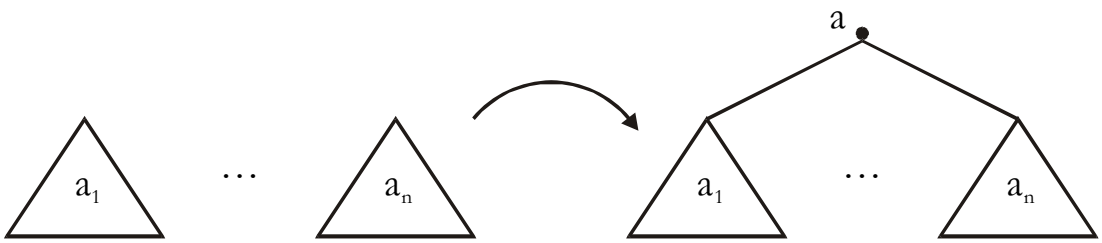
ARBOLES

- Modelo matemático y especificación
- Técnicas de implementación
- Recorridos de árboles binarios
- Arboles de búsqueda
- Colas de prioridad y montículos

5.1 Modelo matemático y especificación

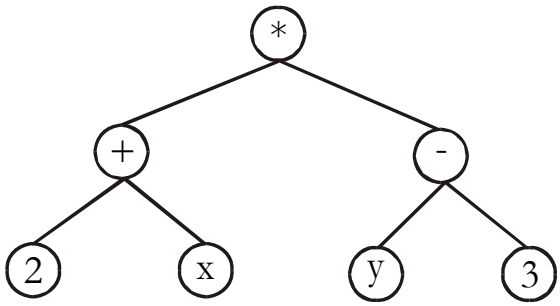
- Los árboles son estructuras jerárquicas formadas por *nodos*, de acuerdo con la siguiente construcción inductiva:
  - Un solo nodo forma un árbol *a*; se dice que el nodo es *raíz* del árbol.

  - Dados *n* árboles ya construidos a_1, … , a_n , se puede construir un nuevo árbol *a* añadiendo un nuevo nodo como raíz y conectándolo con las raíces de los a_i. Se dice que los a_i son los *hijos* de *a*.

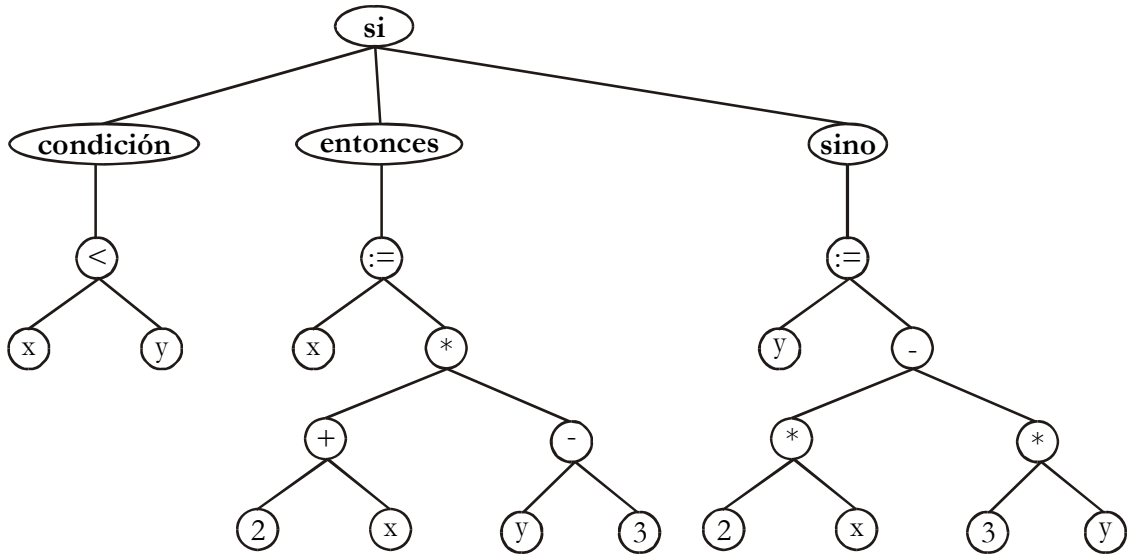


- Los árboles tienen muchos usos para representar jerarquías y clasificaciones, dentro y fuera de la Informática: árboles genealógicos, árboles taxonómicos, organización de un libro en capítulos y secciones, estructura de directorios y archivos, árbol de llamadas de una función recursiva, …

  Se utilizan también para representar estructuras sintácticas, como expresiones



o incluso sentencias de un lenguaje



- Podemos clasificar los árboles atendiendo a distintos criterios
  - Ordenados o no ordenados

    Un árbol es ordenado si el orden de los hijos de cada nodo es relevante.
  - Etiquetados o no etiquetados

    Un árbol está etiquetado si hay informaciones asociadas a sus nodos.
  - Generales o n-arios

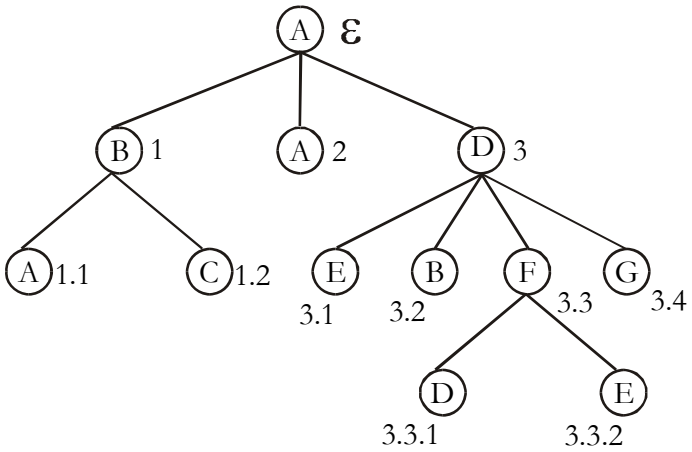
    Un árbol se llama *general* si no hay una limitación fijada al número de hijos de cada nodo. Si el número de hijos está limitado a un valor fijo *n*, se dice que el árbol es *n*-ario (*binario*, *ternario*, …).
  - Con o sin punto de interés

    En un árbol con punto de interés hay un nodo distinguido (aparte de la raíz, que es distinguida en cualquier árbol).

5.1.1 Modelo matemático

Arboles generales

- Adoptamos un modelo de árbol basado en la idea de representar las posiciones de los nodos como cadenas de números naturales positivos:
  - La raíz de un árbol tiene como posición la *cadena vacía* ε.
  - Si un cierto nodo de un árbol tiene posición $\alpha \in \aleph_+^*$, el hijo número *i* de ese nodo tendrá posición $\alpha.i$.



- De esta forma, un árbol se puede modelar como una aplicación:

  $a : N \rightarrow V$

  donde $N \subseteq \aleph_+^*$ es el conjunto de posiciones de los nodos, y V es el conjunto de valores posibles (*etiquetas*) asociados a los nodos.

  En el ejemplo anterior

  $N = \{ \varepsilon, 1, 2, 3, 1.1, 1.2, 3.1, 3.2, 3.3, 3.4, 3.3.1, 3.3.2 \}$

  $a(\varepsilon) = 'A'$
  $a(1) = 'B' \quad a(2) = 'A' \quad a(3) = 'D' \quad$ etc.

- En general, un conjunto $N \subseteq \aleph_+^*$ de cadenas de números naturales positivos, debe cumplir las siguientes condiciones para ser válido como conjunto de posiciones de los nodos de un árbol general:
  - N es finito.
  - $\varepsilon \in N$ posición de la raíz.
  - $\alpha.i \in N \Rightarrow \alpha \in N$ cerrado bajo prefijos.
  - $\alpha.i \in N \wedge 1 \leq j < i \Rightarrow \alpha.j \in N$ hijos consecutivos sin huecos.

Terminología

- Dado un árbol $a : N \to V$
  - *Nodo* es cada posición, junto con la información asociada: $(\alpha, a(\alpha))$
  - *Raíz* es el nodo de posición ε.

    Hojas son los nodos de posición α tal que no existe *i* tal que $\alpha.i \in N$.

    Nodos internos son los nodos que no son hojas.
  - Un nodo $\alpha.i$ tiene como *padre* a α, y se dice que es *hijo* de α.
  - Dos nodos de posiciones $\alpha.i$, $\alpha.j$ $(i \neq j)$ se llaman *hermanos*.
  - *Camino* es una sucesión de nodos tal que cada uno es padre del siguiente:

    $$\alpha, \alpha.i_1, \ldots, \alpha.i_1. \ldots .i_n$$

    n es la *longitud* del camino.
  - *Rama* es cualquier camino que comience en la raíz y termine en una hoja.
  - El *nivel* o *profundidad* de un nodo de posición α es $|\alpha|+1$. Es decir: $n+1$, siendo *n* la longitud del camino (único) que va de la raíz al nodo. En particular,
    - el nivel de la raíz es 1, y
    - el nivel de un nodo es igual al número de nodos del camino que va desde la raíz al nodo.
  - La *talla*, *altura* o *profundidad* de un árbol es el máximo de todos los niveles de nodos del árbol. Equivalentemente: $1+n$, siendo *n* el máximo de las longitudes de las ramas.
  - El *grado* o *aridad* de un nodo interno es su número de hijos. La aridad de un árbol es el máximo de las aridades de todos sus nodos internos.
  - Si hay un camino del nodo de posición α al nodo de posición β (i.e., si α es prefijo de β), se dice que α es *antepasado* de β y que β es *descendiente* de α.
  - Cada nodo de un árbol *a* determina un *subárbol* a_0 con raíz en ese nodo. Formalmente, si el nodo tiene posición α, entonces:

    $$a_0 : N_0 \to V$$

    siendo

    $$N_0 =_{def} \{ \beta \in \aleph_+^* \mid \alpha\beta \in N \}$$
    $$a_0(\beta) =_{def} a(\alpha\beta) \quad \text{para cada } \beta \in N_0$$
  - Los subárboles de un árbol *a* (si existen), se llaman árboles *hijos* de *a*.

Arboles binarios

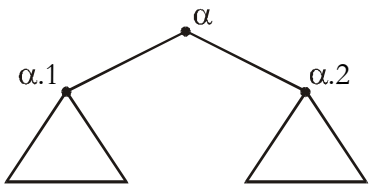
- Los árboles binarios se definen de tal modo que cada nodo interno tiene como máximo dos hijos.
  - Las posiciones de los nodos de estos árboles pueden representarse como cadenas $\alpha \in \{1,2\}^*$.
  - En caso de que un nodo interno (con posición α) tenga un solo hijo, se distingue si éste es *hijo izquierdo* (con posición $\alpha.1$) o *hijo derecho* (con posición $\alpha.2$).

  Un árbol binario etiquetado con valores de tipo V se modela entonces como una aplicación:

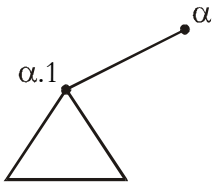
  $a : N \to V$

  donde el conjunto N de posiciones de los nodos de a debe cumplir las siguientes condiciones:

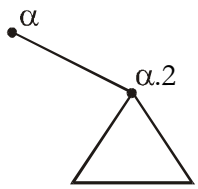
  - $N \subseteq \{1,2\}^*$, finito.
  - $\alpha.i \in N \Rightarrow \alpha \in N$ cerrado bajo prefijos.
  - No se exige que no haya huecos, por lo que si $\alpha \in N$ se puede dar uno de los 4 casos siguientes:
    - $\alpha.1 \in N$ y $\alpha.2 \in N$

    - $\alpha.1 \in N$ y $\alpha.2 \notin N$

    - $\alpha.1 \notin N$ y $\alpha.2 \in N$

    - $\alpha.1 \notin N$ y $\alpha.2 \notin N$

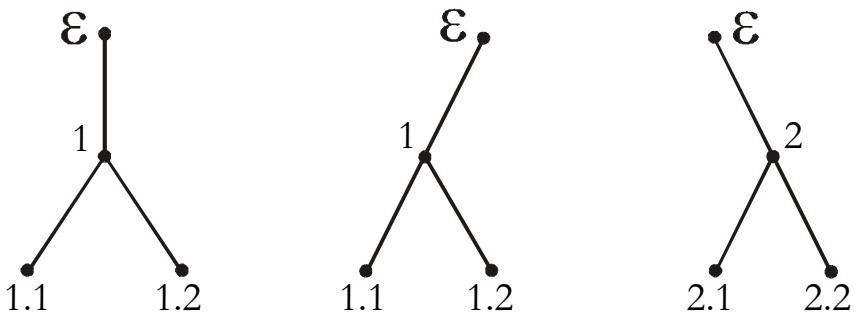
      

- Cuando falta alguno de los hijos de un nodo interno, se dice que el hijo inexistente es *vacío*. En particular, podemos decir que las hojas tienen dos hijos vacíos. Por lo tanto, si aceptamos la idea de *árbol vacío*, cuyo conjunto de posiciones N es $\varnothing \subseteq \{1,2\}^*$, en un árbol binario cada nodo tiene exactamente dos hijos.

Arboles n-arios

- Se definen como generalización de los árboles binarios, de tal manera que cada nodo interno tiene exactamente *n* hijos ordenados, cualquiera de los cuales puede ser vacío (i.e., se admiten "huecos" en la sucesión de hijos de un nodo). En particular, las hojas tienen *n* hijos vacíos.

  Observemos que "árbol n-ario" no es lo mismo que "árbol de grado *n*". Por ejemplo,



a_1 a_2 a_3

  a_1 es un árbol general de grado 2, pero no es un árbol binario. Los árboles a_2 y a_3 sí son binarios.

Arboles con punto de interés

- Como representación matemática de un árbol con punto de interés podemos adoptar una pareja de la forma:

  (a, α_0)

  donde *a* es un árbol

  $a : N \rightarrow V$
  $N \subseteq \aleph_+^*$

  y $\alpha_0 \in N$ indica la posición del punto de interés.

5.1.2 Especificación

- En los árboles binarios necesitamos operaciones para: generar el árbol vacío, construir árboles no vacíos, consultar la raíz y los hijos, y reconocer el árbol vacío.

  Gracias al concepto de *árbol vacío*, podemos suponer que cualquier árbol binario no vacío se construye a partir de un elemento raíz y dos árboles hijos (que pueden a su vez ser o no vacíos).

