Estructuras de Datos
Clase 11 – Árboles Generales (segunda parte)

Dr. Sergio A. Gómez
http://cs.uns.edu.ar/~sag

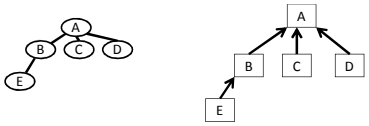
Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
Bahía Blanca, Argentina

Representaciones de árboles

- Del padre
- Lista de hijos
- Goodrich & Tamassia: Del padre + Lista de hijos
- Hijo extremo izquierdo – hermano derecho
- Variantes de las de más arriba usando arreglos con cursores (de interés histórico o para modelar árboles en almacenamiento externo)

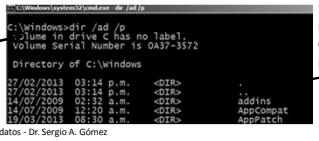
Representación del padre

- Cada nodo conoce su rótulo y a su padre.



- Aplicación: Directorio padre en sistemas de archivos
- El directorio padre se denota con .. (dos puntos consecutivos).

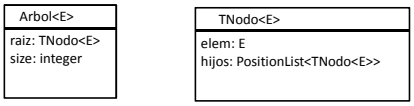
El directorio corriente es c:\windows
El comando *cd..* lleva al directorio padre.



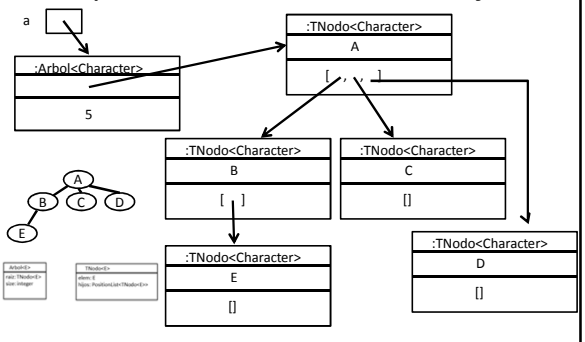
".." es la referencia al directorio padre

Representación de lista de hijos

- Cada nodo conoce su rótulo (elemento) y la lista de sus nodos hijos
- El árbol conoce el nodo raíz del árbol



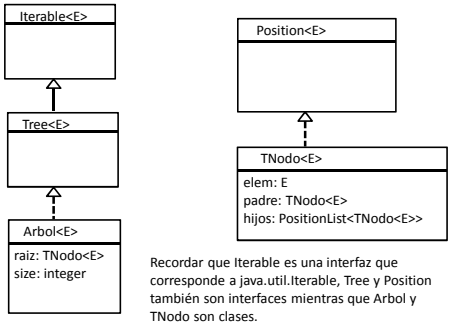
Representación de lista de hijos



Representación en GT: Colección de hijos

- GT = Lista de hijos + padre
- El árbol conoce la raíz
- Cada nodo conoce el rótulo (elemento), la lista de nodos hijos y el nodo padre.
- (Discutiremos la estructura de datos y la implementación de algunas de las operaciones, el resto quedará a cargo del alumno.)

Representación de árboles en GT



Recordar que Iterable es una interfaz que corresponde a java.util.Iterable, Tree y Position también son interfaces mientras que Arbol y TNodo son clases.

Representación de árboles en GT

```
public class TNodo<E> implements Position<E>
{
        private E elemento;
        private TNodo<E> padre;
        private PositionList<TNodo<E>> hijos;

        public TNodo( E elemento, TNodo<E> padre ) {
                this.elemento = elemento;
                this.padre = padre;
                hijos = new DoubleLinkedList<TNodo<E>>();
        }
        public TNodo(E elemento ) { this( elemento, null); }
        public E element() { return elemento; }
        public PositionList<TNodo<E>> getHijos() { return hijos; }
        public void setElemento( E elemento ) { this.elemento = elemento; }
        public TNodo<E> getPadre() { return padre; }
        public void setPadre( TNodo<E> padre ) { this.padre = padre; }
}
```

Arbol<E>: raiz: TNodo<E>, size: integer

TNodo<E>: elem: E, hijos: PositionList<TNodo<E>>

Representación de árboles en GT

```
public class Arbol<E> implements Tree<E>
{
        protected TNodo<E> raiz;
        protected int size;

        public Arbol() { raiz = null; size = 0; }
        public boolean isEmpty() { return raiz == null; }
        public void createRoot( E e ) throws InvalidOperationException {
                if( !isEmpty() )
                    throw new InvalidOperationException( "Arbol no vacío" );
                raiz = new TNodo<E>( e );
                size = 1;
        }

        public Position<E> root() throws EmptyTreeException {
                if( isEmpty() )
                        throw new EmptyTreeException( "Árbol vacío" );
                return raiz;
        }
```

TNodo<E>: elem: E, padre: TNodo<E>, hijos: PositionList<TNodo<E>>


```
public boolean isExternal(Position<E> v) throws InvalidPositionException
{
        TNodo<E> nodo = checkPosition( v );
        return nodo.getHijos().isEmpty();
}

public Position<E> addFirstChild(Position<E> p, E e) throws
        InvalidPositionException {
        TNodo<E> padre = checkPosition(p);
        TNodo<E> nodo = new TNodo<E>( e, padre );
        padre.getHijos().addFirst( nodo );
        size++;
        return nodo;
}
```

