Algoritmos y Estructuras de Datos

Árboles Generales y Árboles Binarios

Guillermo Román Díez

groman@fi.upm.es

Universidad Politécnica de Madrid

Curso 2021/2022

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, ...)

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'

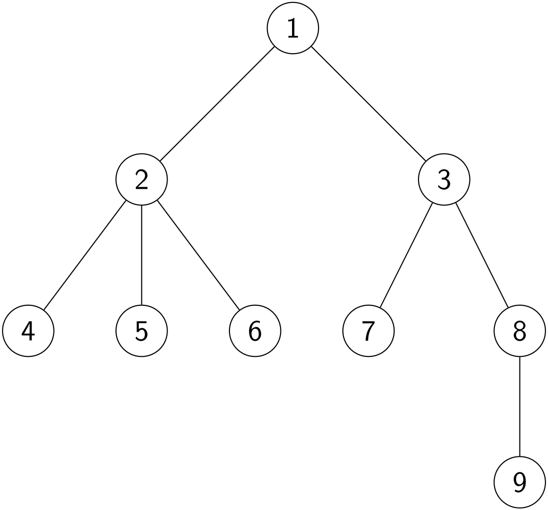
Terminología

- **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. . .

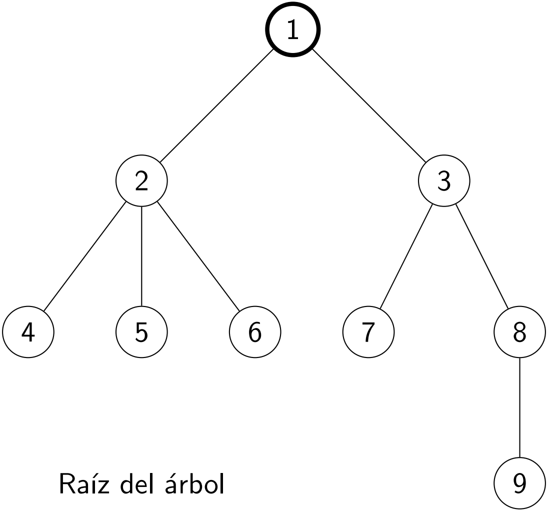
Terminología

- 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

Terminología

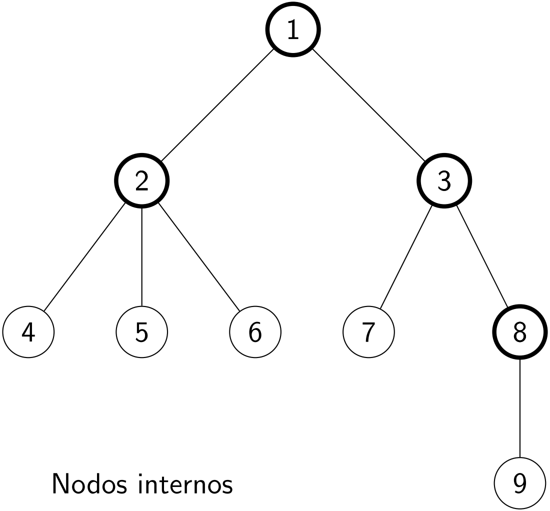


Terminología



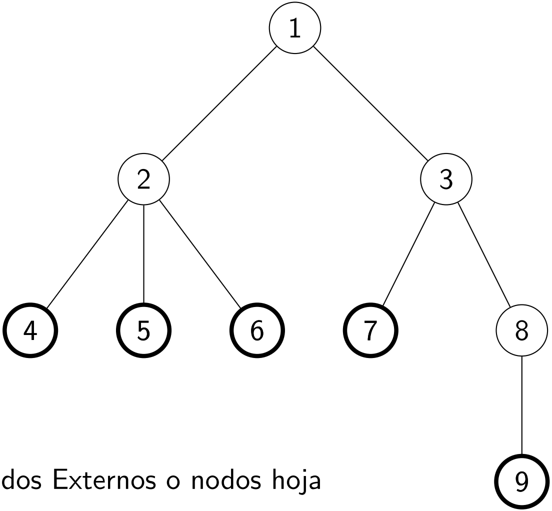
Raíz del árbol

Terminología



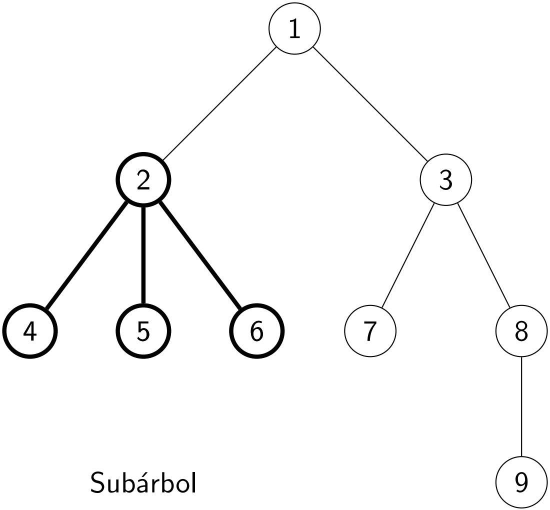
Nodos internos

Terminología



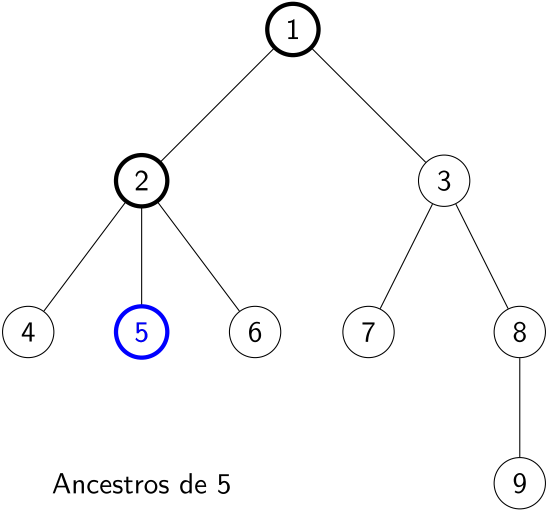
Nodos Externos o nodos hoja

Terminología



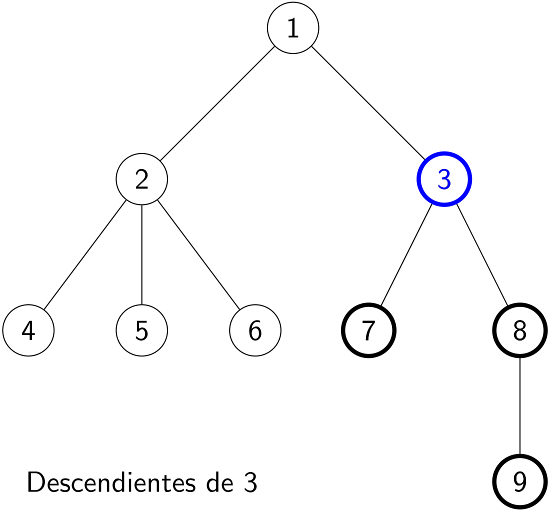