- De esta forma, llegamos a la siguiente especificación algebraica

```
tad ARBIN[E :: ANY]
 usa
    BOOL
 tipos
    Arbin[Elem]
 operaciones
    Nuevo: → Arbin[Elem]                                          /* gen */
    Cons: (Arbin[Elem], Elem, Arbin[Elem]) → Arbin[Elem]          /* gen */
    hijoIz, hijoDr: Arbin[Elem] – → Arbin[Elem]                   /* mod */
    raiz: Arbin[Elem] – → Elem                                    /* obs */
    esVacio: Arbin[Elem] → Bool                                   /* obs */
 ecuaciones
    ∀ iz, dr : Arbin[Elem] : ∀ x : Elem :
    def hijoIz(Cons(iz, x, dr))
    hijoIz(Cons(iz, x, dr))      = iz
    def hijoDr(Cons(iz, x, dr))
    hijoDr(Cons(iz, x, dr))      = dr
    def raiz(Cons(iz, x, dr))
    raiz(Cons(iz, x, dr))        = x
    esVacio(Nuevo)               = cierto
    esVacio(Cons(iz, x, dr))     = falso
 errores
    hijoIz(Nuevo)
    hijoDr(Nuevo)
    raiz(Nuevo)
ftad
```

5.2 Técnicas de implementación

Implementación dinámica de los árboles binarios

Tipo representante

- Estructura de datos

```
template <class TElem>
class TNodoArbin {
  private:
    TElem _elem;
    TNodoArbin<TElem> *_iz, *_dr;
...
}
template <class TElem>
class TArbinDinamico {
...
  private:
     TNodoArbin<TElem>* _ra;
...
}
```

- El problema es que, con la elección de generadoras que hemos hecho –*Nuevo* y *Cons*– resulta muy costoso controlar la compartición de estructura.

  Por ejemplo:

```
typedef TArbinDinamico<TEntero> TArbinEnt;
TArbinEnt construyeArbin( int ini, int fin ) {
  int m;

  if ( ini > fin )
    return TArbinEnt( );
  else {
    m = (ini + fin) / 2;
    TArbinEnt iz = construyeArbin( ini, m-1 );
    TArbinEnt dr = construyeArbin( m+1, fin);
    return TArbinEnt( iz, m, dr );
  };
}
```

  ¿Cómo se debe implementar la constructora *TArbin(TArbin, TElem, TArbin)* para que este ejemplo funcione correctamente?

  ¿Cuántas copias y anulaciones se realizan en la invocación *construyeArbin(1,2)* ?

Interfaz de la implementación

```
#ifndef arbin_dinamicoH
#define arbin_dinamicoH

#include <iostream>
#include "secuencia_dinamica.h"

using namespace std;

template <class TElem>
class TArbinDinamico;

template <class TElem>
class TNodoArbin {
  protected:
    TElem _elem;
    TNodoArbin<TElem> *_iz, *_dr;
    TNodoArbin( const TElem&, TNodoArbin<TElem>*, TNodoArbin<TElem>* );
  public:
    const TElem& elem() const;
    TNodoArbin<TElem> * iz() const;
    TNodoArbin<TElem> * dr() const;
    friend TArbinDinamico<TElem>;
};
```

```
template <class TElem>
class TArbinDinamico {
  public:

  // Constructoras, destructora y operador de asignación
     TArbinDinamico( );
     TArbinDinamico( const TArbinDinamico<TElem>&,
                     const TElem&,
                     const TArbinDinamico<TElem>& );
     TArbinDinamico( const TArbinDinamico<TElem>& );
     ~TArbinDinamico( );
     TArbinDinamico<TElem>& operator=( const TArbinDinamico<TElem>& );

  // Operaciones de los árboles
     TArbinDinamico<TElem> hijoIz ( ) const throw (EAccesoIndebido);
     // Pre : ! esVacio( )
     // Post : devuelve una copia del subárbol izquierdo
     // Lanza la excepción EAccesoIndebido si el árbol está vacío

     TArbinDinamico<TElem> hijoDr ( ) const throw (EAccesoIndebido);
     // Pre : ! esVacio( )
     // Post : devuelve una copia del subárbol derecho
     // Lanza la excepción EAccesoIndebido si el árbol está vacío

     // observadoras
     const TElem& raiz( ) const throw (EAccesoIndebido);
     // Pre : ! esVacio( )
     // Post : devuelve el elemento almacenado en la raíz
     // Lanza la excepción EAccesoIndebido si el árbol está vacío

     bool esVacio( ) const;
     // Pre: true
     // Post: Devuelve true | false según si el árbol está o no vacío
```

```
      // Recorridos
      TSecuenciaDinamica<TElem> preOrd( ) const;
      // Pre : ! esVacio( )
      // Post : devuelve el recorrido en pre-orden del árbol

      TSecuenciaDinamica<TElem> postOrd( ) const;
      // Pre : ! esVacio( )
      // Post : devuelve el recorrido en post-orden del árbol

      TSecuenciaDinamica<TElem> inOrd( ) const;
      // Pre : ! esVacio( )
      // Post : devuelve el recorrido en in-orden del árbol

   // Escritura
      void escribe( ostream& salida ) const;

   private:
   // Variables privadas
      TNodoArbin<TElem>* _ra;

   // Operaciones privadas
      void libera();
      static void TArbinDinamico<TElem>::liberaAux( TNodoArbin<TElem>* );
      void copia( const TArbinDinamico<TElem>& );
      static TNodoArbin<TElem>* copiaAux( TNodoArbin<TElem>* );
   // operación privada de escritura
      static void escribeAux( ostream& salida,
                              TNodoArbin<TElem>* p, string prefijo );
   // Constructora privada para los subárboles
      TArbinDinamico( TNodoArbin<TElem>* );
};
```

- En los algoritmos recursivos debemos utilizar una operación auxiliar debido a que *TArbinDinamico* y *TNodoArbin* son tipos distintos.

Implementación de las operaciones

- Clase de los nodos

```
template <class TElem>
TNodoArbin<TElem>::TNodoArbin( const TElem& elem,
                               TNodoArbin<TElem>* iz = 0,
                               TNodoArbin<TElem>* dr = 0 ) :
  _elem(elem), _iz(iz), _dr(dr) {
};

template <class TElem>
const TElem& TNodoArbin<TElem>::elem() const {
  return _elem;
}

template <class TElem>
TNodoArbin<TElem>* TNodoArbin<TElem>::iz() const {
  return _iz;
}

template <class TElem>
TNodoArbin<TElem>* TNodoArbin<TElem>::dr() const {
  return _dr;
}
```

- Constructoras, destructora y operador de asignación

```
template <class TElem>
TArbinDinamico<TElem>::TArbinDinamico( ) :
  _ra( 0 ) {
};

template <class TElem>
TArbinDinamico<TElem>::TArbinDinamico( const TArbinDinamico<TElem>& iz,
                                       const TElem& elem,
                                       const TArbinDinamico<TElem>& dr ) :
  _ra( new TNodoArbin<TElem>( elem, copiaAux(iz._ra), copiaAux(dr._ra) ) ){
};

template <class TElem>
TArbinDinamico<TElem>::TArbinDinamico( const TArbinDinamico<TElem>& arbin){
   copia(arbin);
};

template <class TElem>
TArbinDinamico<TElem>::~TArbinDinamico( ) {
  libera();
};

template <class TElem>
TArbinDinamico<TElem>&
TArbinDinamico<TElem>::operator=( const TArbinDinamico<TElem>& arbin ) {
  if( this != &arbin ) {
    libera();
    copia(arbin);
  }
  return *this;
};
```

- Operaciones de los árboles

```
template <class TElem>
TArbinDinamico<TElem> TArbinDinamico<TElem>::hijoIz ( ) const
  throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido();
  else
    return TArbinDinamico<TElem>( copiaAux(_ra->iz()) );
};

template <class TElem>
TArbinDinamico<TElem> TArbinDinamico<TElem>::hijoDr ( ) const
  throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido();
  else
    return TArbinDinamico<TElem>( copiaAux(_ra->dr()) );
};

template <class TElem>
const TElem& TArbinDinamico<TElem>::raiz( ) const throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido();
  else
    return _ra->elem();
};

template <class TElem>
bool TArbinDinamico<TElem>::esVacio( ) const {
  return _ra == 0;
};
```

- Operaciones de entrada/salida

```
template <class TElem>
void TArbinDinamico<TElem>::escribeAux( ostream& salida,
                                        TNodoArbin<TElem>* p,
                                        string prefijo ) {
  if ( p != 0 ) {
    salida << ( prefijo + " : " ) << p->elem() << endl;
    escribeAux( salida, p->iz(), prefijo + ".1" );
    escribeAux( salida, p->dr(), prefijo + ".2" );
  }
}

template <class TElem>
void TArbinDinamico<TElem>::escribe( ostream& salida ) const {
  escribeAux( salida, _ra, "0" );
};
```

- Operaciones privadas

```
// Constructora privada para los subárboles
template <class TElem>
TArbinDinamico<TElem>::TArbinDinamico( TNodoArbin<TElem>* ra ) :
  _ra( ra ) {
};
```

- Copia y anulación

```
// anulación
template <class TElem>
void TArbinDinamico<TElem>::liberaAux( TNodoArbin<TElem>* p ) {
  if ( p != 0 ){
    liberaAux(p->iz());
    liberaAux(p->dr());
    delete p;
  }
};

template <class TElem>
void TArbinDinamico<TElem>::libera() {
  liberaAux( _ra );
};

// copia
template <class TElem>
TNodoArbin<TElem>* TArbinDinamico<TElem>::copiaAux( TNodoArbin<TElem>* p ){
  TNodoArbin<TElem>* r;

  if ( p == 0 )
    r = 0;
  else
    r = new TNodoArbin<TElem>( p->elem(),
                               copiaAux( p->iz() ),
                               copiaAux( p->dr() ) );
  return r;
};

template <class TElem>
void TArbinDinamico<TElem>::copia(const TArbinDinamico<TElem>& arb) {
  _ra = copiaAux( arb._ra );
};
```

Complejidad de las operaciones

- Tomando n = número de elementos del árbol

Operación	Tipo primitivo	Tipo definido
TArbin( )	$O(1)$	$O(1)$
TArbin(TArbin, TElem, TArbin)	$O(n)$ *	$O(n * T(\text{ TElem(TElem\&) }))$ *
TArbin(TArbin&)	$O(n)$	$O(n * T(\text{ TElem(TElem\&) }))$
~TArbin( )	$O(n)$	$O(n * T(\text{ ~TElem( ) }))$
TArbin& operator=(TArbin&)	$O(n)$	$O(n * T(\text{ ~TElem( ) }) + n * T(\text{ TElem(TElem\&) }))$
TArbin hijoIz( )	$O(n)$ **	$O(n * T(\text{ TElem(TElem\&) }))$ **
TArbin hijoDr( )	$O(n)$ ***	$O(n * T(\text{ TElem(TElem\&) }))$ ***
TElem& raiz( )	$O(1)$	$O(1)$
bool esVacio( )	$O(1)$	$O(1)$
escribe(ostream&)	$O(n)$	$O(n * T(\text{ operator<<(TElem) }))$

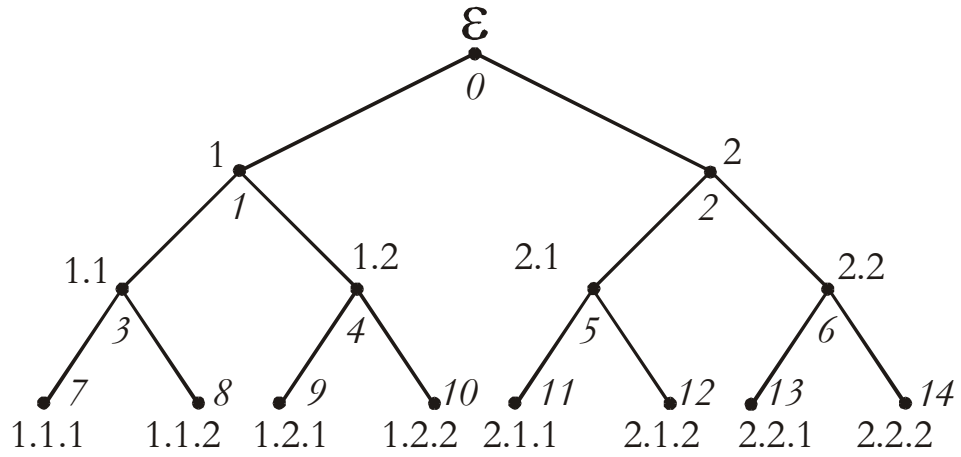
* Siendo n el número de nodos del árbol resultante

** Considerando el caso peor de un árbol donde el subárbol derecho es vacío

*** Considerando el caso peor de un árbol donde el subárbol izquierdo es vacío

Representación estática

- La idea consiste en calcular, en función de la posición de cada nodo del árbol, el índice del vector donde vamos a almacenar la información asociada a ese nodo. Para hacer esto necesitamos establecer una biyección entre posiciones y números positivos:



Esta numeración de posiciones corresponde a una biyección

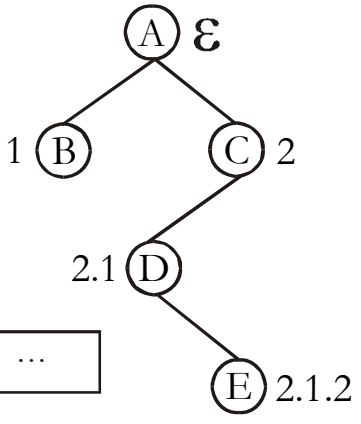
$$\text{índice}: \{1, 2\}^* \rightarrow \aleph$$

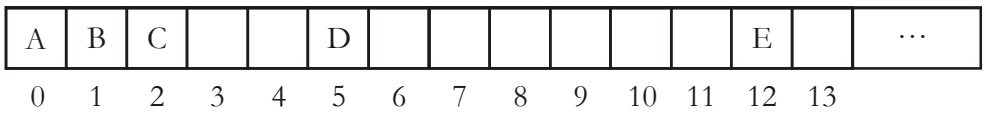
que admite la siguiente definición recursiva:

$$\text{índice}(\varepsilon) = 0$$
$$\text{índice}(\alpha.1) = 2 * \text{índice}(\alpha) + 1$$
$$\text{índice}(\alpha.2) = 2 * \text{índice}(\alpha) + 2$$

