UNIVERSIDAD COMPLUTENSE MADRID

Tema 6: Diseño e implementación de TADs arborescentes

Miguel Gómez-Zamalloa
Estructuras de Datos y Algoritmos (EDA)
Grado en Desarrollo de Videojuegos
Facultad de Informática

Contexto

- Análisis de eficiencia
  1. Análisis de eficiencia de los algoritmos
- Algoritmos
  2. Divide y vencerás (DV), o *Divide-and-Conquer*
  3. Vuelta atrás (VA), o *backtracking*
- Estructuras de datos
  4. Diseño e implementación de TADs
  5. Tipos de datos lineales
  6. **Tipos de datos arborescentes**
  7. Diccionarios
  8. Aplicación de TADs

Contenidos

Tema 6: Diseño e implementación de TADs arborescentes

- Introducción
- Implementación de árboles binarios (versión sin compartición)
- Recorridos de árboles binarios
- Implementación eficiente de árboles binarios. Punteros inteligentes
- Árboles generales
- Árboles binarios de búsqueda y el TAD `set`

Introducción

Introducción

- En el capítulo anterior estudiamos distintos TADs lineales para representar datos organizados de manera secuencial.
- En este capítulo usaremos árboles para representar de manera intuitiva datos organizados en jerarquías.

Introducción

- Este tipo de estructuras jerárquicas surge de manera natural dentro y fuera de la Informática:
  - Organización de un documento en capítulos, secciones, etc.
  - Estructura de directorios y archivos de un sistema operativo.
  - Árboles de sintaxis de programas.

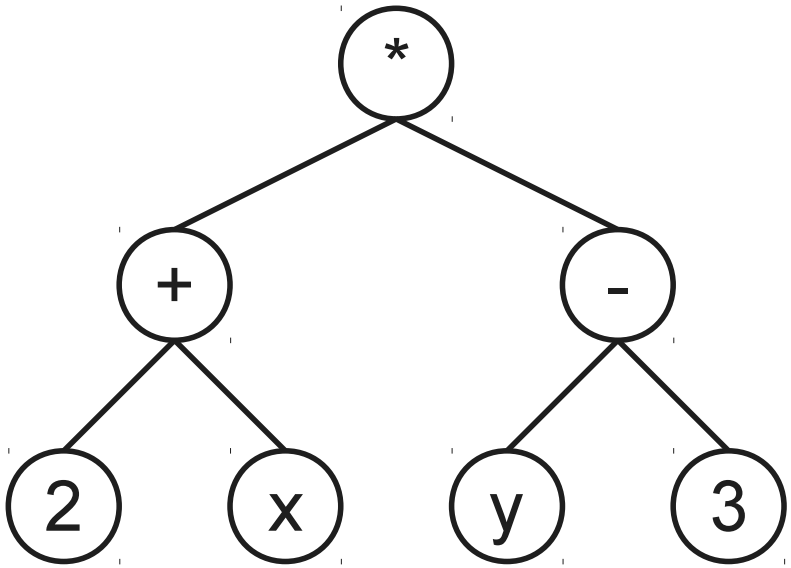


Figura: Aritmética con árboles

- Desde un punto de vista matemático, los árboles son estructuras jerárquicas formadas por *nodos*, que se construyen de manera inductiva:
  - Un solo nodo es un árbol a. El nodo es la *raíz* del árbol.
  - Dados *n* árboles $a_1, \ldots, a_n$, podemos construir un nuevo árbol a añadiendo un nuevo nodo como raíz y conectándolo con las raíces de los árboles a_i. Se dice que los a_i son *subárboles* de a.

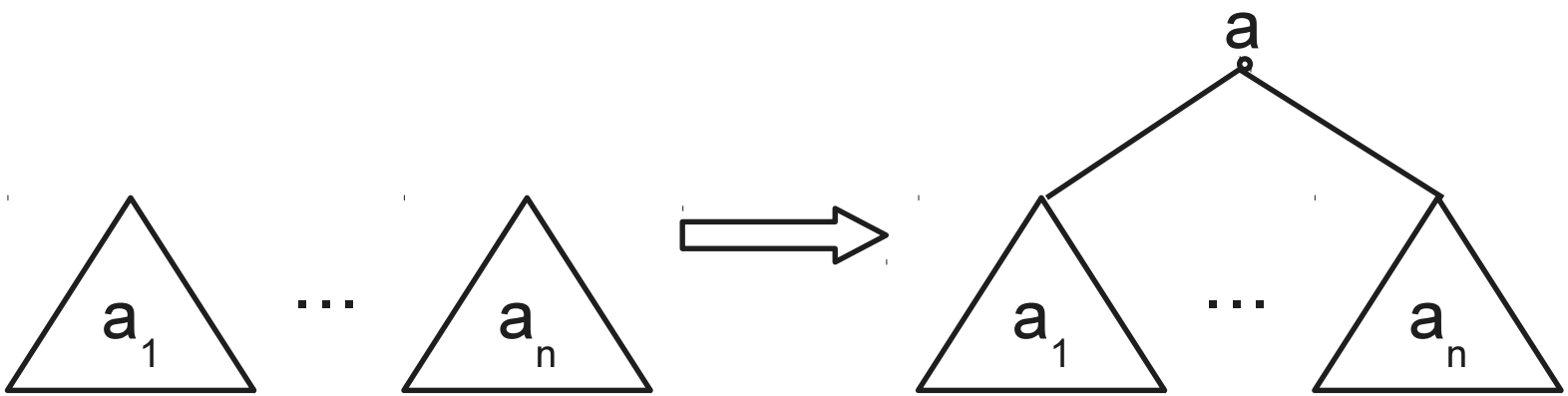


Figura: Construcción de un árbol

- Para identificar los distintos nodos de un árbol, vamos a usar una función que asigna a cada posición una cadena de números naturales con el siguiente criterio:
  - La raíz del árbol tiene como posición la *cadena vacía* ϵ.
  - Si un cierto nodo tiene como posición la cadena $\alpha \in \aleph^*$, el hijo i-ésimo de ese nodo tendrá como posición la cadena $\alpha.i$.
- Por ejemplo, la siguiente figura muestra un árbol y las cadenas que identifican las posiciones de sus nodos.

