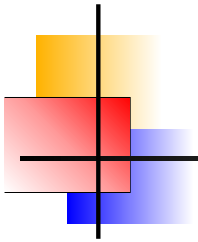


Árboles Binarios

Árbol binario equilibrado

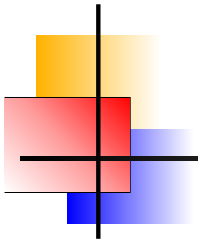
"*Un árbol en el que para todo nodo, el valor absoluto de la diferencia de altura entre los dos subárboles hijos es como máximo 1.*"

- Es decir, un árbol en el que para todo nodo con altura h, o bien sus dos hijos tienen la misma altura, $h-1$, o un hijo tiene altura $h-1$ y el otro $h-2$
- Todo subárbol de un árbol equilibrado es también equilibrado



Árboles Binarios

```
public interface BinaryTree<E> extends Tree<E> {
  public Position<E> left(Position<E> v)
    throws InvalidPositionException,
        BoundaryViolationException;

  public Position<E> right(Position<E> v)
    throws InvalidPositionException,
        BoundaryViolationException;

  public boolean hasLeft(Position<E> v) throws
      InvalidPositionException;

  public boolean hasRight(Position<E> v) throws
      InvalidPositionException;
}
```

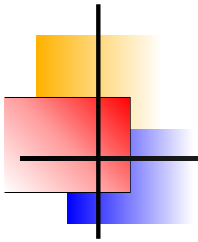


Ejemplos recorrido

Ejemplo

Método que devuelve la altura de un árbol binario

```
int height(BinaryTree<E> tree, Position<E> v)
      throws InvalidPositionException {
  if (tree.isExternal(v)) return 0;

  int hi = 0, hd = 0;

  if (tree.hasLeft(v))
    hi = height(tree, tree.left(v));
  if (tree.hasRight(v))
    hd = height(tree, tree.right(v));
  return 1 + Math.max(hi,hd);
}
```

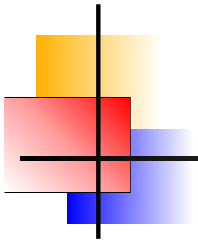


Ejemplos recorrido

Ejemplo

Recorrido de un árbol binario en pre-orden y en post-orden

```
void preorder(BinaryTree<E> tree, Position<E> v)
        throws InvalidPositionException {
  /* visit v.element() */
  if (t.hasLeft(v))  preorder(tree, tree.left(v));
  if (t.hasRight(v)) preorder(tree, tree.right(v));
}

void postorder(BinaryTree<E> tree, Position<E> v)
        throws InvalidPositionException {
  if (t.hasLeft(v))  postorder(tree, tree.left(v));
  if (t.hasRight(v)) postorder(tree, tree.right(v));
  /* visit v.element() */
}
```



Ejemplos recorrido

Ejemplo

Los árboles binarios también permiten el recorrido en **inorden**

```
void inorder(BinaryTree<E> tree, Position<E> v)
        throws InvalidPositionException {
  if (tree.hasLeft(v))  inorder(tree, tree.left(v));
  /* visit v.element() */
  if (tree.hasRight(v)) inorder(tree, tree.right(v));
}
```