Con ayuda de *índice*, podemos representar un árbol binario en un vector, almacenando la información del nodo α en la posición *índice*(α). Por ejemplo:

$$\text{índice}(\varepsilon) = 0$$
$$\text{índice}(1) = 2 \cdot \text{índice}(\varepsilon) + 1 = 1$$
$$\text{índice}(2) = 2 \cdot \text{índice}(\varepsilon) + 2 = 2$$
$$\text{índice}(2.1) = 2 \cdot \text{índice}(2) + 1 = 5$$
$$\text{índice}(2.1.2) = 2 \cdot \text{índice}(2.1) + 2 = 12$$

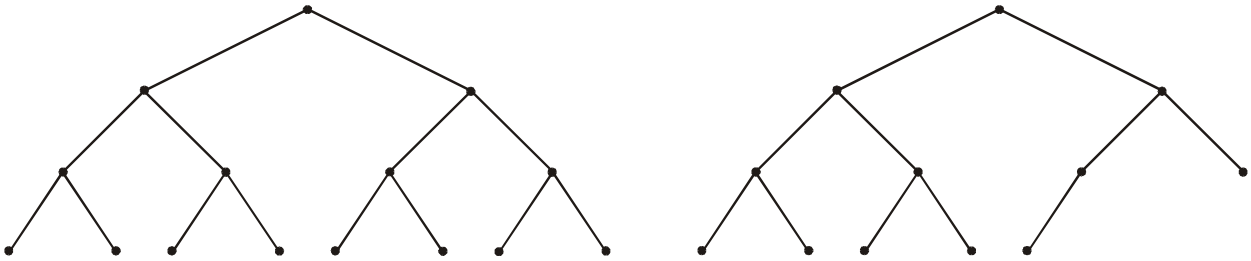




0 1 2 3 4 5 6 7 8 9 10 11 12 13

- Como vemos, pueden quedar muchos espacios desocupados si el árbol tiene pocos nodos en relación con su número de niveles. Por este motivo, esta representación sólo se suele aplicar a una clase especial de árboles binarios, que definimos a continuación.
  - Un árbol binario de talla *n* se llama *completo* si y sólo si todos sus nodos internos tienen dos hijos no vacíos, y todas sus hojas están en el nivel *n*.
  - Un árbol binario de talla *n* se llama *semicompleto* si y sólo si es completo o tiene vacantes una serie de posiciones consecutivas del nivel *n*, de manera que al rellenar dichas posiciones con nuevas hojas se obtiene un árbol completo.

  Por ejemplo:



- Un árbol binario completo de talla *n* tiene el máximo número de nodos que puede tener un árbol de esa talla. En concreto se verifica:
  - El número de nodos de cualquier nivel *i* en un árbol binario completo es:
    $m_i = 2^{i-1}$
  - El número total de nodos de un árbol binario completo de talla *n* es:
    $M_n = 2^n - 1.$

  Como se puede demostrar fácilmente por inducción sobre el nivel *i*

  $i = 1 \qquad m_1 = 1 = 2^{1-1}$

  $i > 1 \qquad m_i = 2 \cdot m_{i-1} =_{H.I.} 2 \cdot 2^{i-2} = 2^{i-1}$

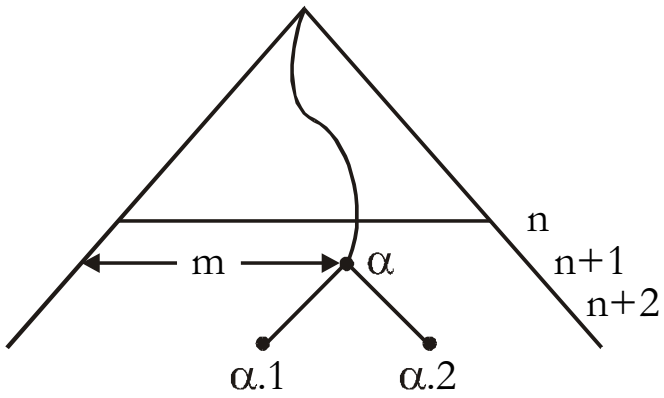
  y aplicando el resultado de la suma de una progresión geométrica

  $$M_n = \sum_{i=1}^{n} m_i = \sum_{i=1}^{n} 2^{i-1} = 2^n - 1$$

  Como corolario, la talla de un árbol binario completo con *M* nodos es *log(M+1)*.

- Usando estos dos resultados podemos demostrar que la definición recursiva de *índice* presentada anteriormente es correcta. La definición no recursiva es como sigue: si α es un posición de nivel $n+1$ ($n \geq 0$)

índice(α) = número de posiciones de niveles 1..n +
número de posiciones de nivel n+1 hasta α inclusive – 1
= $(2^n - 1) + m - 1$
= $2^n + m - 2$



índice(ε) = 0

índice(α.1) = número de posiciones de niveles 1..(n+1) +
número de posiciones de nivel n+2 hasta α.1 inclusive – 1
= $(2^{n+1} - 1) + 2 \cdot (m - 1) + 1 - 1$
= $2^{n+1} + 2m - 3$
= $2^{n+1} + 2m - 4 + 1$
= $2 \cdot (2^n + m - 2) + 1$
= $2 \cdot$ índice(α) + 1

índice(α.2) = número de posiciones de niveles 1..(n+1) +
número de posiciones de nivel n+2 hasta α.2 inclusive – 1
= $(2^{n+1} - 1) + 2 \cdot (m - 1) + 2 - 1$
= $2^{n+1} + 2m - 2$
= $2^{n+1} + 2m - 4 + 2$
= $2 \cdot (2^n + m - 2) + 2$
= $2 \cdot$ índice(α) + 2

Implementación estática de árboles binarios semicompletos

- Recapitulando lo expuesto hasta ahora tenemos que, si un árbol binario completo de talla *n* tiene 2^n-1 nodos, entonces, declarando

```
const int max = 64 - 1; // (potencia de 2) – 1
TElem espacio[max];
```

  tendremos un vector donde podemos almacenar cualquier árbol binario de talla $\leq n$

- Dado un nodo α almacenado en la posición $índice(\alpha) = i$, tal que $0 \leq i < max$, son válidas las siguientes fórmulas para el cálculo de otro nodos relacionados con i:
  - Hijo izquierdo: $2i+1$, si $2i+1 < max$
  - Hijo derecho: $2i+2$, si $2i+2 < max$
  - Padre: $(i-1)$ *div* 2, si $i > 0$

    Dado un cierto $i > 0$,
    - si i es impar, entonces su padre estará en una posición $j \geq 0$ tal que

      $i = 2j + 1$

      $\Leftrightarrow \quad j = (i-1) / 2$

      $\Leftrightarrow \quad j = (i-1)$ *div* 2
    - si i es par, entonces su padre estará en una posición $j \geq 0$ tal que

      $i = 2j + 2$

      $\Leftrightarrow \quad j = (i-2) / 2$

      $\Leftrightarrow \quad j = (i-1)$ *div* 2

- Esta representación es útil para árboles completos y semicompletos, cuando estos se generan y procesan mediante operaciones que hagan crecer al árbol por niveles –habría que cambiar la especificación del TAD–.

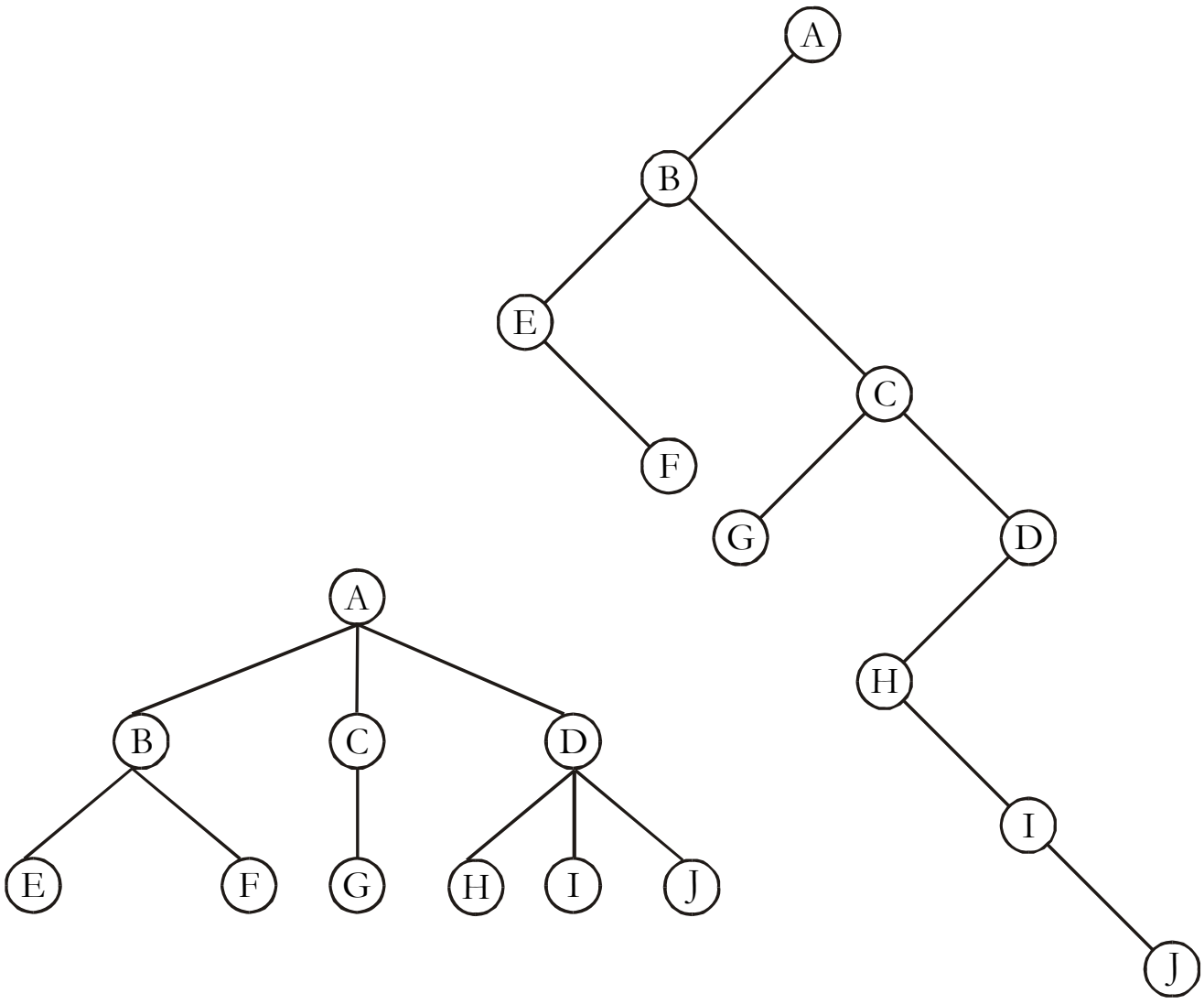
Implementación de los árboles generales

- La idea más habitual consiste en representar los nodos del árbol como registros con tres campos: información asociada, puntero al primer hijo, y puntero al hermano derecho.

  En realidad, esta idea se basa en la posibilidad de representar un árbol general cualquiera como árbol binario, a través de la siguiente conversión:

ARBOL GENERAL	ARBOL BINARIO
Primer hijo	Hijo izquierdo
Hermano derecho	Hijo derecho

Por ejemplo:



5.3 Recorridos

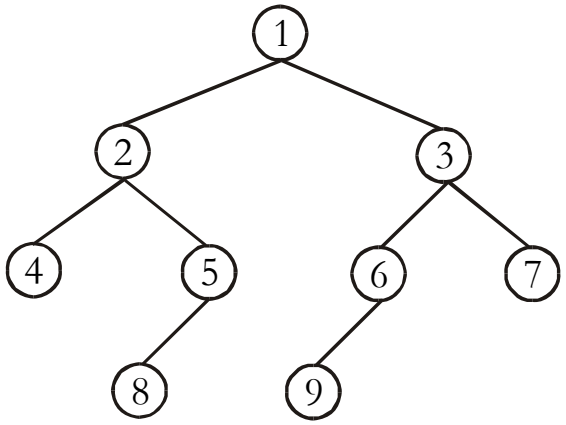
- En muchas aplicaciones los árboles se utilizan como una estructura intermedia que luego ha de transformarse en una estructura lineal que será procesada. Esta transformación implica el recorrido del árbol, visitando cada uno de sus nodos.

  Los principales recorridos de árboles binarios se clasifican como sigue:

  Recorridos
  - En profundidad
    - Preorden (RID)
    - Inorden (IRD)
    - Postorden (IDR)
  - Por niveles

  Los recorridos en profundidad se basan en la relación padre-hijos y se clasifican según el orden en que se consideren la raíz (R), el hijo izquierdo (I) y el hijo derecho (D).

  Ejemplo:



  Preorden: 1, 2, 4, 5, 8, 3, 6, 9, 7
  Inorden: 4, 2, 8, 5, 1, 9, 6, 3, 7
  Postorden: 4, 8, 5, 2, 9, 6, 7, 3, 1
  Niveles: 1, 2, 3, 4, 5, 6, 7, 8, 9

  Los recorridos de árboles tienen aplicaciones muy diversas, dependiendo de la información que representen los árboles en cuestión.