Subárbol

Terminología



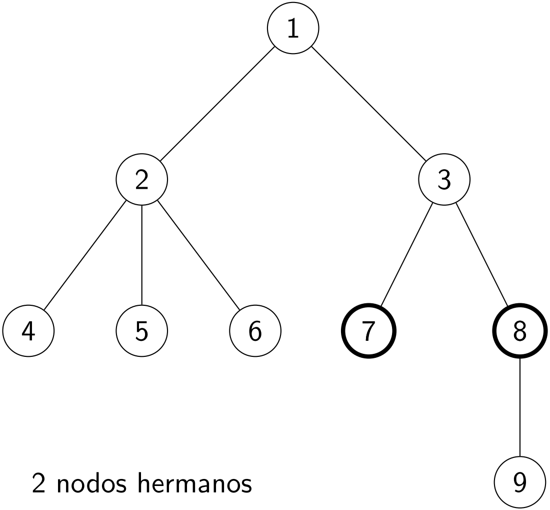
Ancestros de 5

Terminología



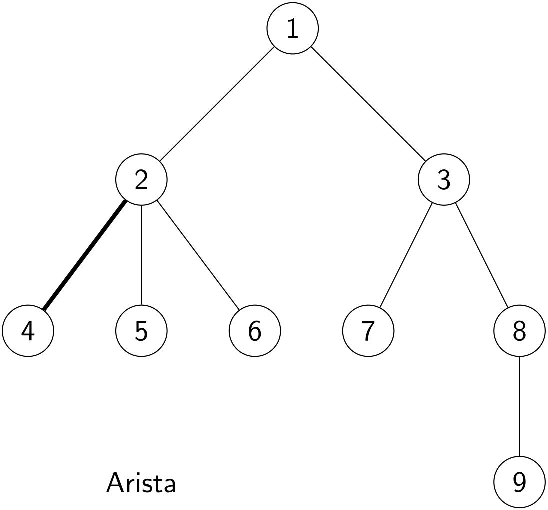
Descendientes de 3

Terminología

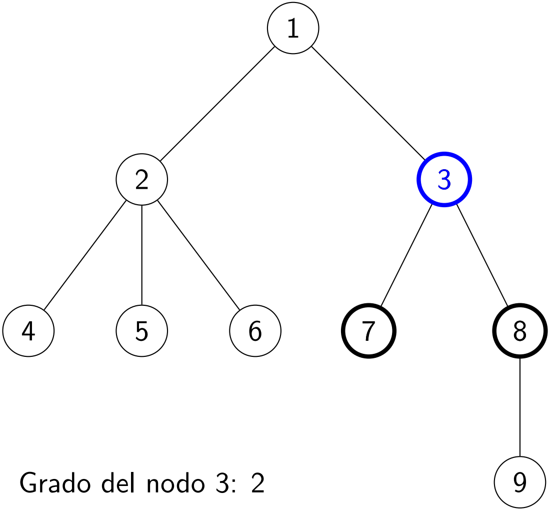


2 nodos hermanos

Terminología

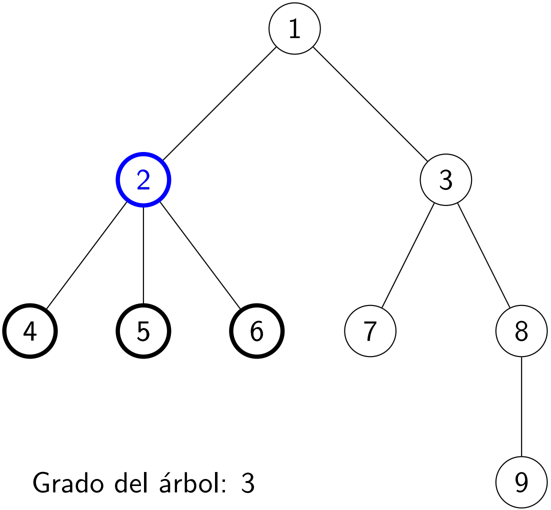


Terminología



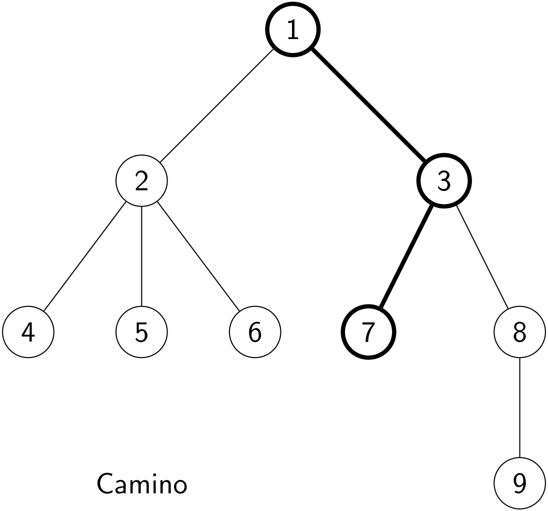
Grado del nodo 3: 2

Terminología



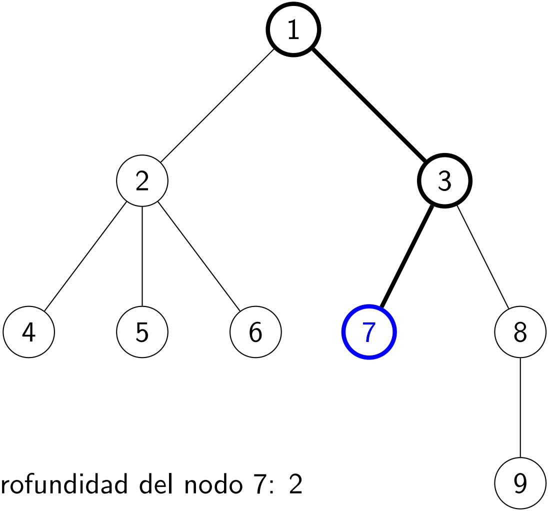
Grado del árbol: 3

Terminología



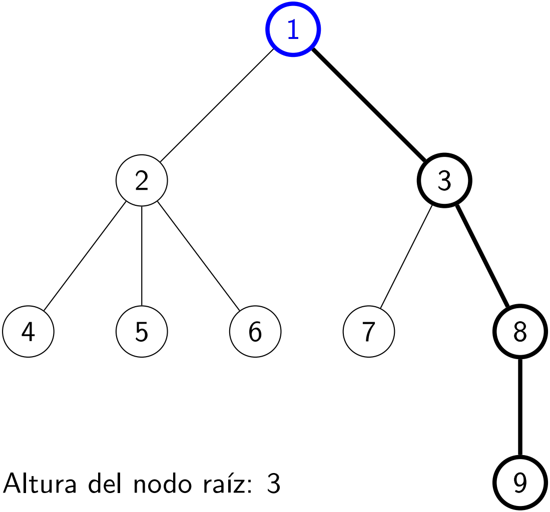
Camino

Terminología



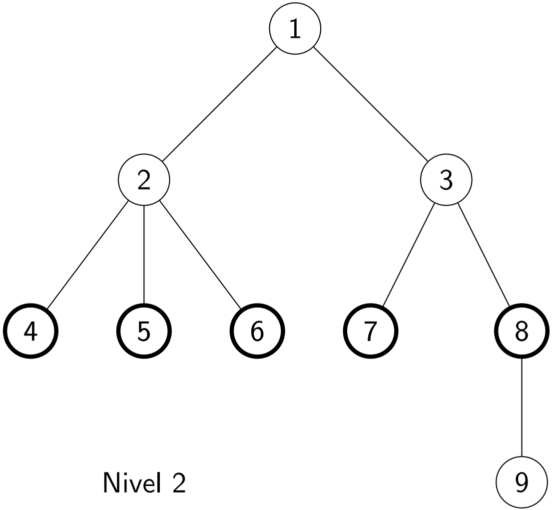
Profundidad del nodo 7: 2

Terminología

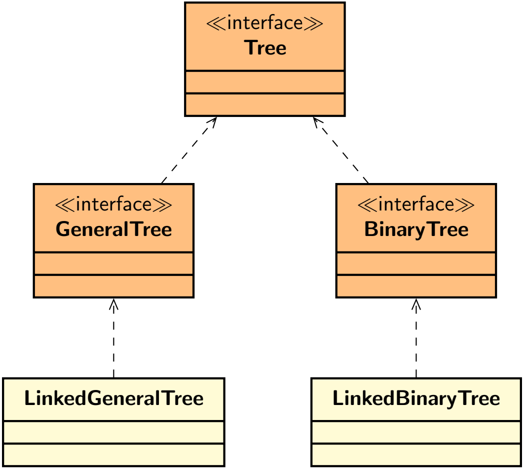


Altura del nodo raíz: 3

Terminología



Jerarquía de clases e interfaces en aedlib



Interfaz Tree<E>

```
public interface Tree<E> extends Iterable<E> {

  public int size();

  public boolean isEmpty();