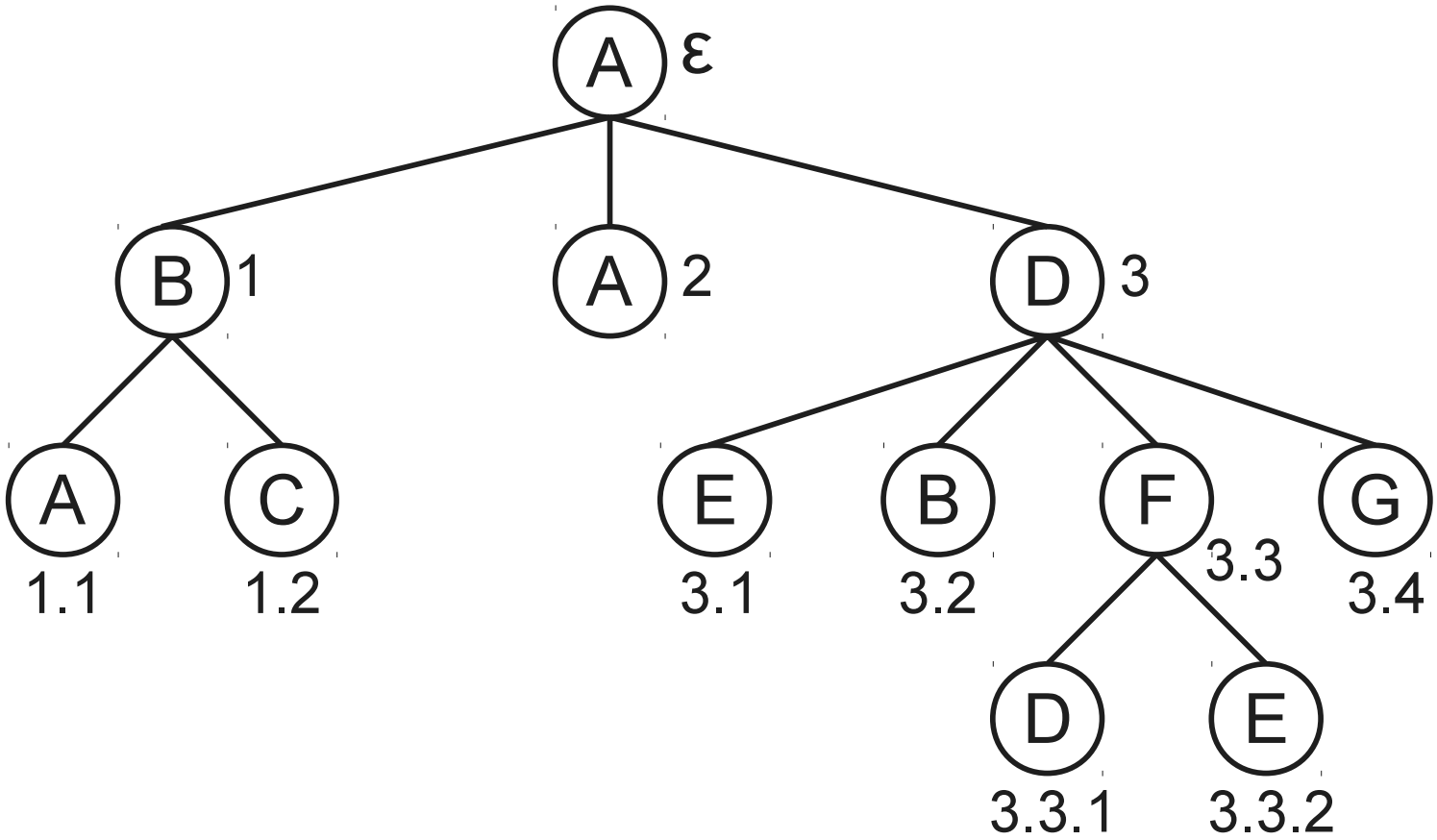


Figura: Modelo matemático de un árbol

- Un árbol puede describirse 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 asociados a los nodos.
- Podemos describir el árbol de la figura de la siguiente manera:

$$\begin{aligned} N &= \{\epsilon, 1, 2, 3, 1.1, 1.2, 3.1, 3.2, 3.3, 3.4, 3.3.1, 3.3.2\} \\ V &= \{A, B, C, D, E, F, G\} \end{aligned}$$

$$\begin{array}{lll} \mathrm{a}(\epsilon) = A & & \\ \mathrm{a}(1) = B & \mathrm{a}(2) = A & \mathrm{a}(3) = D \\ \mathrm{a}(1.1) = A & \mathrm{a}(1.2) = C & \mathrm{a}(3.1) = E \qquad \text{etc.} \end{array}$$

- Antes de seguir adelante, debemos establecer un vocabulario común que nos permita describir árboles. Dado un árbol a : $N \to V$
- Cada *nodo* es una tupla $(\alpha, \mathrm{a}(\alpha))$ que contiene la posición y el valor asociado al nodo. Distinguimos 3 tipos de nodos:
  - La *raíz* es el nodo de posición ϵ.
  - Las *hojas* son los nodos de posición α tales que no existe i tal que $\alpha.i \in N$
  - Los *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 $\alpha.i$ y $\alpha.j$ $(i \neq j)$ se llaman *hermanos*.