Recorridos en profundidad

Especificación algebraica

- Podemos especificar las nuevas operaciones como un enriquecimiento del TAD ARBIN.

```
tad REC-PROF-ARBIN[E :: ANY]
 hereda
   ARBIN[E]
 usa
   SEC[E]
 operaciones
   preOrd, inOrd, postOrd: Arbin[Elem] → Sec[Elem]                  /* obs */
 operaciones privadas
   preOrdAcu, inOrdAcu, postOrdAcu:
     (Arbin[Elem], Sec[Elem]) → Sec[Elem]                           /* obs */
 ecuaciones
   ∀ a, iz, dr : Arbin[Elem] : ∀ xs : Sec[Elem] : ∀ x : Elem :

   preOrd(a)                        = preOrdAcu(a, SEC.Nuevo)
   preOrdAcu(Nuevo, xs)             = xs
   preOrdAcu(Cons(iz, x, dr), xs)  =
                           preOrdAcu(dr, preOrdAcu(iz, Inserta(xs, x)))

   inOrd(a)                         = inOrdAcu(a, SEC.Nuevo)
   inOrdAcu(Nuevo, xs)              = xs
   inOrdAcu(Cons(iz, x, dr), xs)   =
                           inOrdAcu(dr, Inserta(inOrdAcu(iz, xs), x))

   postOrd(a)                       = postOrdAcu(a, SEC.Nuevo)
   postOrdAcu(Nuevo, xs)            = xs
   postOrdAcu(Cons(iz, x, dr), xs) =
                           Inserta(postOrdAcu(dr, postOrdAcu(iz, xs)), x)
ftad
```

- La implementación recursiva es directa a partir de la especificación. Veamos como ejemplo la implementación del *preOrden*:

```
template <class TElem>
void preOrdAcu( TNodoArbin<TElem>* p, TSecuenciaDinamica<TElem>& xs ) {
  if ( p != 0 ) {
    xs.inserta( p->elem() );
    preOrdAcu( p->iz(), xs );
    preOrdAcu( p->dr(), xs );
  }
};

template <class TElem>
TSecuenciaDinamica<TElem> TArbinDinamico<TElem>::preOrd( ) const {
  TSecuenciaDinamica<TElem> r;

  preOrdAcu( _ra, r );
  return r;
};
```

Evidentemente los otros dos recorridos se implementan de forma análoga.

```
template <class TElem>
void postOrdAcu( TNodoArbin<TElem>* p, TSecuenciaDinamica<TElem>& xs ){
  if ( p != 0 ) {
    postOrdAcu( p->iz(), xs );
    postOrdAcu( p->dr(), xs );
    xs.inserta( p->elem() );
  }
};

template <class TElem>
TSecuenciaDinamica<TElem> TArbinDinamico<TElem>::postOrd( ) const {
  TSecuenciaDinamica<TElem> r;

  postOrdAcu( _ra, r );
  return r;
};
```

```
template <class TElem>
void inOrdAcu( TNodoArbin<TElem>* p, TSecuenciaDinamica<TElem>& xs ) {
  if ( p != 0 ) {
    inOrdAcu( p->iz(), xs );
    xs.inserta( p->elem() );
    inOrdAcu( p->dr(), xs );
  }
};

template <class TElem>
TSecuenciaDinamica<TElem> TArbinDinamico<TElem>::inOrd( ) const {
  TSecuenciaDinamica<TElem> r;

  inOrdAcu( _ra, r );
  return r;
};
```

- En cuanto a la complejidad, dado que la inserción en las secuencias es $O(1)$, obtenemos una complejidad $O(n)$ para los tres recorridos:

$$\mathrm{T}(n) = \begin{cases} c_1 & \textbf{si } 0 \le n < b \\ a \cdot \mathrm{T}(n/b) + c \cdot n^k & \textbf{si } n \ge b \end{cases}$$

$$\mathrm{T}(n) \in \begin{cases} O(n^{\mathrm{k}}) & \textbf{si } a < b^k \\ O(n^{\mathrm{k}} \cdot \log n) & \textbf{si } a = b^k \\ O(n^{\log_b a}) & \textbf{si } a > b^k \end{cases}$$

si suponemos que se trata de un árbol completo, entonces tenemos:

$a = 2,\ b = 2,\ k = 0$

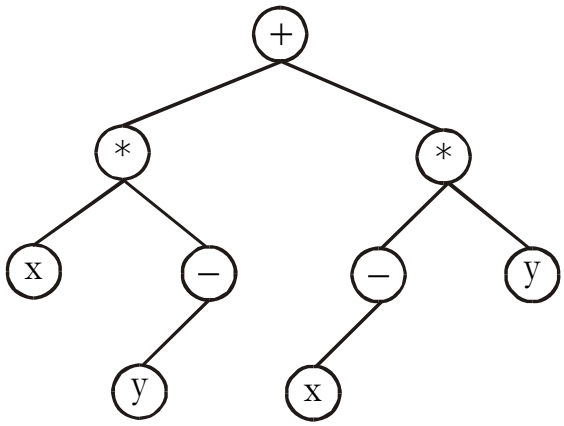
y por lo tanto

$a > b^k \Rightarrow \mathrm{T}(n) \in O(n^{\log_b a}) = O(n)$

Recorrido por niveles

- En este caso, el orden de recorrido no está tan relacionado con la estructura recursiva del árbol y por lo tanto no es natural utilizar un algoritmo recursivo.

  Se utiliza una cola de árboles, de forma que para recorrer un árbol con hijos, se visita la raíz y se ponen en la cola los dos hijos. Por ejemplo, para el árbol



Mostramos la cola de árboles indicando las posiciones de las raíces de los árboles almacenados en ella, cuando éstos se consideran como subárboles de *a*:

Cola (último a la derecha)	Recorrido
ε	
	+
1, 2	
	+ *
2, 1.1, 1.2	
	+ * *
1.1, 1.2, 2.1, 2.2	
	+ * * x
1.2, 2.1, 2.2	
	+ * * x –
2.1, 2.2, 1.2.1	
etc. …	

Especificación algebraica

- Necesitamos utilizar una cola de árboles auxiliar. La cola se inicializa con el árbol a recorrer. El recorrido de la cola de árboles tiene como caso base la cola vacía. Si de la cola se extrae un árbol vacío, se avanza sin más. Y si de la cola se extrae un árbol no vacío, se inserta la raíz en la secuencia resultado, y se insertan en la cola el hijo izquierdo y el derecho, por este orden

```
tad REC-NIVELES-ARBIN[E :: ANY]
  hereda
     ARBIN[E]
  usa
     SEC[E]
  usa privadamente
     COLA[ARBIN[E]]
  operaciones
     niveles: Arbin[Elem] → Sec[Elem]                              /* obs */
  operaciones privadas
     nivelesCola: (Cola[Arbin[Elem]], Sec[Elem]) → Sec[Elem]    /* obs */
  ecuaciones
     ∀ a : Arbin[Elem] : ∀ as : Cola[Arbin[Elem]] : ∀ xs : Sec[Elem]
     niveles(a)          = nivelesCola(PonDetras(a, COLA.Nuevo),
                                       SEC.Nuevo)

     nivelesCola(as, xs)  = xs  si COLA.esVacio(as)
     nivelesCola(as, xs)  = nivelesCola(quitaPrim(as), xs)
                                 si NOT COLA.esVacio(as) AND
                                    ARBIN.esVacio(primero(as))
     nivelesCola(as, xs)  =
                nivelesCola(PonDetras(hijoDr(primero(as)),
                                      PonDetras(hijoIz(primero(as)),
                                                quitaPrim(as))),
                            Inserta(xs, raiz(primero(as))))
                                 si NOT COLA.esVacio(as) AND
                                    NOT ARBIN.esVacio(primero(as))
ftad
```

- La implementación resultante

```
template <class TElem>
TSecuenciaDinamica<TElem> niveles( TArbinDinamico<TElem> a ) {
  TSecuenciaDinamica<TElem> r;
  TColaDinamica< TArbinDinamico<TElem> > c;
  TArbinDinamico<TElem> aux;

  c.ponDetras(a);
  while ( ! c.esVacio() ) {
    aux = c.primero();
    c.quitaPrim();
    if ( ! aux.esVacio() ) {
      r.inserta(aux.raiz());
      c.ponDetras(aux.hijoIz());
      c.ponDetras(aux.hijoDr());
    }
  }
  return r;
}
```

- Si todas las operaciones involucradas tienen complejidad $O(1)$, entonces el recorrido por niveles resulta de $O(n)$, siendo n el número de nodos: cada nodo pasa una única vez por la primera posición de la cola.

  Sin embargo, en *aux = c.primero( )* se hace una copia del árbol –$O(n)$– y las operaciones *hijoIz* e *hijoDr* también realizan copias –$2 \cdot O(n/2)$–.

  Suponiendo que se trata de un árbol completo, y aprovechándonos de la estructura recursiva de los árboles, podemos realizar el análisis como si se tratáse de un algoritmo recursivo:

$$\mathrm{T}(n) = \begin{cases} c_0 & \textbf{si } 0 \le n < 1 \\ 2 \cdot \mathrm{T}(n/2) + n + 2 \cdot n/2 + c_1 & \textbf{si } n \ge 1 \end{cases}$$

  Aprovechando los resultados teóricos para el cálculo de recurrencias con disminución del problema por división, tenemos

  $a = 2,\ b = 2,\ k = 1$

  y por lo tanto

  $a = b^k \Rightarrow \mathrm{T}(n) \in O(n^k \cdot \log n) = O(n \cdot \log n)$

  Eso sí, suponiendo que los elementos son de un tipo primitivo ...

5.4 Arboles de búsqueda

- Permiten representar colecciones de elementos ordenados utilizando memoria dinámica, con una complejidad $O(\log n)$ para las operaciones de inserción, borrado y búsqueda.

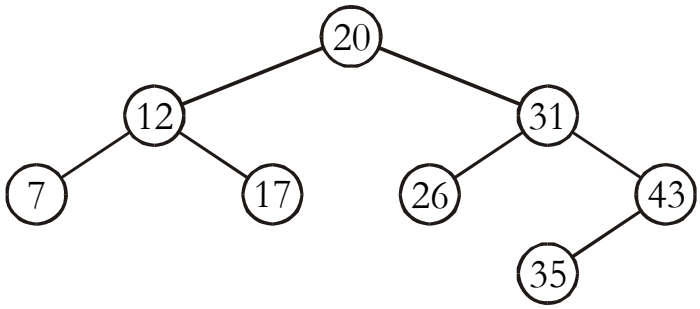
  Vamos a presentar las ideas básicas utilizando una simplificación de los árboles de búsqueda: los árboles ordenados.

5.4.1 Arboles ordenados

- Un árbol ordenado es un árbol binario que almacena elementos de un tipo de la clase ORD (con operaciones de comparación sobre igualdad y orden). Se dice que el árbol *a* está ordenado si se da alguno de los dos casos siguientes:
  - *a* es vacío.
  - *a* no es vacío, sus dos hijos están ordenados, todos los elementos del hijo izquierdo son estrictamente menores que el elemento de la raíz, y el elemento de la raíz es estrictamente menor que todos los elementos del hijo derecho.

  Nótese que según esta definición no se admiten elementos repetidos.

  Por ejemplo, el siguiente es un árbol ordenado:

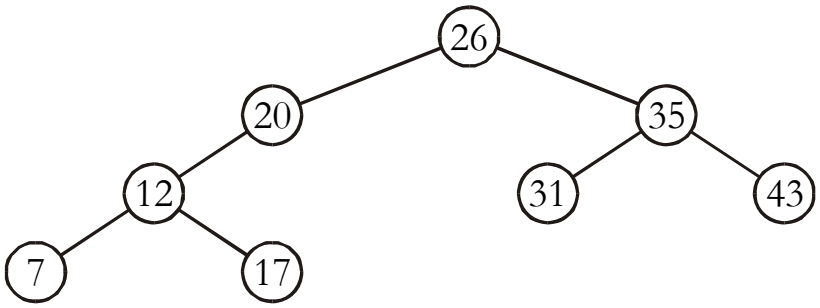


- Especificación algebraica de una operación que reconoce si un árbol binario está ordenado:

```
esOrdenado(Nuevo)           = cierto
esOrdenado(Cons(iz, x, dr)) = esOrdenado(iz) AND menor(iz, x) AND
                              esOrdenado(dr) AND mayor(dr, x)
menor(Nuevo, y)             = cierto
menor(Cons(iz, x, dr), y)   = x < y AND menor(iz, y) AND menor(dr, y)
mayor(Nuevo, y)             = cierto
mayor(Cons(iz, x, dr), y)   = x > y AND mayor(iz, y) AND mayor(dr, y)
```

- Una cualidad muy interesante de los árboles ordenados es que su recorrido en inorden produce una lista ordenada de elementos. De hecho, esta es una forma de caracterizar a los árboles ordenados:
  - Un árbol binario *a* es un árbol ordenado si y sólo si *xs* = *inOrd(a)* está ordenada.

- Es posible construir distintos árboles ordenados con la misma información. Por ejemplo, el siguiente árbol produce el mismo recorrido que el anterior:



- Las operaciones que nos interesan sobre árboles ordenados son: inserción, búsqueda, borrado, recorrido y consulta.

Inserción en un árbol ordenado

- La inserción se especifica de forma que si el árbol está ordenado, siga estándolo después de la inserción. La inserción de un dato *y* en un árbol ordenado *a*:

```
inserta(y, Nuevo)           = Cons(Nuevo, y, Nuevo)
inserta(y, Cons(iz, x, dr)) = Cons(iz, x, dr)             si y == x
inserta(y, Cons(iz, x, dr)) = Cons(inserta(y, iz), x, dr) si y < x
inserta(y, Cons(iz, x, dr)) = Cons(iz, x, inserta(y, dr)) si y > x
```

Se puede observar que el primer ejemplo que presentamos de árbol ordenado es el resultado de insertar sucesivamente: 20, 12, 17, 31, 26, 43, 7 y 35, en un árbol inicialmente vacío.

Búsqueda en un árbol ordenado

- Especificamos esta operación de forma que devuelva como resultado el subárbol obtenido tomando como raíz el nodo que contenga el árbol buscado. Si ningún nodo del árbol, contiene el elemento buscado, se devuelve el árbol vacío.

```
busca(y, Nuevo)           = Nuevo
busca(y, Cons(iz, x, dr)) = Cons(iz, x, dr)  if y == x
busca(y, Cons(iz, x, dr)) = busca(y, iz)     if y < x
busca(y, Cons(iz, x, dr)) = busca(y, dr)     if y > x
```

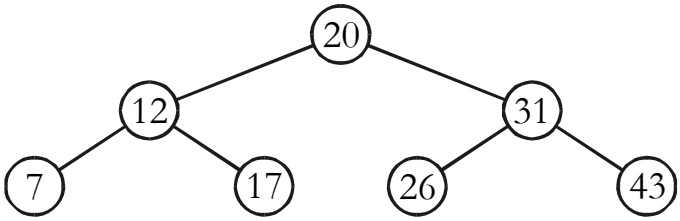
Borrado en un árbol ordenado

- La operación de borrado es la más complicada, porque tenemos que reconstruir el árbol resultado de la supresión. Se busca el valor *y* en el árbol,
  - si la búsqueda fracasa, la operación termina sin modificar el árbol,
  - si la búsqueda tiene éxito y localiza un nodo de posición α, el comportamiento de la operación depende del número de hijos de α:
    - Si α es una hoja, se elimina el nodo α
    - Si α tiene un solo hijo, se elimina el nodo α y se coloca en su lugar el árbol hijo, cuya raíz quedará en la posición α
    - Si α tiene dos hijos se procede del siguiente modo:
      - Se busca el nodo con el valor mínimo en el hijo derecho de α. Sea α' la posición de éste.
      - El elemento del nodo α se reemplaza por el elemento del nodo α'.
      - Se borra el nodo α'.