  public Position<E> addRoot(E e)
             throws NonEmptyTreeException;

  public Position<E> root();

  public Position<E> parent(Position<E> p)
             throws IllegalArgumentException;

  ...
}
```

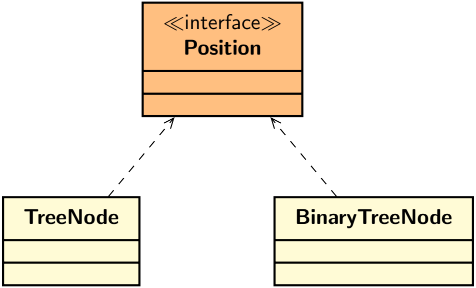
Interfaz `Tree<E>`

```
public interface Tree<E> extends Iterable<E> {

  ...

  public boolean isInternal(Position<E> p);

  public boolean isExternal(Position<E> p);

  public boolean isRoot(Position<E> p);

  public E set(Position<E> p, E e)
                            throws IllegalArgumentException;

  public Iterable<Position<E>> children(Position<E> p);
}
```

Interfaz `Tree<E>`

- `Tree<E>` es un interfaz pensado para trabajar directamente con `Position`
- Los métodos que reciben un `Position<E>` podrán lanzar la excepción `IllegalArgumentException`
- Tenemos el método `children` devuelve un `Iterable` para recorrer los hijos de un nodo
- `Tree<E>` fundamentalmente se dispone de métodos observadores
  - Está pensado para hacer recorridos de árboles ordinarios
  - Los métodos modificadores se definirán en las clases que extienden el interfaz `Tree<E>`
- Los métodos `addRoot` y `set` son los únicos métodos modificadores, pero no son suficientes para construir el árbol

Posiciónes en arboles



TreeNode and BinaryTree internamente

- ```
  class TreeNode<E> extends Node<E,Tree<E>> implements
      Position<E> {

    Position<E> parent;                      // parent
    PositionList<Position<E>> children;      // children
    ...
  }
  ```

- ```
  class BinaryTreeNode<E> extends Node<E,Tree<E>>
      implements Position<E> {
    Position<E> parent;          // parent
    Position<E> left, right;     // children
    ...
  }
  ```