- Un *camino* es una sucesión de nodos $\alpha_1, \alpha_2, \ldots \alpha_n$ en la que cada nodo es padre del siguiente. El camino anterior tiene *longitud n*.
- Una *rama* es cualquier camino que empieza en la raíz y acaba en una hoja.
- El *nivel* o *profundidad* de un nodo es la longitud del camino que va desde la raíz hasta al nodo. En particular, el nivel de la raíz es 1.
- La *talla* o *altura* de un árbol es el máximo de los niveles de todos los nodos del árbol.
- 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.

- Decimos que α es *antepasado* de β (resp. β es *descenciente* de α) si existe un camino desde α hasta β.
- Cada nodo de un árbol a determina un *subárbol* a_0 con raíz en ese nodo.
- Dado un árbol a, los subárboles de a (si existen) se llaman *árboles hijos* de a.

- Distinguimos distintos tipos de árboles en función de sus características:
  - Ordenados o no ordenados. Un árbol es ordenado si el orden de los hijos de cada nodo es relevante.
  - Generales o n-ários. Un árbol es n-ário si el máximo numero de hijos de cualquier nodo es n. Un árbol es general si no existe una limitación fijada al número máximo de hijos de cada nodo.

Implementación de árboles binarios

Árboles binarios

- Un árbol binario consiste en una estructura recursiva cuyos nodos tienen como mucho dos hijos, un hijo izquierdo y un hijo derecho.
- El TAD de los árboles binarios (lo llamaremos *bintree*) tiene una serie de operaciones básicas con una utilidad muy limitada.
- En apartados siguientes lo extenderemos con otras operaciones.
- Las operaciones básicas son las siguientes:

Implementación de árboles binarios

- `Constructora sin argumentos`: Construye un árbol vacío.
- `Constructora un argumento`: Construye un árbol hoja con el dato recibido por parámetro.
- `Constructora tres argumentos`: Construye un árbol binario a partir de otros dos (el que será el hijo izquierdo y el hijo derecho) y el dato que se almacenará en la raíz.
- `root`: operación observadora que devuelve el elemento almacenado en la raíz del árbol. Es parcial pues falla si el árbol es vacío.
- `left`, `right`: dos operaciones observadoras (ambas parciales) que permiten obtener el hijo izquierdo y el hijo derecho de un árbol dado. Las operaciones no están definidas para árboles vacíos.
- `empty`: otra operación observadora para saber si un árbol tiene algún nodo o no.

Implementación de árboles binarios

- Igual que ocurre con los TADs lineales, podemos implementar el TAD *bintree* utilizando distintas estructuras en memoria.
- Cuando la forma de los árboles no está restringida la implementación más adecuada es la que utiliza nodos.
- Representación: Cada nodo del árbol será representado como un nodo en memoria que contendrá tres atributos: el elemento almacenado y punteros al hijo izquierdo y al hijo derecho.

Implementación de árboles binarios

- *No* debe confundirse un elemento del TAD *bintree* con la estructura en memoria utilizado para almacenarlo.
- Si bien existe una transformación directa entre uno y otro, son conceptos distintos.
- Un árbol es un elemento del TAD construido utilizando las operaciones generadoras anteriores (y que, cuando lo programemos, será un objeto de la clase `bintree`); los nodos en los que nos basamos forman una *estructura jerárquica de nodos* que tiene sentido únicamente *en la implementación*.

Implementación de árboles binarios

- Habrá métodos (privados o protegidos) que trabajan directamente con esta *estructura jerárquica* a la que el usuario del TAD no tendrá acceso directo (en otro caso se rompería la barrera de abstracción impuesta por las operaciones del TAD).
- A continuación aparece la definición de la clase `TreeNode` que, como no podría ser de otra manera, es una clase interna a `bintree`.
- La clase `bintree` necesita únicamente almacenar un *puntero* a la raíz de la *estructura jerárquica de nodos* que representan el árbol.

Implementación de árboles binarios

```
template <class T>
class bintree {
protected:
   /*
    Nodo que almacena internamente el elemento (de tipo T),
    y punteros al hijo izquierdo y derecho, que pueden ser
    nullptr si el hijo es vacío (no existe).
    */
   struct TreeNode;

   using Link = TreeNode*;

   struct TreeNode {
      TreeNode(Link const& l, T const& e, Link const& r) : elem(e), left(l), right(r) {};
      T elem;
      Link left, right;
   };

   // puntero a la raíz del árbol
   Link raiz;

   ...

};
```

Invariante de la representación

- El *invariante de la representación* de esta implementación debe asegurarse de que:
  - Todos los nodos contienen información válida y están ubicados correctamente en memoria.
  - El subárbol izquierdo y el subárbol derecho no comparten nodos.
  - No hay ciclos entre los nodos, o lo que es lo mismo, los nodos alcanzables desde al raíz no incluyen a la propia raíz.
- Con estas ideas el invariante queda:

$$R_{bintree_T}(\text{p})$$
$$\Longleftrightarrow_{def}$$
$$buenaJerarquia(\text{p.raiz})$$

Invariante de la representación

- donde

$$
\begin{aligned}
\mathit{buenaJerarquia}(\text{ptr}) &= \mathit{true} && \texttt{si} \quad \text{ptr} = \text{null} \\
\mathit{buenaJerarquia}(\text{ptr}) &= \text{ubicado}(\text{ptr}) \wedge R_T(\text{ptr.elem}) \wedge \\
&\quad \mathit{nodos}(\text{ptr.left}) \bigcap \mathit{nodos}(\text{ptr.right}) = \emptyset \wedge \\
&\quad \mathit{ptr} \notin \mathit{nodos}(\text{ptr.left}) \wedge \\
&\quad \mathit{ptr} \notin \mathit{nodos}(\text{ptr.right}) \wedge \\
&\quad \mathit{buenaJerarquia}(\text{ptr.left}) \wedge \\
&\quad \mathit{buenaJerarquia}(\text{ptr.right}) && \texttt{si} \quad \text{ptr} \neq \text{null}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{nodos}(\text{ptr}) &= \emptyset && \texttt{si} \quad \text{ptr} = \text{null} \\
\mathit{nodos}(\text{ptr}) &= \{\mathit{ptr}\} \bigcup \mathit{nodos}(\text{ptr.left}) \bigcup \mathit{nodos}(\text{ptr.right}) \\
& && \texttt{si} \quad \text{ptr} \neq \text{null}
\end{aligned}
$$