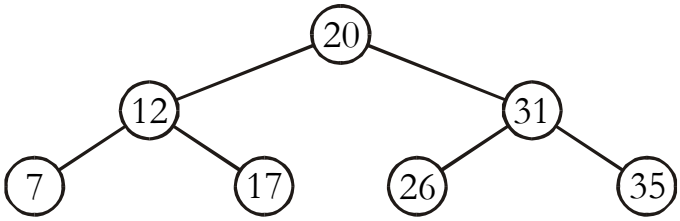
        Nótese que, por ser el mínimo del hijo derecho de α, α' no puede tener hijo izquierdo, por lo tanto, estaremos en la situación de eliminar una hoja o un nodo con un solo hijo –el derecho–.

```
borra(y, Nuevo)           = Nuevo
borra(y, Cons(iz, x, dr)) = dr       si y == x AND esVacío(iz)
borra(y, Cons(iz, x, dr)) = iz       si y == x AND esVacío(dr)
borra(y, Cons(iz, x, dr)) = Cons(iz, z, borra(z, dr))
                                     si y == x AND NOT esVacío(iz) AND
                                        NOT esVacío(dr) AND z = min(dr)
borra(y, Cons(iz, x, dr)) = Cons(borra(y, iz), x, dr)   si y < x
borra(y, Cons(iz, x, dr)) = Cons(iz, x, borra(y, dr))   si y > x
```

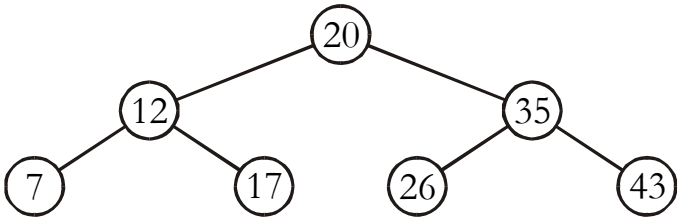
- Por ejemplo, aplicamos algunas operaciones de borrado al primer ejemplo que presentamos de árbol ordenado:
  - *borra*(35, *a*)



  - *borra*(43, *a*)



  - *borra*(31, *a*)



Especificación algebraica del TAD ARB-ORD

- Presentamos la especificación algebraica del TAD ARB-ORD como un enriquecimiento del TAD REC-PROF-ARBIN, que es, a su vez, es un enriquecimiento del TAD ARBIN. Para hacer más legible la especificación, renombramos algunos identificadores.

  Ocultamos las operaciones indeseables de forma que este TAD exporte el tipo *Arbord[Elem]* equipado con las operaciones: *Nuevo*, *raíz*, *esVacío*, *recorre*, *inserta*, *busca*, *borra* y *está*.

```
tad ARB-ORD[E :: ORD]
 hereda
   REC-PROF-ARBIN[E]  renombrando  inOrd a recorre
                                   Arbin[Elem] a Arbord[Elem]
                      ocultando    preOrd, postOrd,
                                   Cons, hijoIz, hijoDr
 tipo
      Arbord[Elem]
```

```
operaciones
  inserta: (Elem, Arbord[Elem]) → Arbord[Elem]                    /* gen */
  busca: (Elem, Arbord[Elem]) → Arbord[Elem]                      /* mod */
  borra: (Elem, Arbord[Elem]) → Arbord[Elem]                      /* mod */
  está: (Elem, Arbord[Elem]) → Bool                               /* obs */
operaciones privadas
  esOrdenado: Arbord[Elem] → Bool                                 /* obs */
  min: Arbord[Elem] – → Elem                                      /* obs */
  mayor, menor: (Arbord[Elem], Elem) → Bool                       /* obs */
ecuaciones
  ∀ x, y, z : Elem : ∀ a, iz, dr : Arbord[Elem] :
  def min(a) si NOT esVacío(a)
  min(Cons(iz, x, dr))          = x       si     esVacío(iz)
  min(Cons(iz, x, dr))          = min(iz) si NOT esVacío(iz)

  menor(Nuevo, y)               = cierto
  menor(Cons(iz, x, dr), y)     = x < y AND menor(iz, y) AND menor(dr, y)
  mayor(Nuevo, y)               = cierto
  mayor(Cons(iz, x, dr), y)     = x > y AND mayor(iz, y) AND mayor(dr, y)

  esOrdenado(Nuevo)             = cierto
  esOrdenado(Cons(iz, x, dr))   = esOrdenado(iz) AND menor(iz, x) AND
                                  esOrdenado(dr) AND mayor(dr, x)

  inserta(y, Nuevo)             = Cons(Nuevo, y, Nuevo)
  inserta(y, Cons(iz, x, dr))   = Cons(iz, x, dr)               si y == x
  inserta(y, Cons(iz, x, dr))   = Cons(inserta(y, iz), x, dr)   si y < x
  inserta(y, Cons(iz, x, dr))   = Cons(iz, x, inserta(y, dr))   si y > x

  busca(y, Nuevo)               = Nuevo
  busca(y, Cons(iz, x, dr))     = Cons(iz, x, dr)   si y == x
  busca(y, Cons(iz, x, dr))     = busca(y, iz)      si y < x
  busca(y, Cons(iz, x, dr))     = busca(y, dr)      si y > x

  borra(y, Nuevo)               = Nuevo
  borra(y, Cons(iz, x, dr))     = dr      si y == x AND esVacío(iz)
  borra(y, Cons(iz, x, dr))     = iz      si y == x AND esVacío(dr)
  borra(y, Cons(iz, x, dr))     = Cons(iz, z, borra(z, dr))
                                        si y == x AND NOT esVacío(iz) AND
                                           NOT esVacío(dr) AND z = min(dr)
  borra(y, Cons(iz, x, dr))     = Cons(borra(y, iz), x, dr)   si y < x
  borra(y, Cons(iz, x, dr))     = Cons(iz, x, borra(y, dr))   si y > x

  está(y, a)                    = NOT esVacío(busca(y, a))
errores
  min(Nuevo)
ftad
```

Complejidad de las operaciones

- Claramente, la complejidad de todas las operaciones está determinada por la complejidad de la búsqueda. El tiempo de una búsqueda en el caso peor es $O(t)$, siendo t la talla del árbol.
  - El caso peor se da en un árbol *degenerado* reducido a una sola rama, en cuyo caso la talla de un árbol de búsqueda con n nodos es n. Tales árboles pueden producirse a partir del árbol vacío por la inserción consecutiva de n elementos ordenados –en orden creciente o decreciente–.
  - Se puede demostrar que el promedio de las longitudes de los caminos en un árbol de búsqueda generado por la inserción de una sucesión aleatoria de n elementos con claves distintas es asintóticamente

    $$t_n = 2 \cdot (\ln n + \gamma + 1)$$

    siendo $\gamma = 0{,}577\ldots$ la constante de Euler, y suponiendo que las $n!$ permutaciones posibles de los nodos que se insertan son equiprobables.

- Si se quiere garantizar complejidad de $O(\log n)$ en el caso peor, es necesario restringirse a trabajar con alguna subclase de los árboles ordenados en la cual la talla se mantenga logarítmica con respecto al número de nodos.

5.4.2 Especificación de los árboles de búsqueda

- La diferencia entre los árboles de búsqueda y los árboles ordenados radica en las condiciones que se exigen a los elementos que, en el caso de los árboles de búsqueda, son una pareja (*clave*, *valor*).

Acceso por clave

- En muchas aplicaciones las comparaciones entre elementos se pueden establecer en base a una parte de la información total que almacenan dichos elementos: un campo *clave*. Por ejemplo, el DNI de una persona.

- En esta situación, lo que deberíamos exigirle a los elementos de un árbol de búsqueda es que dispusieran de una operación observadora cuyo resultado fuese de un tipo ordenado:

  `clave: Elem → Clave`

  Siendo *Clave* un tipo que pertenece a la clase ORD. Definiríamos así la clase de los tipos que tienen un operación de la forma *clave*, y especificaríamos e implementaríamos los árboles de búsqueda en términos de esta clase.

- Sin embargo, podemos generalizar aún más el planteamiento si consideramos que la clave y el valor son datos separados, de forma que en cada nodo del árbol se almacenen dos datos. Así, el TAD ARBUS tendrá dos parámetros: el TAD de las claves, que ha de ser ordenado, y el TAD de los valores.
  - El inconveniente de esta aproximación es que puede dar lugar a que se almacenen información repetida: la clave aparece en el nodo del árbol y como parte del valor asociado.

    Sin embargo, considerando que las claves suelen ser pequeñas, en comparación con los valores, e incluso se pueden compartir si son representadas con punteros, esto no constituye un problema grave.

- Para minimizar el problema de la compartición de estructura que las operaciones de los árboles, según las hemos especificado hasta ahora provocan, haremos que el acceso por clave devuelva directamente el valor asociado y no el subárbol que lo tiene como raíz.

  El único inconveniente de esta aproximación es que la *consulta* se convierte entonces en una operación parcial.

Combinación de elementos con la misma clave

- Al separar las claves y los valores, se nos plantea la cuestión de qué hacer cuando intentamos insertar un elemento asociado con una clave que ya está en el árbol. Dependiendo del tipo de los elementos tendrá sentido realizar distintas operaciones.

- Una posible solución es exigir que el tipo de los valores tenga una operación modificadora que *combine* elementos, con la signatura:

  $(\oplus) : (Elem, Elem) \rightarrow Elem$

  De forma que cuando insertemos un elemento asociado a una clave que ya exista, utilicemos esta operación para *combinar* los valores y obtener el nuevo dato que se debe almacenar en el árbol.

- Sin embargo, en muchas aplicaciones de los árboles de búsqueda la función de combinación es simplemente una proyección que conlleva la sustitución del valor antiguo por el nuevo.

  Es por ello, y para simplificar la especificación del TAD, que optamos directamente por este comportamiento, de forma que cuando insertamos un elemento asociado con una clave que ya está en el árbol, simplemente sustituimos el valor antiguo por el nuevo.

  Nótese que esta solución también permite la combinación de valores, aunque dejando la responsabilidad de realizarla a los clientes del árbol de búsqueda:
  - se obtiene el valor asociado con la clave,
  - se combina ese valor con el nuevo, y
  - se inserta el resultado de la combinación.

Especificación de los árboles de búsqueda

- La especificación de los árboles de búsqueda es muy similar a la de los árboles ordenados, pero sustituyendo los elementos por parejas de (*clave*, *valor*) y la operación *busca*, que devuelve el subárbol que tiene al elemento buscado como raíz, por la operación parcial *consulta*, que devuelve el valor asociado con la clave buscada. Además, ocultamos la operación *raíz*.

```
tad ARBUSCA[C :: ORD, V :: ANY]
 renombra C.Elem a Cla
          V.Elem a Val
 usa
   REC-PROF-ARBIN[PAREJA[C,V]]
     renombrando  inOrd a recorre
                  Arbin[Pareja[Cla,Val]] a Arbus[Cla, Val]
     ocultando    preOrd, postOrd, Cons, hijoIz, hijoDr, raíz
 tipo
   Arbus[Cla, Val]
 operaciones
   inserta: (Cla, Val, Arbus[Cla, Val]) → Arbus[Cla, Val]        /* gen */
   consulta: (Cla, Arbus[Cla, Val]) – → Val                      /* obs */
   borra: (Cla, Arbus[Cla, Val]) → Arbus[Cla, Val]               /* mod */
   está: (Cla, Arbus[Cla, Val]) → Bool                           /* obs */
 operaciones privadas
   esOrdenado: Arbus[Cla, Val] → Bool                            /* obs */
   min: Arbus[Cla, Val] – → Pareja[Cla, Val]                     /* obs */
   mayor, menor: (Arbus[Cla, Val], Cla) → Bool                   /* obs */
 ecuaciones
   ∀ c, c', d : Cla : ∀ x, x', y : Val : ∀ a, iz, dr : Arbus[Cla, Val] :
   def min(a) si NOT esVacío(a)
   min(Cons(iz,Par(c,x),dr))         = Par(c, x)    si     esVacío(iz)
   min(Cons(iz,Par(c,x),dr))         = min(iz)      si NOT esVacío(iz)
   menor(Nuevo, c')                  = cierto
   menor(Cons(iz,Par(c,x),dr), c')   = c < c' AND menor(iz,c') AND
                                       menor(dr, c')
   mayor(Nuevo, c')                  = cierto
   mayor(Cons(iz,Par(c,x),dr), c')   = c > c' AND mayor(iz,c') AND
                                       mayor(dr, c')
   esOrdenado(Nuevo)                 = cierto
   esOrdenado(Cons(iz,Par(c,x),dr))  = esOrdenado(iz) AND menor(iz, c) AND
                                       esOrdenado(dr) AND mayor(dr, c)
```

```
 inserta(c',x',Nuevo)                = Cons(Nuevo, Par(c', x'), Nuevo)
 inserta(c',x',Cons(iz,Par(c,x),dr)) = Cons(iz,Par(c, x'),dr)  si c' == c
 inserta(c',x',Cons(iz,Par(c,x),dr)) = Cons(inserta(c',x',iz),Par(c,x),dr)
                                                              si c' < c
 inserta(c',x',Cons(iz,Par(c,x),dr)) = Cons(iz,Par(c,x),inserta(c',x',dr))
                                                              si c' > c

 def consulta(c', a) si está(c', a)
 consulta(c', Cons(iz,Par(c,x),dr))  =  x                  si c' == c
 consulta(c', Cons(iz,Par(c,x),dr))  =f consulta(c', iz)   si c' < c
 consulta(c', Cons(iz,Par(c,x),dr))  =f consulta(c', dr)   si c' > c

 borra(c', Nuevo)                 = Nuevo
 borra(c', Cons(iz,Par(c,x),dr)) = dr      si c' == c AND esVacío(iz)
 borra(c', Cons(iz,Par(c,x),dr)) = iz      si c' == c AND esVacío(dr)
 borra(c', Cons(iz,Par(c,x),dr)) = Cons(iz, Par(d,y), borra(d, dr))
                                   si c' == c AND NOT esVacío(iz) AND
                                      NOT esVacío(dr) AND Par(d,y) = min(dr)
 borra(c', Cons(iz,Par(c,x),dr)) = Cons(borra(c',iz),Par(c,x),dr) si c' < c
 borra(c', Cons(iz,Par(c,x),dr)) = Cons(iz,Par(c,x),borra(c',dr)) si c' > c

 está(c', Nuevo)                  = falso
 está(c', Cons(iz,Par(c,x),dr))  = c == c' OR está(c', iz) OR está(c', dr)

errores
 min(Nuevo)
 consulta(c', a) si NOT está(c', a)
ftad
```