Ejemplos

Ejemplo

Método que indica si un nodo w es ancestro de otro nodo v

Ejemplos

Ejemplo

Método que indica si un nodo w es ancestro de otro nodo v

```
boolean ancestro(Tree<E> tree,
                 Position<E> w,
                 Position<E> v) {

  if (w == v) return true;
  if (tree.isRoot(v)) return false;
  return esAncestro (tree,w,tree.parent(v));
}
```

Ejemplos

Ejemplo

Método que indica si un nodo es hermano de otro

Ejemplos

Ejemplo

Método que indica si un nodo es hermano de otro

```
boolean isSibling(Tree<E> t,
                  Position<E> w,
                  Position<E> v) {
 return w != v && t.parent(w) == t.parent(v);
}
```

Ejemplos

Ejemplo

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

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

- Las estructuras de datos lineales (p.e. listas, arrays) presentan un único recorrido *lógico* de todos sus elementos
- En el caso de árboles, es posible recorrer todos sus elementos en diferente orden
  - Recorrido en **profundidad**
    - Se *visitan* todos los nodos de una rama (subárbol) antes de pasar al siguiente rama
    - En **pre-orden**: se visita cada nodo antes de visitar los subárboles *hijos*
      En el ejemplo anterior: $1, 2, 4, 5, 6, 3, 7, 8, 9$
    - En **post-orden**: se visita cada nodo después de visitar los subárboles *hijos*
      En el ejemplo anterior: $4, 5, 6, 2, 7, 9, 8, 3, 1$
  - Recorrido en **anchura** (por niveles)
    - Se *visitan* todos los nodos de un nivel antes de visitar los nodos del siguiente nivel
      En el ejemplo anterior: $1, 2, 3, 4, 5, 6, 7, 8, 9$

Ejemplos recorrido: Pre-orden

Ejemplo

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

```
void preorder(Tree<E> tree)
  if (tree.isEmpty()) {
    return;
  }
  preorder(tree,tree.root());
}

void preorder(Tree<E> tree, Position<E> v) {

  // AQUI "visitamos" el nodo "v"
  for (Position<E> w : tree.children(v)) {
    preorder(tree, w);
  }
}
```

Ejemplos recorrido: Post-orden

Ejemplo

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

```
void postorder(Tree<E> tree) {
  if (tree.isEmpty()) {
    return;
  }
  postorder(tree,tree.root());
}

void postorder(Tree<E> tree, Position<E> v) {

  for (Position<E> w : tree.children(v)) {
    postorder(tree, w);
  }
  // AQUI "visitamos" el nodo "v"
}
```

Ejemplos recorrido: Anchura

Ejemplo

Método que recorre un árbol en **anchura**

```
void breadth(Tree<E> tree) {
  FIFO<Position<E>> fifo = new FIFOList<Position<E>>();
  fifo.enqueue(tree.root());
  while (!fifo.isEmpty()) {
    Position<E> v = fifo.dequeue();

    // AQUI "visitamos" el nodo "v"

    for (Position<E> w : tree.children(v)) {
      fifo.enqueue(w);
    }
  }
}
```

El interfaz `GeneralTree<E>`

- Este interfaz extiende el interfaz `Tree<E>` con los métodos modificadores necesarios para construir árboles

```java
public interface GeneralTree<E> extends Tree<E> {

  Position<E> addChildFirst(Position<E> parentPos, E e);

  Position<E> addChildLast(Position<E> parentPos, E e);

  Position<E> insertSiblingBefore(Position<E> siblingPos, E e);

  Position<E> insertSiblingAfter(Position<E> siblingPos, E e);

  void removeSubtree(Position<E> p);
}
```

- La clase `LinkedGeneralTree` implementa el intefaz `GeneralTree`

El interfaz `GeneralTree<E>`

```
GeneralTree<Integer> tree = new LinkedGeneralTree<Integer>();
tree.addRoot(1);

Position<Integer> n2 = tree.addChildLast(tree.root(), 2);
Position<Integer> n3 = tree.addChildLast(tree.root(), 3);

Position<Integer> n4 = tree.addChildLast(n2, 4);
Position<Integer> n6 = tree.addChildLast(n2, 6);
Position<Integer> n5 = tree.insertSiblingBefore(n6, 5);

Position<Integer> n7 = tree.addChildLast(n3, 7);
Position<Integer> n8 = tree.insertSiblingAfter(n7, 8);

Position<Integer> n9 = tree.addChildLast(n8, 9);
```

Árboles Binarios

- Es un tipo especial de árbol en el que todo nodo tiene como mucho 2 hijos
  - Es un árbol general de 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/full 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 (*perfect binary tree*)

"*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 (*balanced binary tree*)