Relación de equivalencia

- La definición de la relación de equivalencia que se utiliza para saber si dos árboles son o no iguales también sigue una definición recursiva:

$$\text{p1} \equiv_{bintree_T} \text{p2}$$

$$\Longleftrightarrow_{def}$$

$$iguales_T(\text{p1.raiz}, \text{p2.raiz})$$

$$\begin{array}{lll}
iguales(\text{ptr1}, \text{ptr2}) = \text{true} & \texttt{si} & \text{ptr1} = \text{null} \wedge \text{ptr2} = \text{null} \\
iguales(\text{ptr1}, \text{ptr2}) = \text{false} & \texttt{si} & (\text{ptr1} = \text{null} \wedge \text{ptr2} \neq \text{null}) \vee \\
 & & (\text{ptr1} \neq \text{null} \wedge \text{ptr2} = \text{null}) \\
iguales(\text{ptr1}, \text{ptr2}) = \text{ptr1.elem} \equiv_T \text{ptr2.elem} \wedge & & \\
\quad iguales(\text{ptr1.left}, \text{ptr2.left}) \wedge & & \\
\quad iguales(\text{ptr1.right}, \text{ptr2.right}) & & \\
 & \texttt{si} & (\text{ptr1} \neq \text{null} \wedge \text{ptr2} \neq \text{null})
\end{array}$$

- Antes de seguir con la implementación de las operaciones debemos crear algunos métodos que trabajan directamente con la *estructura jerárquica*.
- Dada la naturaleza recursiva de la estructura de nodos, todas esas operaciones serán recursivas; recibirán, al menos, un puntero a un nodo que debe verse como la raíz de la estructura de nodos, o al menos la raíz del "subárbol" sobre el que debe operar.

Operaciones auxiliares

- Aunque pueda parecer empezar la casa por el tejado, comencemos con la operación que *libera* toda la memoria ocupada por la estructura jerárquica. El método recibe como parámetro el puntero al primer nodo y va eliminando de forma recursiva:

```
static void libera(Link ra) {
    if (ra != nullptr) {
        libera(ra->left);
        libera(ra->right);
        delete ra;
    }
}
```

Operaciones auxiliares

- Algunas de las operaciones pueden trabajar no con una sino con *dos* estructuras jerárquicas.
- Por ejemplo el siguiente método compara dos estructuras, dados los punteros a sus raices, para comprobar si son iguales y contienen los mismos elementos.

Operaciones auxiliares

```
static bool compara(Link r1, Link r2) {
    if (r1 == r2)
        return true;
    else if ((r1 == nullptr) || (r2 == nullptr))
        // En el if anterior nos aseguramos de
        // que r1 != r2. Si uno es nullptr, el
        // otro entonces no lo será, luego
        // son distintos.
        return false;
    else {
        return (r1->elem == r2->elem) &&
            compara(r1->left, r2->left) &&
            compara(r1->right, r2->right);
    }
}
```

Operaciones auxiliares

- Este método estático se utilizará para implementar el operador de igualdad del TAD bintree.
- Por último, algunas de las operaciones pueden devolver otra información.
- Por ejemplo, el siguiente método hace una *copia* de una estructura jerárquica y devuelve el puntero a la raíz de la copia.
- Se utilizaría en el constructor por copia y en el operador de asignación.
- Igual que todas las anteriores la implementación es recursiva.

Operaciones auxiliares

```
static Link copia(Link ra) {
    if (ra == nullptr)
        return nullptr;

    return new TreeNode(copia(ra->left),
                        ra->elem,
                        copia(ra->right));
}
```

- Los constructores para construir un árbol vacío y un árbol hoja quedarían así:

```
public:
    // árbol vacío
    bintree() : raiz(nullptr) {}

    // árbol hoja
    bintree(T const& e) : raiz(new TreeNode(nullptr, e, nullptr)) {}
```

- Sin embargo el constructor que recibe dos árboles ya construidos plantea una problemática importante:

```
bintree(const bintree& iz, const T& elem, const bintree& dr);
```

- Una implementación ingenua crearía un nuevo nodo y "cosería" los punteros haciendo que el hijo izquierdo del nuevo nodo fuera la raíz de `iz` y el derecho la raíz de `dr`:

```
// IMPLEMENTACIÓN NO VALIDA (POR EL MOMENTO...)
bintree(const bintree& iz, const T& e, const bintree& dr) :
    raiz(new TreeNode(iz.raiz, e, dr.raiz)) {}
```

Problema de compartición

- Esta implementación, no obstante, no es válida porque tendríamos una poco deseable compartición de memoria: la estructura jerárquica de los nodos de `iz` y de `dr` formarían también parte de la estructura jerárquica del nodo recién construido.
- Eso tiene como consecuencia inmediata que al destruir `iz` se destruirían automáticamente los nodos del árbol más grande.