Implementación de los árboles de búsqueda

Tipo representante

```
template <class TClave, class TValor>
class TNodoArbus {
  private:
    TClave _clave;
    TValor _valor;
    TNodoArbus<TClave,TValor> *_iz, *_dr;
  ...
};

template <class TClave, class TValor>
class TArbus {
  ...
  private:
     TNodoArbus<TClave,TValor>* _ra;
  ...
};
```

Interfaz de la implementación

```
#include <iostream>
#include "secuencia_dinamica.h"
#include "excepciones.h"

using namespace std;

// Excepciones generadas por las operaciones de este TAD
// Acceso con una clave que no está en el árbol : EClaveErronea

// El tipo TClave debe implementar
//    operator==
//    operator<

template <class TClave, class TValor>
class TArbus;
```

```
template <class TClave, class TValor>
class TNodoArbus {
  private:
    TClave _clave;
    TValor _valor;
    TNodoArbus<TClave,TValor> *_iz, *_dr;
    TNodoArbus( const TClave&, const TValor&,
                TNodoArbus<TClave,TValor>*, TNodoArbus<TClave,TValor>* );
  public:
    const TClave& clave() const;
    const TValor& valor() const;
    TNodoArbus<TClave,TValor> * iz() const;
    TNodoArbus<TClave,TValor> * dr() const;
    friend TArbus<TClave,TValor>;
};
```

```
template <class TClave, class TValor>
class TArbus {
  public:

  // Constructoras, destructora y operador de asignación
     TArbus( );
     TArbus( const TArbus<TClave,TValor>& );
     ~TArbus( );
     TArbus<TClave,TValor>& operator=( const TArbus<TClave,TValor>& );

  // Operaciones de los árboles de búsqueda
     void inserta( const TClave&, const TValor& );
     // Pre : true
     // Post : inserta un par (clave, valor),
     //        si la clave ya está, se sustituye el valor antiguo

     void borra( const TClave& );
     // Pre : true
     // Post : elimina un par (clave, valor) a partir de una clave dada,
     //        si la clave no está, el árbol no se modifica

     // observadoras
     const TValor& consulta( const TClave& ) const throw (EClaveErronea);
     // Pre : esta( clave )
     // Post : devuelve el valor asociado con la clave dada
     // Lanza la excepción EClaveErronea si el árbol no contiene la clave
```

```
bool esta( const TClave& ) const;
// Pre : true
// Post : devuelve true|false según si el árbol contiene o no la clave

bool esVacio( ) const;
// Pre: true
// Post: Devuelve true | false según si el árbol está o no vacío

TSecuenciaDinamica<TValor> recorre( ) const;
// Pre : true
// Post : devuelve los valores del árbol, ordenados por clave

// Escritura
void escribe( ostream& salida ) const;

private:
// Variables privadas
     TNodoArbus<TClave,TValor>* _ra;

// Operaciones privadas

void libera();
static void TArbus<TClave,TValor>::liberaAux( TNodoArbus<TClave,TValor>* );

void copia( const TArbus<TClave,TValor>& );
static TNodoArbus<TClave,TValor>* copiaAux( TNodoArbus<TClave,TValor>* );

// operación privada de escritura
static void escribeAux( ostream&, TNodoArbus<TClave,TValor>* , string );

// operaciones auxiliares para los algoritmos recursivos
static void insertaAux( const TClave&, const TValor&,
                        TNodoArbus<TClave,TValor>* &  );

static TNodoArbus<TClave,TValor>* busca( const TClave&,
                                         TNodoArbus<TClave,TValor>* );

static void borraAux( const TClave&, TNodoArbus<TClave,TValor>* & );

static void borraRaiz( TNodoArbus<TClave,TValor>* & );

static void borraConMin( TNodoArbus<TClave,TValor>* &,
                         TNodoArbus<TClave,TValor>* & );
};
```

Implementación de las operaciones

- Sólo nos ocupamos de las operaciones que no tienen una equivalente en los árboles binarios. En todas ellas utilizamos una operación privada auxiliar que se encarga de recorrer recursivamente el árbol de nodos.

- La inserción.

```
template <class TClave, class TValor>
void TArbus<TClave,TValor>::inserta( const TClave& clave,
                                     const TValor& valor ) {
  insertaAux( clave, valor, _ra );
};

template <class TClave, class TValor>
void TArbus<TClave,TValor>::insertaAux( const TClave& clave,
                                        const TValor& valor,
                                        TNodoArbus<TClave,TValor>* & p) {
  if ( p == 0 )
    p = new TNodoArbus<TClave,TValor>( clave, valor );
  else if ( clave == p->clave() )
    p->_valor = valor;
  else if ( clave < p->clave( ) )
    insertaAux( clave, valor, p->_iz );
  else
    insertaAux( clave, valor, p->_dr );
};
```

- Las observadoras *consulta* y *esta* se apoyan en una operación privada auxiliar que busca una clave en el árbol y devuelve el puntero al nodo que la contiene , o 0 si no se encuentra en el árbol.

```
template <class TClave, class TValor>
const TValor& TArbus<TClave,TValor>::consulta( const TClave& clave ) const
  throw (EClaveErronea) {
  TNodoArbus<TClave,TValor>* aux;

  aux = busca(clave, _ra);
  if ( aux == 0 )
    throw EClaveErronea( );
  else
    return aux->valor();
};

template <class TClave, class TValor>
bool TArbus<TClave,TValor>::esta( const TClave& clave ) const {
  return busca(clave, _ra) != 0;
};

template <class TClave, class TValor>
TNodoArbus<TClave,TValor>*
TArbus<TClave,TValor>::busca( const TClave& clave,
                              TNodoArbus<TClave,TValor>* p )
{
  TNodoArbus<TClave,TValor>* r;

  if ( p == 0 )
    r = 0;
  else if ( clave == p->clave() )
    r = p;
  else if ( clave < p->clave() )
    r = busca(clave, p->iz());
  else if ( clave > p->clave() )
    r = busca(clave, p->dr());
  return r;
};
```

- El borrado.

```
template <class TClave, class TValor>
void TArbus<TClave,TValor>::borra( const TClave& clave ) {
  borraAux( clave, _ra );
};

template <class TClave, class TValor>
void TArbus<TClave,TValor>::borraAux( const TClave& clave,
                                      TNodoArbus<TClave,TValor>* & p ) {
  if ( p != 0 ) {
    if ( clave == p->clave( ) )
      borraRaiz(p);
    else if ( clave < p->clave( ) )
      borraAux( clave, p->_iz );
    else
      borraAux( clave, p->_dr );
  }
};
```

Procedimiento auxiliar que se encarga de borrar el nodo, una vez encontrado

```
template <class TClave, class TValor>
void TArbus<TClave,TValor>::borraRaiz( TNodoArbus<TClave,TValor>* & p ) {
  TNodoArbus<TClave,TValor>* aux;

  if ( p->iz() == 0 ) {
    aux = p;
    p = p->dr();
    delete aux;
  }
  else if ( p->dr() == 0 ) {
    aux = p;
    p = p->iz();
    delete aux;
  }
  else
    borraConMin( p, p->_dr );
};
```

Procedimiento auxiliar que se encarga de eliminar un nodo interno cuando ninguno de sus dos hijos es vacío. Dejando fijo el puntero *p* al nodo que deseamos eliminar, descendemos por los hijos izquierdos de su hijo derecho, con el parámetro *q*, hasta llegar al menor –el que no tiene hijo izquierdo–, y en ese punto realizamos el borrado

```
template <class TClave, class TValor>
void TArbus<TClave,TValor>::borraConMin( TNodoArbus<TClave,TValor>* & p,
                                         TNodoArbus<TClave,TValor>* & q ) {
  TNodoArbus<TClave,TValor>* aux;

  if ( q->iz() != 0 )
    borraConMin( p, q->_iz );
  else {
    p->_clave = q->clave();
    p->_valor = q->valor();
    aux = q;
    q = q->dr();
    delete aux;
  }
};
```

5.5 Colas de prioridad y montículos

5.5.1 Colas de prioridad

- La idea de cola de prioridad es semejante a la de cola, pero con la diferencia de que los elementos que entran en la cola van saliendo de ella para ser atendidos por *orden de prioridad* en lugar de por *orden de llegada*.
  - Vamos a exigir que los elementos de una cola de prioridad pertenezcan a la clase de tipos ordenados, de forma que la prioridad venga determinada por dicho orden. Así, si $x < y$ entenderemos que x tiene más prioridad que y.
  - Imponemos la restricción de que en una cola de prioridad no puede haber elementos repetidos –con la misma prioridad–.

- Las operaciones con las que equiparemos a este TAD serán las que nos permitan: crear una cola de prioridad vacía, añadir un elemento, consultar el elemento de mayor prioridad, eliminar el elemento de mayor prioridad, y averiguar si la cola es vacía.

  De esta forma, la especificación algebraica queda:

```
tad COLA-PRIO [E :: ORD]
 usa
   BOOL
 tipo
   CPrio[Elem]
 operaciones
   Nuevo: → CPrio[Elem]                                   /* gen */
   Añade: (Elem, CPrio[Elem]) – → CPrio[Elem]             /* gen */
   quitaMin: CPrio[Elem] – → CPrio[Elem]                  /* mod */
   min: CPrio[Elem] – → Elem                              /* obs */
   esVacio: CPrio[Elem] → Bool                            /* obs */
```

  Utilizamos una operación privada para determinar si es posible insertar un elemento en la cola de prioridad, i.e., si no se encuentra ya.

```
 operaciones privadas
   está: (Elem, CPrio[Elem]) → Bool                       /* obs */
```

Las ecuaciones quedan por tanto

```
 ecuaciones
   ∀ x, y : Elem : ∀ zs : CPrio[Elem] :
   está(x, Nuevo))                    = falso
   está(x, Añade(y, zs))              =d x == y OR está(x, zs)
   def Añade(x, zs) si NOT está(x, zs)
   Añade(y, Añade(x, zs))             =f Añade(x, Añade(y, zs))
   def quitaMin(zs) si NOT esVacio(zs)
   quitaMin(Añade(x, Nuevo))          = Nuevo
   quitaMin(Añade(x, Añade(y, zs)))   =f Añade(y, quitaMin(Añade(x, zs)))


 si x < y
   // El caso "y < x" también queda cubierto,
   // gracias a la conmutatividad de "Añade"
   def min(zs) si NOT esVacio(zs)
   min(Añade(x, Nuevo))               = x
   min(Añade(x, Añade(y, zs)))        =d min(Añade(x, zs))    si x < y
   // El caso "y < x" también queda cubierto,
   // gracias a la conmutatividad de "Añade"
   esVacio(Nuevo)                     = cierto
   esVacio(Añade(x, zs))              =d falso
 errores
   Añade(x, zs) si está(x, zs)
   quitaMin(Nuevo)
   min(Nuevo)
ftad
```

Implementaciones secuenciales

- La implementación de las colas de prioridad usando un tipo representante con estructura secuencial siempre resulta costosa para alguna de las operaciones. concretamente, si se utilizasen listas o listas ordenadas según las prioridades –con representación estática o enlazada– resultarían las siguientes complejidades –en el caso peor– para los tiempos de ejecución:

Operación	Lista	Lista ordenada
Nuevo	$O(1)$	$O(1)$
Añade	$O(1)$	$O(n)$
quitaMin	$O(n)$	$O(1)$
min	$O(n)$	$O(1)$
esVacio	$O(1)$	$O(1)$

- La inserción en la representación que utiliza una lista ordenada tiene complejidad $O(n)$
  - en una representación enlazada porque la búsqueda del lugar de inserción es secuencial, y
  - en un representación estática porque aunque la búsqueda puede ser binaria, $O(\log n)$, es necesario realizar un desplazamiento de los elementos de $O(n)$.

- Para una aplicación que efectúe todas las inserciones al principio, y todas las consultas y eliminaciones en una segunda fase, podrían realizarse n inserciones en tiempo $O(1)$ y a continuación ordenarse la lista, antes de la segunda fase, con un buen algoritmo de ordenación. Esto lograría un coste $O(n \cdot \log n)$ para el proceso de construcción de la cola de prioridad.

- Otra solución sería utilizar árboles de búsqueda con lo que conseguiríamos complejidad logarítmica para *Añade*, *min* y *quitaMin*.

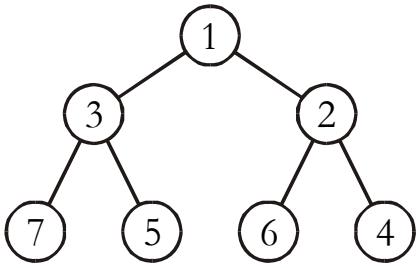
- Existe una solución aún mejor utilizando *montículos*, un tipo especial de árboles, que consigue en el caso peor $O(1)$ para *min* y complejidad $O(\log n)$ para *Añade* y *quitaMin*.

5.5.2 Montículos

- Se dice que un árbol binario *a* es un *montículo* –de mínimos– (en ingés, *heap*) si y sólo si
  - *a* es semicompleto,
  - la raíz de *a* –si existe– precede en prioridad a los elementos almacenados en los hijos, y
  - los hijos –si existen– son a su vez montículos.