"*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 boolean hasLeft(Position<E> p);

  public boolean hasRight(Position<E> p);

  public Position<E> left(Position<E> p);

  public Position<E> right(Position<E> p);

  public Position<E> insertLeft(Position<E> parentPos, E e)
         throws NodeAlreadyExistsException;

  public Position<E> insertRight(Position<E> parentPos, E e)
         throws NodeAlreadyExistsException;

  public void removeSubtree (Position<E> pos);
}
```

Árboles Binarios

- La clase `LinkedBinaryTree` implementa el interfaz `BinaryTree`

```
BinaryTree<Integer> tree = new LinkedBinaryTree<Integer>();
tree.addRoot(1);
Position<Integer> left = tree.insertLeft(tree.root(), 2);
Position<Integer> right = tree.insertRight(tree.root(), 3);
Position<Integer> n4 = tree.insertLeft(left,4);
Position<Integer> n5 = tree.insertRight(left,5);
Position<Integer> n6 = tree.insertLeft(right,6);
```

Ejemplos recorrido

Ejemplo

Método que devuelve la altura de un árbol binario

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

Ejemplos recorrido

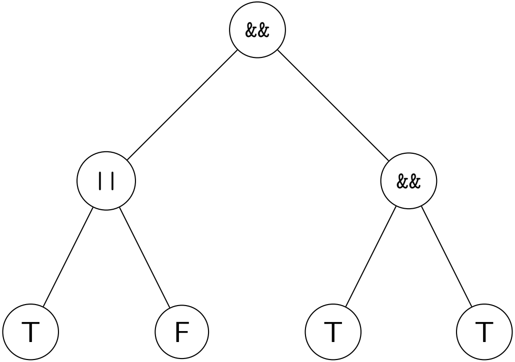
Ejemplo

Los árboles binarios también permiten el recorrido en **in-orden**

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

Expresiones mediante árboles

- Mediante árboles binarios se pueden representar expresiones booleanas



Imprimir paréntesis

Ejercicio

Imprimir la expresión contenida en un árbol con sus correspondientes paréntesis

Imprimir paréntesis

Ejercicio

Imprimir la expresión contenida en un árbol con sus correspondientes paréntesis

```
void imprimirExp(BinaryTree<E> t, Position<E> v) {
  if (!t.isExternal(v)) System.out.print("(");
  if (t.hasLeft(v))  imprimirExp(t,t.left(v));
  System.out.print(v.element());
  if (t.hasRight(v)) imprimirExp(t,t.right(v));
  if (!t.isExternal(v)) System.out.print(")");
}
```

Evaluar expresión booleana

Ejercicio

Evaluar la expresión booleana contenida en un árbol

Evaluar expresión booleana

Ejercicio

Evaluar la expresión booleana contenida en un árbol

```
boolean eval(BinaryTree<Character> t,
             Position<Character> v) {

  switch(v.element()) {
    case 'T': return true;
    case 'F': return false;
    case '|':
      return  eval(t,t.left(v)) || eval(t,t.right(v));
    case '&':
      return  eval(t,t.left(v)) && eval(t,t.right(v));
    default:
      throw new IAE("Nodo incorrecto " + t.element());
  }
}
```

Evaluar expresión aritmética

Ejercicio

Evaluar la expresión aritmética contenida en un árbol

Evaluar expresión aritmética

Ejercicio

Evaluar la expresión aritmética contenida en un árbol

```
int eval(BinaryTree<Character> t, Position<Character> v) {

  if (Character.isDigit(v.element()))
     return Character.getNumericValue(v.element());