Problema de compartición

- Para solucionar el problema hay tres alternativas:
  1. Hacer una copia de las estructuras jerárquicas de nodos de `iz` y `dr` (evitando así la compartición).
  2. Utilizar esas estructuras jerárquicas para el nuevo árbol y *vaciar* `iz` y `dr` de forma que la llamada al constructor los *vacíe*.
  3. Gestionar la compartición (con punteros inteligentes).
- En principio nos decantaremos por la primera opción (aunque más adelante buscaremos una solución mejor).

```
bintree(const bintree& iz, const T& e, const bintree& dr) :
  raiz(new TreeNode(copia(iz.raiz), e, copia(dr.raiz))) {}
```

Operaciones públicas

- Una copia similar hay que realizar en las operaciones `left` y `right` (para evitar el mismo problema de compartición) lo que automáticamente hace que el coste de estas operaciones sea $\mathcal{O}(n)$:

```
/**
 Devuelve un árbol copia del árbol izquierdo.
 Es una operación parcial (falla con el árbol vacío).
*/
bintree left() const {
    if (empty())
        throw std::domain_error("El arbol vacio no tiene hijo
     izquierdo.");

    return bintree<T>(copia(raiz->left));
}
```

Operaciones públicas

```
/**
 Devuelve un árbol copia del árbol derecho.
 Es una operación parcial (falla con el árbol vacío).
*/
bintree right() const {
    if (empty())
        throw std::domain_error("El arbol vacio no tiene hijo
     derecho.");

    return bintree<T>(copia(raiz->right));
}
```

Operaciones públicas

- La otra operación observadora, utilizada para acceder al elemento que hay en la raíz, no tiene necesidad de copias:

```
/**
 Devuelve el elemento almacenado en la raiz
*/
const T& root() const {
    if (empty())
        throw std::domain_error("El arbol vacio no tiene raiz
    .");
    return raiz->elem;
}
```

Operaciones públicas

- Otra operación observadora sencilla de implementar es `empty`:

```
/**
 Operación observadora que devuelve si el árbol
 es vacío (no contiene elementos) o no.
 */
bool empty() const {
    return raiz == nullptr;
}
```

Operaciones públicas

- La última operación observadora que implementaremos será el operador de igualdad. En este caso delegamos en el método estático que compara las estructuras jerárquicas de nodos:

```
/**
 Operación observadora que indica si dos bintree
 son equivalentes.
 */
bool operator==(const bintree &a) const {
    return compara(raiz, a.raiz);
}
```

Operaciones públicas

- Con esta terminamos la implementación de las operaciones del TAD.
- Existen otras operaciones observadoras que suelen ser habituales en la implementación de los árboles binarios que permiten conocer algunas de las propiedades definidas en la introducción.
- La implementación de la mayoría de ellas requiere la creación de métodos recursivos auxiliares que trabajen con la estructura jerárquica de los nodos.

Operaciones públicas

- Así, la complejidad de las operaciones del TAD *bintree* con esta estrategia queda como sigue:

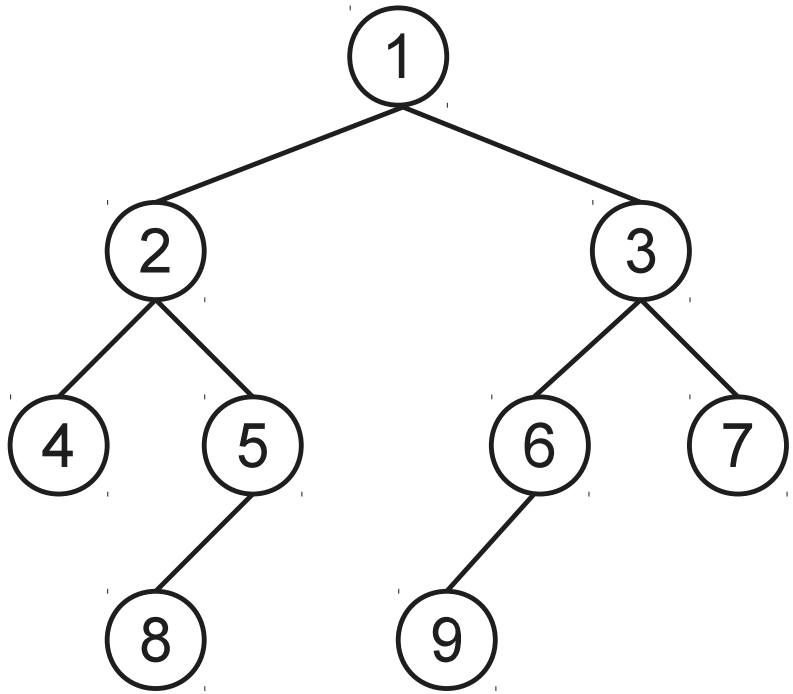
Operación	Complejidad
`Constructor sin argumentos`	$\mathcal{O}(1)$
`Constructor con 1 argumento`	$\mathcal{O}(1)$
`Constructor con 3 argumentos`	$\mathcal{O}(n)$
`left`	$\mathcal{O}(n)$
`right`	$\mathcal{O}(n)$
`root`	$\mathcal{O}(1)$
`empty`	$\mathcal{O}(1)$
`operator==`	$\mathcal{O}(n)$

Recorridos de árboles binarios

Recorridos de árboles binarios

- Además de las operaciones observadoras indicadas anteriormente podemos *enriquecer* el TAD `bintree` con una serie de operaciones que permitan recorrer todos los elementos del árbol.
- Si bien en el caso de las estructuras lineales el recorrido no ofrecía demasiadas posibilidades (solo había dos órdenes posibles, de principio al final o al contrario), en los árboles binarios hay distintas formas de recorrer el árbol.
- Los cuatro recorridos que veremos procederán de la misma forma: serán operaciones observadoras que devolverán un vector con los elementos almacenados en el árbol donde cada elemento aparecerá una única vez.
- El orden concreto en el que aparecerán dependerá del recorrido concreto (ver siguiente slide).