  Algunos autores utilizan la terminología "árbol parcialmente ordenado y semicompleto" en lugar de montículo.

  Por ejemplo:



- Las operaciones con las que queremos equipar a los montículos son las que luego nos permitan implementar las operaciones de las colas de prioridad: construir un montículo vacío, consultar la raíz de un montículo, insertar un elemento y eliminar la raíz del montículo.

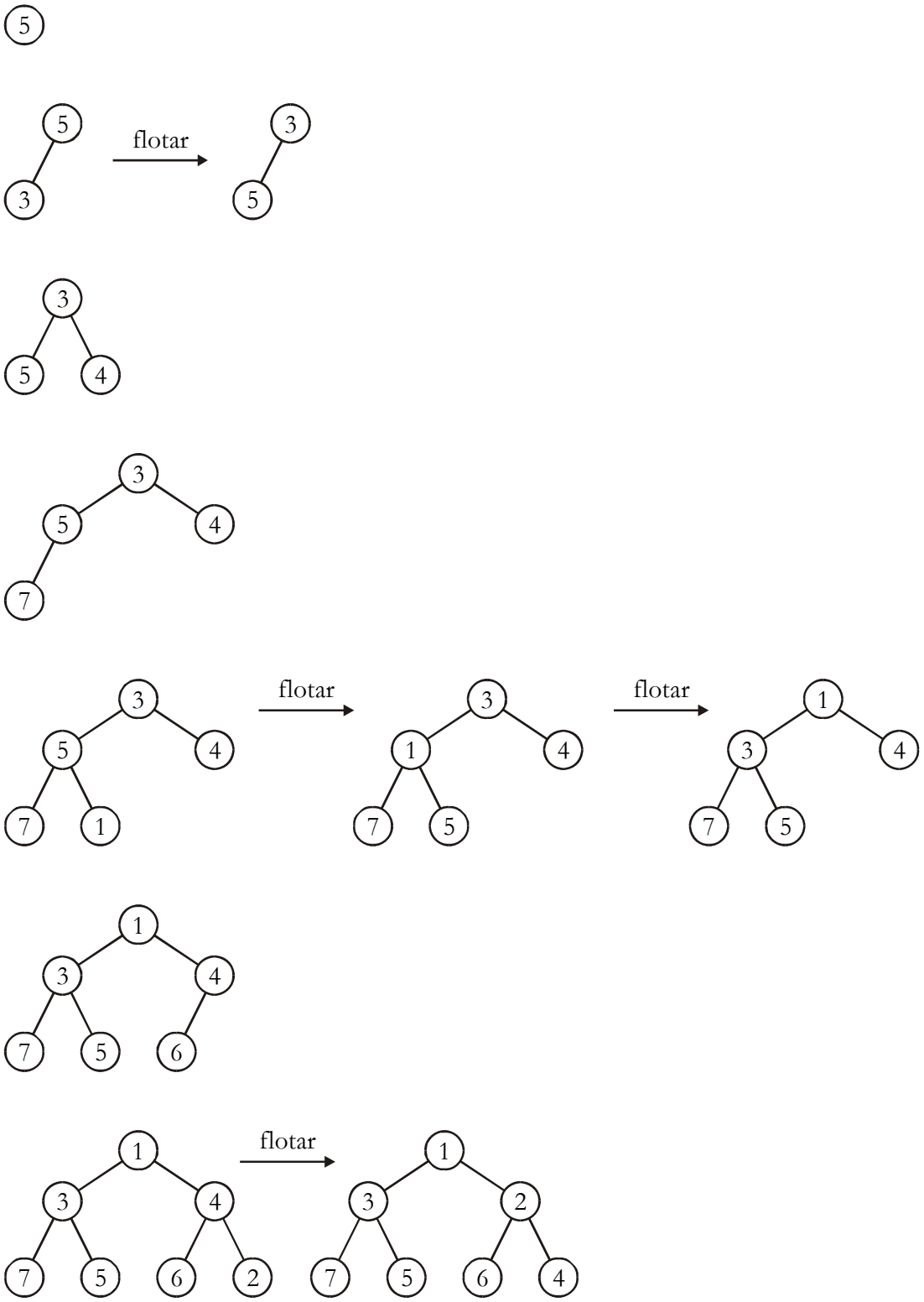
Inserción en un montículo

- Para *insertar* un nuevo elemento x en un montículo *a* se utiliza el siguiente algoritmo:

  (I.1) Se añade x como nueva hoja en la primera posición libre del último nivel. El resultado es un árbol semicompleto, si *a* lo era.

  (I.2) Se deja *flotar* x; i.e., mientras x no se encuentre en la raíz y sea menor que su padre, se intercambia x con su padre. El resultado es un montículo, suponiendo que *a* lo fuese y que x fuese diferente (en prioridad) de todos los elementos de *a*.

- Por ejemplo, veamos cómo se construye un montículo por inserciones sucesivas de 5, 3, 4, 7, 1, 6, 2



Obsérvese que el coste de una inserción en el caso peor será $O(\log n)$.

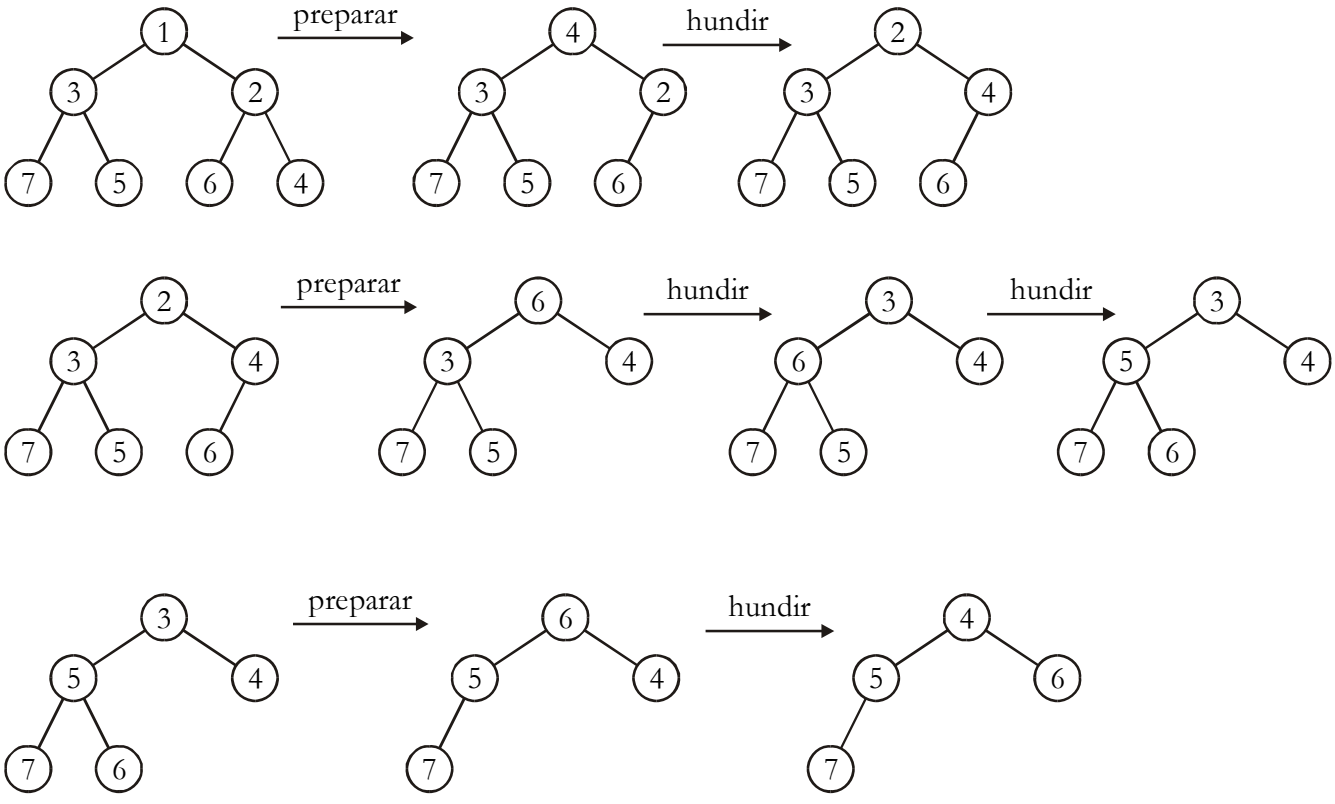
Eliminación del mínimo en un montículo

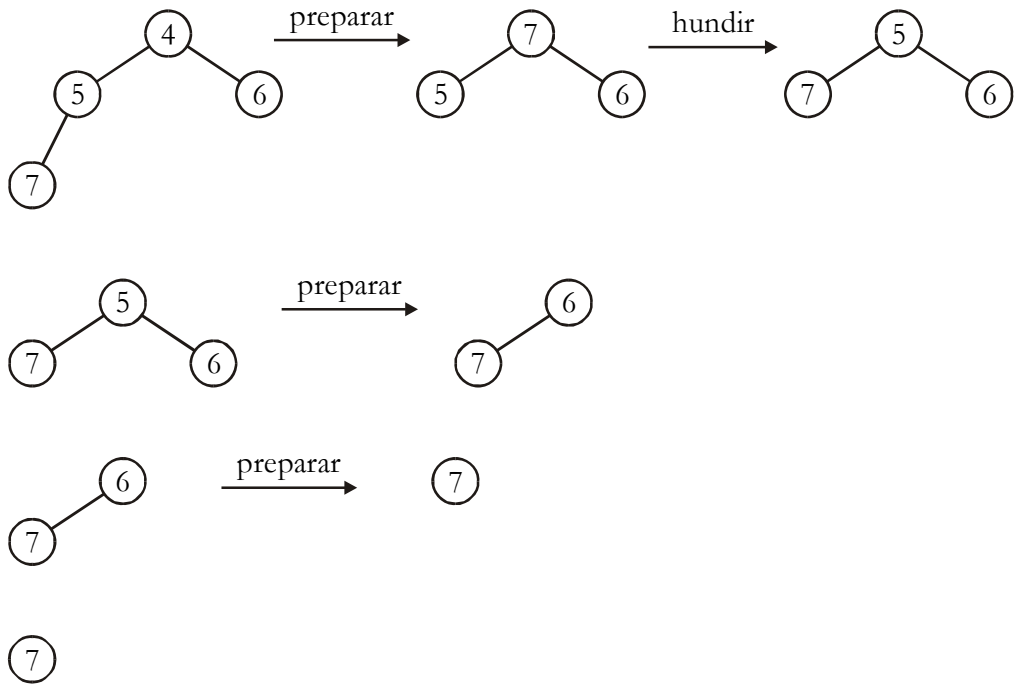
- Para *eliminar* el elemento de la raíz de un montículo *a*, se utiliza el siguiente algoritmo:

  (E.1) Si *a* es vacío, la operación no tiene sentido. Si *a* tiene un solo nodo, el resultado de la eliminación es el montículo vacío. Si *a* tiene dos o más nodos, se quita la última hoja y se pone el elemento *x* que ésta contenga en el lugar de la raíz, que queda eliminada. Esto da un árbol semicompleto, si *a* lo era. Se pasa a (E.2).

  (E.2) Se deja *hundirse x*; i.e., mientras *x* ocupe una posición con hijos y sea mayor que alguno de sus hijos, se intercambia *x* con el *hijo elegido*, que es aquel que sea menor que *x*, si hay un solo hijo que cumpla esa condición, o el hijo menor, si ambos son menores que *x*. El resultado es un montículo, suponiendo que *a* lo fuese.

- Como ejemplo, vamos a realizar eliminaciones sucesivas del mínimo a partir del montículo construido en el ejemplo anterior, hasta llegar al montículo vacío:





Obsérvese que el coste de una eliminación en el caso peor será $O(\log n)$.

- La serie de eliminaciones enumera los elementos que formaban el montículo inicial en orden creciente. Esta idea es la base del llamado *algoritmo de ordenación mediante montículo* (en inglés, *heapSort*).

- Se puede establecer una correspondencia directa entre las operaciones de los montículos y las de las colas de prioridad

Cola de prioridad	Montículo	T(*n*)
Nuevo	Nuevo	$O(1)$
Añade	inserta	$O(\log n)$
quitaMin	eliminaRaíz	$O(\log n)$
min	raíz	$O(1)$
esVacio	esVacio	$O(1)$

A continuación presentaremos una implementación estática de las colas de prioridad que utiliza un montículo como tipo representante.

Implementación de las operaciones auxiliares de borrado e inserción

- Tratamos estas dos operaciones –*flota* y *hunde*– por separado para luego utilizarlas en el algoritmo de ordenación mediante montículo.

- Un montículo se almacena en un array de la forma

```
const int max = 64 - 1; // (potencia de 2) – 1
TElem espacio[max];
```

Suponemos que el tipo *TElem* está dotado de igualdad y orden.

- Procedimiento para dejar flotar un elemento:

```
template <class TElem>
void flota( TElem v[], int i ) {
// Pre : v = V AND
//       dejando flotar v[i] se puede lograr que v[0..i] sea un montículo
  TElem x;
  int actual, padre;
  bool flotando;

  x = v[i];
  actual = i;
  flotando = true;
  while ( (actual > 0) && flotando ){
    padre = (actual - 1) / 2;
    if ( x < v[padre] ) {
      v[actual] = v[padre];
      actual = padre;
    }
   else
     flotando = false;
  }
  v[actual] = x;
// Post : v[0..i] es un montículo AND
//        v se ha obtenido a partir de V dejando flotar V[i]
};
```

Tiempo en el caso peor: $O(\log n)$

- Procedimiento para dejar hundirse un elemento

```
template <class TElem>
void hunde( TElem v[], int i, int j ) {
// Pre : v = V AND 0 <= i <= j AND
//       hundiendo v[i] se puede lograr que v[0..j] sea un montículo
  TElem x;
  int actual, hijo;
  bool hundiendo;

  x = v[i];
  actual = i;
  hundiendo = true;
  while ( (2*actual + 1 <= j) && hundiendo ) {
    hijo = hijoElegido(v, actual, j);
    if ( x > v[hijo] ) {
      v[actual] = v[hijo];
      actual = hijo;
    }
    else
      hundiendo = false;
  }
  v[actual] = x;
// Post : v[0..j] es un montículo AND
          v se ha obtenido a partir de V dejando hundirse V[i]
};
```

Tiempo en el caso peor: $O(\log n)$

Procedimiento auxiliar para elegir un hijo

```
template <class TElem>
int hijoElegido( TElem v[], int i, int j ) {
// Pre : 1 <= 2i+1 <= j
  int r;

  if ( 2*i + 1 == j )              // i sólo tiene hijo izquierdo
    r = 2*i + 1;
  else if ( v[2*i+1] < v[2*i+2] )  // i tiene dos hijos
    r = 2*i + 1;
  else
    r = 2*i + 2;
  return r;
// Post : hijoElegido es la posición del hijo menor de i
};
```

5.5.3 Implementación de las colas de prioridad

Tipo representante

- Representamos las colas de prioridad como montículos

```
template <class TElem>
class TCPrio
{
  public:

  // Tamaño inicial por defecto
     static const int MAX = 64 - 1;

  ...

  private:
  // Variables privadas
     int _capacidad;
     int _ult;
     TElem *_espacio;

  ...

};
```

Para evitar que el array donde almacenamos los datos se pueda llenar, utilizamos el puntero *_espacio* que apuntará a un array ubicado dinámicamente.