  switch(v.element()) {
    case '+':
      return  eval(t,t.left(v)) + eval(t,t.right(v));
    case '-':
      return  eval(t,t.left(v)) - eval(t,t.right(v));
    case '*':
      return  eval(t,t.left(v)) * eval(t,t.right(v));
    case '/':
      return  eval(t,t.left(v)) / eval(t,t.right(v));
    default:
      throw new IAE("Elemento incorrecto " + v.element());
  }
}
```

Busquedas en arboles

- Hemos visto como buscar un elemento en un árbol – visitar todos los nodes del árbol.

Pregunta

¿Que complejidad tiene este algoritmo?

Busquedas en arboles

- Hemos visto como buscar un elemento en un árbol – visitar todos los nodes del árbol.

Pregunta

¿Que complejidad tiene este algoritmo?

- $O(n)$

Pregunta

¿Podemos mejorar?

Busquedas en arboles

- Hemos visto como buscar un elemento en un árbol – visitar todos los nodes del árbol.

Pregunta

¿Que complejidad tiene este algoritmo?

- $O(n)$

Pregunta

¿Podemos mejorar?

- Podemos implementar (búsqueda binaría) “binary search” usando un árbol binario.

Árboles Binarios de Búsqueda (Binary Search Trees)

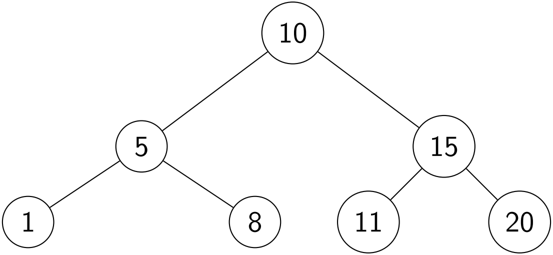
Árbol Binario de Búsqueda

"Un árbol binario de busqueda es un árbol binario con claves dentro elementos, donde para todo nodo con clave k:

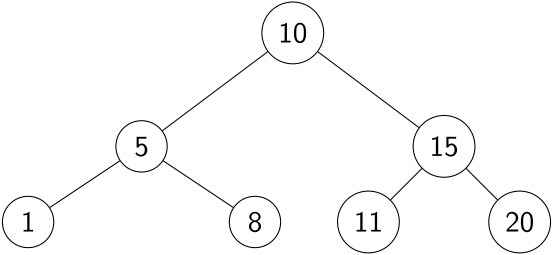
- *su clave $k \geq k'$ para todos claves k' en su subarbol izquierdo*
- *su clave $k \leq k''$ para todos claves k'' en su subarbol derecho*

"

Árboles Binarios de Búsqueda: Ejemplo

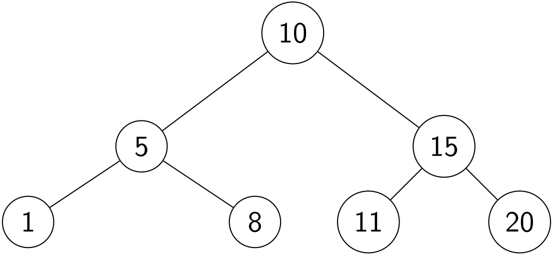


Árboles Binarios de Búsqueda: Busqueda



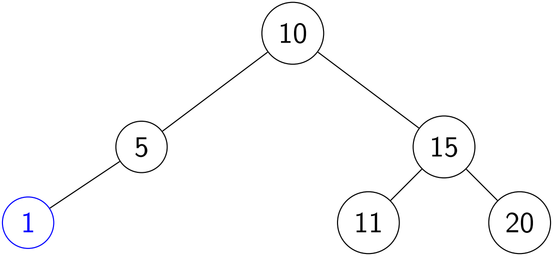