TNodo<E>: elem: E, padre: TNodo<E>, hijos: PositionList<TNodo<E>>


```
public Position<E> addBefore(Position<E> p, Position<E> rb, E e)
        throws InvalidPositionException {
        TNodo<E> padre = checkPosition( p );
        TNodo<E> hermanoDerecho = checkPosition( rb );
        TNodo<E> nuevo = new TNodo<E>( e, padre );
        PositionList<TNodo<E>> hijosPadre = padre.getHijos();
        // encontre: true si encontré ubicación en lista de hijos de su padre
        boolean encontre = false;
        Position<TNodo<E>> pp = hijosPadre.first();
        while( pp != null && !encontre )
                if( hermanoDerecho == pp.element() )
                    encontre = true
                else
                    pp = (pp != hijosPadre.last() ? hijosPadre.next(pp) : null);
        if( !encontre )
                throw new InvalidPositionException( "p no es padre de rb" );
        hijosPadre.addBefore( pp, nuevo );
        size++;
        return nuevo;
}
```



```
public void removeExternalNode (Position<E> p)
        throws InvalidPositionException {
        if( isEmpty() ) throw new InvalidPositionException( "Arbol vacío" );
        TNodo<E> n = checkPosition( p );
        if( !n.getHijos().isEmpty() )
            throw new InvalidPositionException( "p no es hoja" );
        TNodo<E> padre = n.getParent();
        PositionList<TNodo<E>> hijosPadre = padre.getHijos();
        boolean encontre = false; Position<TNodo<E>> pp = null;
        Iterable<Position<TNodo<E>> posiciones = hijosPadre.positions();
        Iterator<Position<TNodo<<E>> it = posiciones.iterator();
        while( it.hasNext() && !encontre ) {
                pp = it.next();
                if( pp.element() == n ) encontre = true;
        }
        if( !encontre )
            throw new InvalidPositionException( "p no aparece en la lista de"
                + "hijos de su padre--- árbol corrupto???" );
        hijosPadre.remove( pp );
        size--;
}
```

Arbol<E>: raiz: TNodo<E>, size: integer

TNodo<E>: elem: E, padre: TNodo<E>, hijos: PositionList<TNodo<E>>

Árboles generales: children

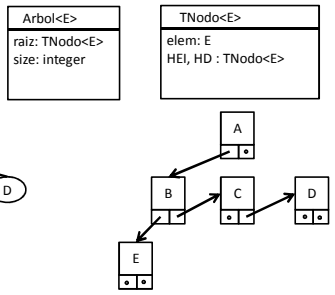
```
public Iterable<Position<E>> children( Position<E> v )
                    throws InvalidPositionException
{
     if( v == null )
          throw new
                InvalidPositionException( "Children: Posición inválida");
    TNodo<E> p = checkPosition(v);
    PositionList<Position<E>> lista =
                    new ListaDoblementeEnlazada<TNodo<E>>();
    for( TNodo<E> n : p.getHijos() )
          lista.addLast(n);
    return lista;
}   El tiempo de ejecución es del orden de la cantidad de hijos de v
```


Árboles generales: Iterador

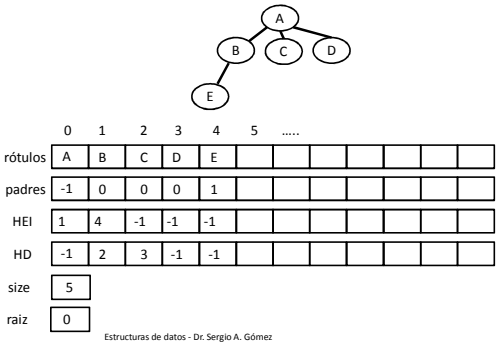
```
public Iterable<Position<E>> positions()  {
  PositionList<Position<E>> l = new ListaDoblementeEnzalada<Position<E>>();
  if( !isEmpty() ) pre( l, raiz );
  return l;
}                                   T_positions(n) = O(n) si árbol this tiene n nodos

private void pre(PositionList<Position<E>> l, TNodo<E> r) {
  l.addLast( r );
  for( TNodo<E> h : r.getHijos())
          pre( l, h );
}           El tiempo de ejecución es lineal en la cantidad de nodos del árbol

public Iterator<E> iterator() {    // T_iterator(n) = O(n) si árbol this tiene n nodos
   PositionList<E> l = new ListaDoblementeEnlazada<E>();
   for( Position<E> p : positions() )
          l.addLast( p.element() );
   return   l.iterator();
}
```


Representación HEI-HD

Cada nodo conoce su rótulo y la identidad de su hijo extremo izquierdo y de su hermano derecho (se puede combinar con la representación del padre de ser necesario)



Representaciones con arreglos



Recorrido preorden

```
public void preorden() {
  pre( raiz );
}

// r representa el cursor de la raíz del subárbol listado actualmente
private void pre( int r ) {
    System.out.println( rotulos[r] );
    int h = HEI[r];         // h representa el hijo actual procesado
    while( h != -1 ) {
         pre( h );
         h = HD[h];
    }
}
```