La variable *_capacidad* almacena en cada instante el tamaño del array ubicado. Cuando estando todas las posiciones del array ocupadas (*_ult* == *_capacidad*–1) se intenta insertar un nuevo elemento

- se ubica un array del doble de capacidad,
- se copia el array antiguo sobre el recién ubicado, y
- se anula el array antiguo.

Interfaz de la implementación

```
template <class TElem>
class TCPrio {
  public:
  // Tamaño inicial por defecto
     static const int MAX = 64 - 1;

  // Constructoras, destructora y operador de asignación
     TCPrio( int capacidad );
     TCPrio( const TCPrio<TElem>& );
     ~TCPrio( );
     TCPrio<TElem>& operator=( const TCPrio<TElem>& );

  // Operaciones de las colas
     void anyade(const TElem&);
     // Pre: true
     // Post: Se añade un elemento a la cola

     const TElem& min( ) const throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Devuelve el menor elemento (el más prioritario)
     // Lanza la excepción EAccesoIndebido si la cola está vacía

     void quitaMin( ) throw (EAccesoIndebido);
     // Pre: ! esVacio( )
     // Post: Elimina el menor elemento
     // Lanza la excepción EAccesoIndebido si la cola está vacía

     bool esVacio( ) const;
     // Pre: true
     // Post: Devuelve true | false según si la cola está o no vacía

  // Escritura
     void escribe( ostream& salida ) const;

  private:
  // Variables privadas
     int _capacidad;
     int _ult;
     TElem *_espacio;

  // Operaciones privadas
     void libera();
     void copia(const TCPrio<TElem>& pila);
};
```

Implementación de las operaciones

- Constructoras, destructora y operador de asignación

```
template <class TElem>
TCPrio<TElem>::TCPrio( int capacidad = MAX ) :
  _capacidad(capacidad), _ult(-1), _espacio(new TElem[_capacidad]) {
};

template <class TElem>
TCPrio<TElem>::TCPrio( const TCPrio<TElem>& cola ) {
   copia(cola);
};

template <class TElem>
TCPrio<TElem>::~TCPrio( ) {
  libera();
};

template <class TElem>
TCPrio<TElem>&
TCPrio<TElem>::operator=( const TCPrio<TElem>& cola ) {
  if( this != &cola ) {
    libera();
    copia(cola);
  }
  return *this;
};
```

- Operaciones de las colas

```
template <class TElem>
void TCPrio<TElem>::anyade(const TElem& elem) {
  if( _ult == _capacidad-1 ){
    _capacidad *= 2;
    TElem* nuevo = new TElem[_capacidad];
    for( int i = 0; i <= _ult; i++ )
      nuevo[i] = _espacio[i];
    delete [] _espacio;
    _espacio = nuevo;
  }
  _ult++;
  _espacio[_ult] = elem;
  flota(_espacio, _ult);
};
```

```
template <class TElem>
const TElem& TCPrio<TElem>::min( ) const throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido();
  else
    return _espacio[0];
};

template <class TElem>
void TCPrio<TElem>::quitaMin( ) throw (EAccesoIndebido) {
  if( esVacio() )
    throw EAccesoIndebido();
  else {
    _ult--;
    if ( _ult > -1 ) {
      _espacio[0] = _espacio[_ult+1];
      hunde(_espacio, 0, _ult);
    }
  }
};

template <class TElem>
bool TCPrio<TElem>::esVacio( ) const {
  return _ult == -1;
};
```

- Operaciones de entrada/salida

```
template <class TElem>
void TCPrio<TElem>::escribe( ostream& salida ) const {
  for( int i = 0; i <= _ult; i++ )
    salida << _espacio[i] << endl;
};
```

- Operaciones privadas

```
template <class TElem>
void TCPrio<TElem>::libera() {
  delete [] _espacio;
};

template <class TElem>
void TCPrio<TElem>::copia(const TCPrio<TElem>& cola) {
  _capacidad = cola._capacidad;
  _ult = cola._ult;
  _espacio = new TElem[ _capacidad ];
  for( int i = 0; i <= _ult; ++i )
    _espacio[i] = cola._espacio[i];
};
```

5.5.4 Algoritmos de ordenación con montículos

Ordenación con ayuda de una cola de prioridad

- La primera idea consiste en aprovecharnos de que los elementos de un montículo salen ordenados crecientemente. Por lo tanto, si creamos una cola de prioridad con los elementos del vector a ordenar y luego vamos sacando los elementos de la cola de prioridad, hasta dejarla vacía, e insertándolos en posiciones sucesivas del vector, el resultado final estará ordenado.

```
template <class TElem>
void ordenaCP( TElem v[], int n ) {

// Pre : max = capacidad del array AND
//       v = V AND 0 <= n <= max AND
//       ∀ i, j : 0 <= i < j < n : v[i] != v[j]

  TCPrio<TElem> xs;
  int i;

  for ( i = 0; i < n; i++ )
    xs.anyade( v[i] );
  for ( i = 0; i < n; i++ ) {
    v[i] = xs.min();
    xs.quitaMin();
  }

// Post : v[0..n-1] es una permutación de V[0..n-1] AND
//        v[n..max-1] = V[n..max-1] AND
//        v[0..n-1] ordenado en orden creciente

};
```

Tiempo en el caso peor: $O(n \cdot \log n)$.

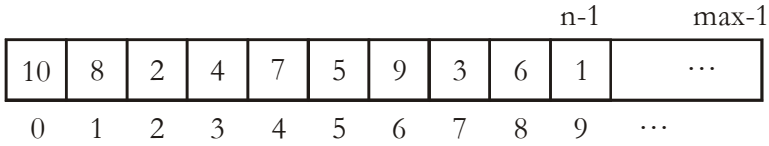
Ordenación sin necesidad de espacio adicional

- En el anterior procedimiento se necesita un espacio auxiliar de $O(n)$, que es la máxima dimensión que alcanza la cola de prioridad. En 1964, J.W.J. Williams y R.W. Floyd construyeron un algoritmo de ordenación de vectores basado en montículos, que no necesita espacio auxiliar. A cambio, el algoritmo no es modular, porque utiliza el espacio del mismo vector que se está ordenando para representar el montículo.

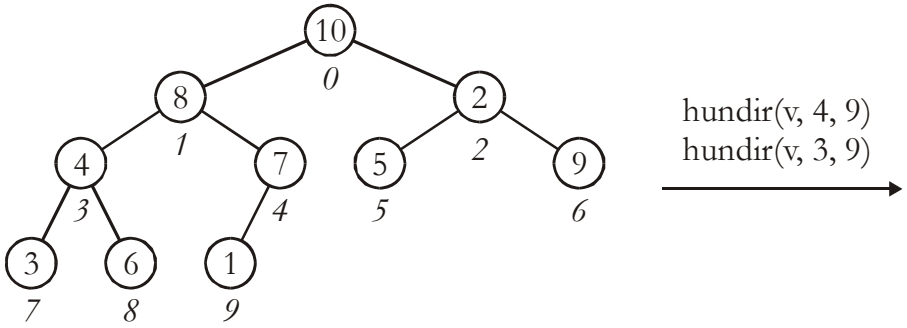
- La idea del algoritmo se compone de dos pasos:
  - Se reorganiza el vector de manera que pase a representar un montículo.
  - Mientras que el montículo no se quede vacío, se extrae el mínimo. Durante este proceso, habrá una parte del vector que ya está ordenada y que contiene los elementos menores, y otra, que representa un montículo, donde están los elementos sin ordenar.

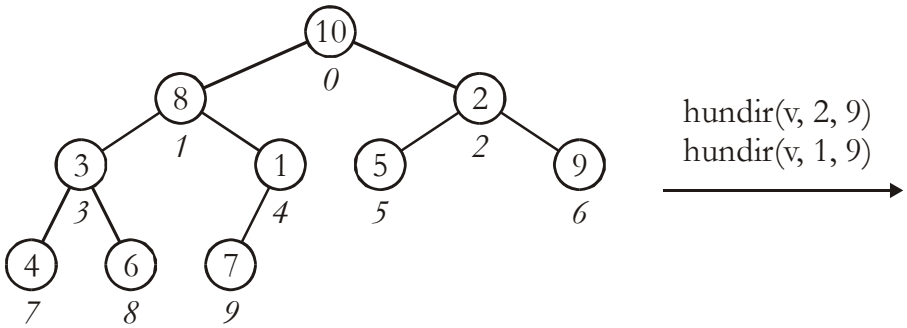
- El algoritmo que reorganiza un vector para que pase a representar un montículo se basa en las siguientes ideas:
  - Inicialmente el vector representa un árbol semicompleto.
  - Se van convirtiendo en montículos todos los subárboles, empezando por los más profundos. Si los dos subárboles de *a* son montículos, basta con hundir la raíz de *a* para que *a* también lo sea.
  - Las hojas son montículos.
  - Para que un nodo que está en la posición *i* tenga un hijo, se tiene que cumplir que $2i+1 \leq n-1$; por lo tanto, los nodos internos del árbol son los que ocupan las posiciones *v* [0 .. (*n*–2) *div* 2].

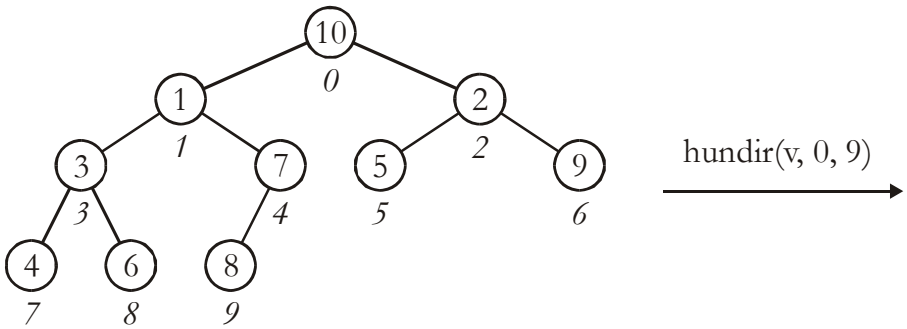
- Por ejemplo, queremos ordenar las 10 (n = 10) primeras posiciones del vector:

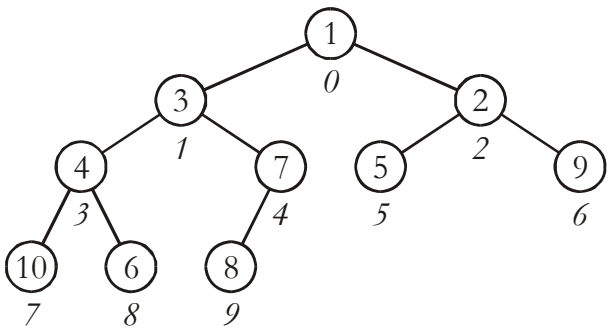


La conversión de v [0..9] en un montículo:









- Con todo esto, el procedimiento queda:

```
template <class TElem>
void hazMont( TElem v[], int n ) {

// Pre : max = capacidad del array AND
//       v = V AND 0 <= n <= max AND
//       ∀ i, j : 0 <= i < j < n : v[i] != v[j] }

  if ( n > 1 )
    for ( int i = (n - 2) / 2; i >= 0; i-- )
      hunde(v, i, n-1);

// Post : montículo(v[0..n-1]) AND v[n..max-1] = V[n..max-1] AND
//        v[0..n-1] es una permutación de V[0..n-1]

};
```

- El análisis de la complejidad de este procedimiento no es trivial. Vamos a suponer que lo hubiésemos implementado recursivamente, para así poder aplicar los resultados teóricos que conocemos para algoritmos recursivos. La formulación recursiva equivalente, distinguiendo casos según la talla t
  - $t = 1$. Caso directo. No se hace nada.
  - $t > 1$. Caso recursivo. Se reorganizan los dos hijos con llamadas recursivas, y luego se hunde la raíz.

  El tiempo $T(t)$ en función de la talla se comporta según la recurrencia:

$$T(t) = \begin{cases} c_0 & \text{si } t = 1 \\ 2 \cdot T(t-1) + c \cdot t & \text{si } t > 1 \end{cases}$$

  donde se tiene

  $a = 2 > 1$ llamadas
  $b = 1$ disminución del tamaño por sustracción

  cuya complejidad viene dada por:

  $O(a^{t \text{ div } b}) = O(2^t) = O(n)$

- Finalmente, el procedimiento de ordenación, que utiliza el procedimiento auxiliar *hazMont*, queda:

```
template <class TElem>
void ordena( TElem v[], int n ) {

// Pre : max = capacidad del array AND
//       v = V AND 0 <= n <= max AND
//       ∀ i, j : 0 <= i < j < n : v[i] != v[j]

  hazMont(v, n);
  if ( n > 1 )
    for ( int i = n-1; i > 0; i-- ) {
      TElem aux = v[i];
      v[i] = v[0];
      v[0] = aux;
      hunde(v, 0, i-1);
    }

// Post : v[0..n-1] es una permutación de V[0..n-1] AND
//        v[n..max-1] = V[n..max-1] AND
//        v[0..n-1] está ordenado en orden decreciente
};
```

La complejidad en tiempo es $O(n \cdot \log n)$, dada por n iteraciones de complejidad $O(\log n)$.