Recorridos de árboles binarios



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

Recorridos de árboles binarios

- Los tres primeros recorridos que consideraremos tienen definiciones recursivas:
  - Recorrido en *preorden*: se visita en primer lugar la *raíz* del árbol y, a continuación, se recorren en preorden el hijo izquierdo y el hijo derecho.
  - Recorrido en *inorden*: la raíz se visita tras el recorrido en inorden del hijo izquierdo y antes del recorrido en inorden del hijo derecho.
  - Recorrido en *postorden*: primero los recorridos en postorden del hijo izquierdo y derecho y al final la raíz.
- La definición de todos ellos es similar.
- Podemos hacer una implementación recursiva directa de la definición anterior si admitimos la existencia de una operación de concatenación de vectores.

Recorridos de árboles binarios

```
vector<T> bintree<T>::preorder() const {
  return preorder(raiz);
}

static vector<T> bintree<T>::preorder(Link p) {
  if (p == nullptr)
    return vector<T>(); // Vector vacío

  vector<T> ret;
  ret.push_back(ra->elem);
  concatena(ret, preorder(p->left));
  concatena(ret, preorder(p->right));

  return ret;
}
```

Recorridos de árboles binarios

- Pero la operación de concatenación no está disponible en *vector*.
- Podemos implementarla por ejemplo basándonos en el método `insert` de la clase `vector` de esta forma:

```
template <class T>
void concatena(vector<T>& v1, const vector<T>& v2){
    v1.insert(v1.end(), v2.begin(), v2.end());
}
```

- Tendría complejidad lineal y daría lugar a una complejidad del recorrido $\mathcal{O}(nlogn)$.
- Existe una implementación mejor que, siendo también recursiva, hace uso de un vector como parámetro de entrada/salida que va *acumulando* el resultado hasta que termina el recorrido.

Recorridos de árboles binarios

- El método `preorder` anterior se convierte en un método con dos parámetros: un puntero a la raíz de la estructura jerárquica de nodos que visitar, y un vector (no necesariamente vacío) al que se *añadirán* por la derecha los elementos del recorrido del resto del árbol.

Recorridos de árboles binarios

```
std::vector<T> preorder() const {
    std::vector<T> pre;
    preorder(raiz, pre);
    return pre;
}

static void preorder(Link a, std::vector<T> & pre) {
    if (a != nullptr) {
        pre.push_back(a->elem);
        preorder(a->left, pre);
        preorder(a->right, pre);
    }
}
```

Recorridos de árboles binarios

- La complejidad del recorrido es $\mathcal{O}(n)$, a lo que se puede llegar tras el análisis de recurrencias que utilizábamos en los algoritmos recursivos de los temas pasados.
- La misma complejidad tienen las implementaciones de los recorridos inorden y postorden que utilizan la misma idea.

Recorridos de árboles binarios

```
std::vector<T> inorder() const {
    std::vector<T> in;
    inorder(raiz, in);
    return in;
}

static void inorder(Link a, std::vector<T> & in) {
    if (a != nullptr) {
        inorder(a->left, in);
        in.push_back(a->elem);
        inorder(a->right, in);
    }
}
```

Recorridos de árboles binarios

```
std::vector<T> postorder() const {
    std::vector<T> post;
    postorder(raiz, post);
    return post;
}

static void postorder(Link a, std::vector<T> & post) {
    if (a != nullptr) {
        postorder(a->left, post);
        postorder(a->right, post);
        post.push_back(a->elem);
    }
}
```

Recorridos de árboles binarios

- El último tipo de recorrido que consideraremos es el recorrido *por niveles* (ver figura), que consiste en visitar primero la raíz, luego todos los nodos del nivel inmediatamente inferior de izquierda a derecha, a continuación todos los nodos del nivel tres, etc.
- La implementación no puede ser recursiva sino iterativa. Hace uso de una cola que contiene todos los subárboles que aún quedan por visitar.

Recorridos de árboles binarios

```
std::vector<T> levelorder() const {
    std::vector<T> levels;
    if (!empty()) {
        std::queue<Link> pendientes;
        pendientes.push(raiz);
        while (!pendientes.empty()) {
            Link sig = pendientes.front();
            pendientes.pop();
            levels.push_back(sig->elem);
            if (sig->left != nullptr)
                pendientes.push(sig->left);
            if (sig->right != nullptr)
                pendientes.push(sig->right);
        }
    }
    return levels;
}
```

Recorridos de árboles binarios

- La complejidad de este recorrido también es $\mathcal{O}(n)$ aunque para llegar a esa conclusión nos limitaremos, por una vez, a utilizar la intuición:
  - Cada una de las operaciones del bucle tienen coste constante, $\mathcal{O}(1)$.
  - Ese bucle se repite una vez por cada nodo del árbol; para eso basta darse cuenta que cada subárbol (o mejor, cada subestructura) aparece una única vez en la cabecera de la cola.

Implementación eficiente de los árboles binarios

Implementación eficiente de los árboles binarios

- El coste de las operaciones observadoras `left` y `right` de los árboles binarios es lineal, ya que devuelven una *copia* nueva de los árboles.
- Eso hace que una función aparentemente inocente como la siguiente, que cuenta el número de nodos, no tenga coste lineal (como uno esperaría).

```
template <typename E>
unsigned int numNodos(const bintree<E> &arbol) {
    if (arbol.empty())
        return 0;
    else
        return 1 +
               numNodos(arbol.left()) +
               numNodos(arbol.right());
}
```

Implementación eficiente de los árboles binarios

- La razón fundamental de esto es que no hemos permitido la *compartición* de la estructura jerárquica de nodos.
- El problema de eficiencia de las operaciones observadoras se podría solucionar permitiendo compartición.
- Desgraciadamente, esta solución de compartición de memoria no funciona en lenguajes como C++.
- Pensemos en el siguiente código, aparentemente inocente:

```
bintree<int> arbol;

// ...
// aquí construimos el árbol con varios nodos
// ...

bintree<int> otro;
otro = arbol.left();
```

Implementación eficiente de los árboles binarios

- Cuando el código anterior termina y las dos variables salen de ámbito, el compilador llamará a sus destructores.
- La destrucción de la variable `arbol` eliminará todos sus nodos; cuando el destructor de `otro` vaya a eliminarlos, se encontrará con que ya no existían, generando un error de ejecución.
- Por lo tanto, el principal problema es la destrucción de la estructura de memoria compartida.