```
boolean member(BinaryTree<Integer> t, Position<Integer> p,
     Integer elem) {
  if (p == null) return false;
  else if (elem == p.element()) return true;
  else if (elem < p.element())
     return member(t, t.left(p), elem);
  else
     return member(t, t.right(p), elem);
}
```

Árboles Binarios de Búsqueda: Otros Operaciones



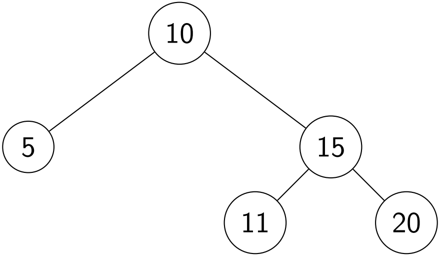
- insercion de una clave nueva es facil (`insert`)
- borrando una clave nueva es un poquito mas complicado (`remove`)
- es trivial traversar las claves en orden (menor - major): $O(n)$

Árboles Binarios de Búsqueda: Remove



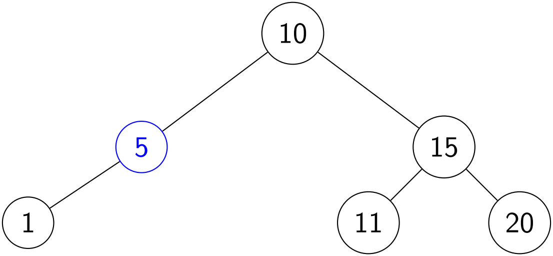
- Borrar un nodo (1) sin hijos es trivial – se borra el nodo.

Árboles Binarios de Búsqueda: Remove



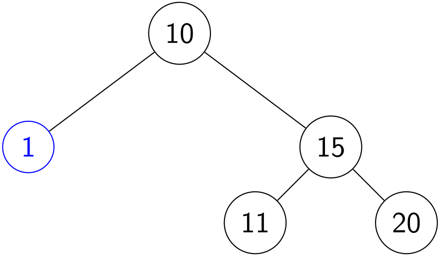
- Borrar un nodo (1) sin hijos es trivial – se borra el nodo.

Árboles Binarios de Búsqueda: Remove



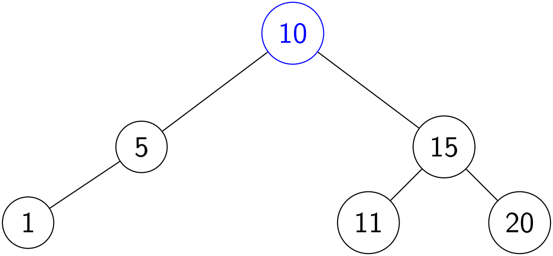
- Borrar un nodo (5) con un hijo es facil – se mueve el node hijo (1) a la posicion de su padre (5).

Árboles Binarios de Búsqueda: Remove



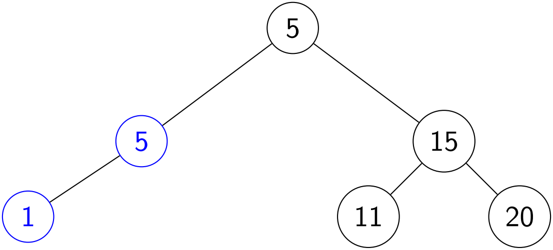
- Borrar un nodo (5) con un hijo es facil – se mueve el node hijo (1) a la posicion de su padre (5).

Árboles Binarios de Búsqueda: Remove



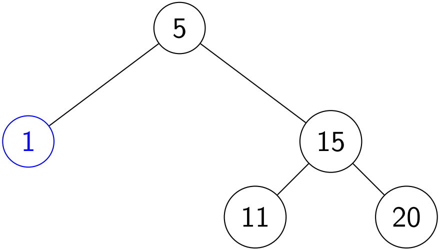
- Borrar un nodo (10) con dos hijo es un poquito mas complicado – se puede **mover la clave maxima desde el subarbol izquierdo** – (5) al nodo con la clave 10. Y despues borrar el antiguo nodo (5).
- O, vice versa, se puede **mover la clave minima desde el subarbol derecha** – (11) – al nodo con la clave 10.

Árboles Binarios de Búsqueda: Remove



- Borrar un nodo (10) con dos hijo es un poquito mas complicado – se puede **mover la clave maxima desde el subarbol izquierdo** – (5) al nodo con la clave 10. Y despues borrar el antiguo nodo (5).
- O, vice versa, se puede **mover la clave minima desde el subarbol derecha** – (11) – al nodo con la clave 10.

Árboles Binarios de Búsqueda: Remove



- Borrar un nodo (10) con dos hijo es un poquito mas complicado – se puede **mover la clave maxima desde el subarbol izquierdo** – (5) al nodo con la clave 10. Y despues borrar el antiguo nodo (5).
- O, vice versa, se puede **mover la clave minima desde el subarbol derecha** – (11) – al nodo con la clave 10.

Complejidad de Búsqueda

Pregunta

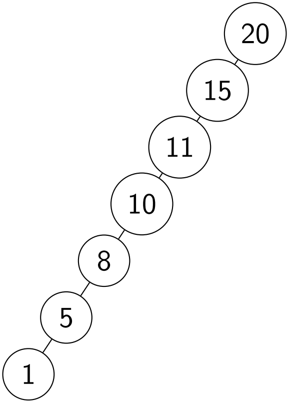
¿Que complejidad algoritmico tiene la busqueda en un arbol binario de busqueda?

Complejidad de Búsqueda

Pregunta

¿Que complejidad algoritmico tiene la busqueda en un arbol binario de busqueda?

$O(n)$, este también es un árbol binario de busqueda:



Búsquedas en Arboles Binarios

Pregunta

¿Podemos mejorar?

Búsquedas en Arboles Binarios

Pregunta

¿Podemos mejorar?

- **Si**, existe una gran variedad de tipos arboles binarios y algoritmos donde se intenta reducir la altitura de arboles.
- Ejemplo: arboles **AVL** – *la diferencia en altitura de dos subarboles es como maximo* 1.
  - AVL es un ejemplo de un *self-balancing binary search tree*: un tipo de árbol que se equilibra si misma – reduce el tamaño de la altitura del árbol durante las operaciónes de insertar y borrar (que son mas complicadas).
  - AVL tiene complejidad algoritmico $O(log\ n)$ de busqueda, insertar y borrar incluso en el peor caso.
- Otro ejemplos:
  - **red-black trees**
  - **splay trees** (es mas barato acceder a nodos frecuentamente usados)
  - **b-trees** (n-ary search trees), usado frequentamente en basos de datos usando un disco duro.