Implementación eficiente de los árboles binarios

- En lenguajes como Java o C#, con recolección automática de basura, la solución anterior es perfectamente válida.
- En el caso de C++ hay que buscar otra aproximación.
- En concreto, se pueden traer ideas del mundo de la recolección de basura a nuestra implementación de los árboles.

Implementación eficiente de los árboles binarios

- Podemos utilizar lo que se llama *conteo de referencias*: cada nodo de la estructura jerárquica de nodos mantiene un contador (entero) que indica *cuántos punteros lo referencian*.
- Así, si tengo un único árbol, todos sus nodos tendrán un 1 en ese contador.
- La operación `left` construye un nuevo árbol cuya raíz apunta al nodo del hijo izquierdo, por lo que su contador *se incrementa*.
- Cuando se invoca al destructor del árbol, se decrementa el contador y si llega a cero se elimina él y recursivamente todos los hijos.
- La implementación de la clase TreeNode con el contador queda así (aparecen subrayados los cambios):

Implementación eficiente de los árboles binarios

```
class TreeNode {
public:
  TreeNode() : left(nullptr), right(nullptr), numRefs(0) {}
  TreeNode(Link iz, const T &elem, Link dr) :
    elem(elem), left(iz), right(dr), numRefs(0) {

    if (left != nullptr) left->addRef();
    if (right != nullptr) right->addRef();
  }

  void addRef() { assert(numRefs >= 0); numRefs++; }
  void remRef() { assert(numRefs > 0); numRefs--; }

  T elem;
  Link left;
  Link right;

  int numRefs;
};
```

Implementación eficiente de los árboles binarios

- El método `left()` ya no necesita copiar la estructura jerárquica de nodos; el constructor especial que recibe el puntero al nodo raíz simplemente incrementa el contador de referencias:

```
class bintree {
public:
    ...
    bintree left() const {
        if (empty()) throw throw std::domain_error("...");
        return bintree(~~copia~~(raiz->left~~)~~);
    }

private:
    ...
    bintree(Link raiz) : raiz(raiz) {
        if (raiz != nullptr)
            raiz->addRef();
    }
}
```

Implementación eficiente de los árboles binarios

- Tampoco se necesita la copia en la construcción de un árbol nuevo a partir de los dos hijos, pues el árbol grande compartirá la estructura.
- El constructor crea el nuevo `TreeNode` (cuyo constructor incrementará los contadores de los nodos izquierdo y derecho) y pone el contador del nodo recién creado a uno:

```
bintree(const bintree &iz, const T &e, const bintree &dr) :
    raiz(new TreeNode(~~copia~~(iz.raiz), e, ~~copia~~(dr.raiz))) {
    raiz->addRef();
}
```

Implementación eficiente de los árboles binarios

- Por último, la liberación de la estructura jerárquica de nodos sólo se realiza si nadie más referencia el nodo.
- Por lo tanto el método de liberación recursivo cambia.

```
static void libera(Link ra) {
    if (ra != nullptr) {
        ra->remRef();
        if (ra->_numRefs == 0) {
            libera(ra->left);
            libera(ra->right);
            delete ra;
        }
    }
}
```

Implementación eficiente de los árboles binarios

- Gracias a estas modificaciones la complejidad de todas las operaciones pasa a ser constante, y el consumo de memoria se reduce y no desperdiciamos nodos con información repetida.

Operación	Complejidad
`Constructor sin argumentos`	$\mathcal{O}(1)$
`Constructor con argumentos`	$\mathcal{O}(1)$
`left`	$\mathcal{O}(1)$
`right`	$\mathcal{O}(1)$
`raiz`	$\mathcal{O}(1)$
`empty`	$\mathcal{O}(1)$

Implementación eficiente de los árboles binarios

- Ahora que disponemos de operaciones eficientes, podríamos implementar todos los recorridos que vimos en el apartado anterior (preorden, inorden, etc) como funciones externas al TAD bintree, con complejidad lineal.
- Ten en cuenta que ahora podemos navegar por su estructura usando los métodos públicos `left` e `right` sin necesidad de hacer copias.

Implementación con Punteros Inteligentes

Implementación con Punteros Inteligentes

- La técnica del conteo de referencias utilizada en la sección previa es muy común y anterior a la existencia de recolectores de basura.
- En realidad, los primeros recolectores de basura se implementaron utilizando esta técnica, por lo que podríamos incluso decir que lo que hemos implementado aquí es un pequeño recolector de basura para los nodos de los árboles.
- Sin embargo, en un desarrollo más grande, el manejo explícito de los contadores invocando al `addRef` y `remRef` es incómodo y propenso a errores, pues es fácil olvidar llamarlas (un olvido de un *remRef* provocaría fugas de memoria).

Implementación con Punteros Inteligentes

- Para evitar este tipo de problemas se han implementado estrategias que gestionan de forma automática esos contadores.
- En particular, son muy utilizados lo que se conoce como *punteros inteligentes* (*smart pointers*), que son variables que se comportan igual que un puntero pero que, además, cuando cambian de valor incrementan/decrementan el contador del dato al que comienzan/dejan de apuntar.
- A continuación mostramos la implementación de la parte básica de la clase `bintree` haciendo uso de punteros inteligentes.
  - Usaremos un puntero inteligente de tipo `shared_ptr` (puntero compartido) en la definición del tipo `Link`, y
  - haremos uso de la función `make_shared` cuando haya que construirlo en las constructoras de `bintree`.

Implementación con Punteros Inteligentes

```
template <class T>
class bintree {
protected:

   // Puntero inteligente en la definición del tipo Link
   using Link = std::shared_ptr<TreeNode>;
   struct TreeNode {
      TreeNode(Link const& l, T const& e, Link const& r)
                           : elem(e), left(l), right(r) {};
      T elem;
      Link left, right;
   };

   // puntero a la raíz del árbol
   Link raiz;

   // constructora privada
   bintree(Link const& r) : raiz(r) {}
```

Implementación con Punteros Inteligentes

```
public:
    // árbol vacío
    bintree() : raiz(nullptr) {}

    // árbol hoja
    bintree(T const& e) : // make_shared para crear nodos
        raiz(std::make_shared<TreeNode>(nullptr, e, nullptr))
    {}

    // árbol con dos hijos
    bintree(bintree<T> const& l, T const& e,
            bintree<T> const& r) :
        // Uso de make_shared para la creación de nodos
        raiz(std::make_shared<TreeNode>(l.raiz, e, r.raiz)) {}

    // consultar si el árbol está vacío
    bool empty() const {
        return raiz == nullptr;
    }
```

Implementación con Punteros Inteligentes

```
    // consultar la raíz
    T const& root() const {
        if (empty()) throw std::domain_error("...");
        else return raiz->elem;
    }

    // consultar el hijo izquierdo
    bintree<T> left() const {
        if (empty()) throw std::domain_error("...");
        else return bintree<T>(raiz->left);
    }

    // consultar el hijo derecho
    bintree<T> right() const {
        if (empty()) throw std::domain_error("...");
        else return bintree(raiz->right);
     }

    ...
};
```

Árboles generales

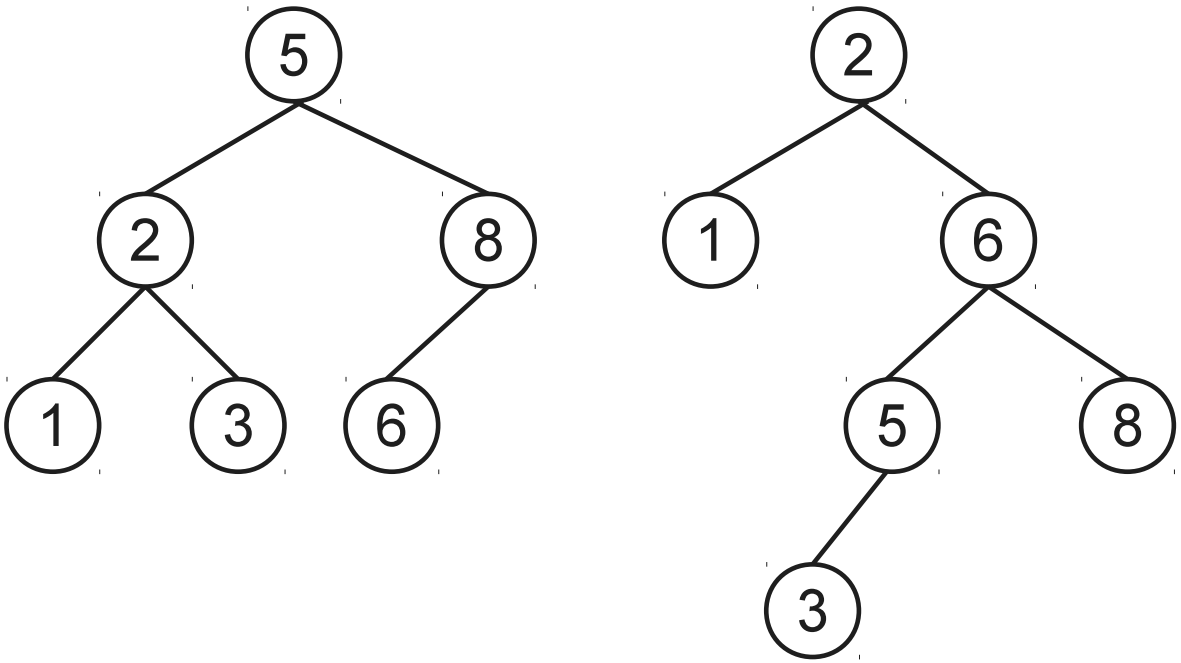
Árboles generales

- Los árboles generales no imponen una limitación *a priori* sobre el número de hijos que puede tener cada nodo.
- Por tanto, para implementarlos debemos buscar mecanismos generales que nos permitan almacenar un número de hijos variable en cada nodo.
- Dos posibles soluciones:
  - Cada nodo contiene una lista de punteros a sus nodos hijos. Si el número de hijos se conoce en el momento de la creación del nodo y no se permite añadir hijos a un nodo ya creado, en lugar de una lista podemos utilizar un array.
  - Cada nodo contiene un puntero al primer hijo y otro puntero al hermano derecho. De esa forma, podemos acceder al hijo i-ésimo de un nodo accediendo al primer hijo y luego recorriendo $i - 1$ punteros al hermano derecho.

Árboles Binarios de Búsqueda

Árboles Binarios de Búsqueda

- Un árbol binario de búsqueda (BST) es una estructura de datos de tipo árbol binario con la siguiente propiedad recursiva:
  - La raiz es mayor que todos los nodos del hijo izquierdo.
  - La raiz es menor que todos los nodos del hijo derecho.
  - El hijo izquierdo y el derecho cumplen también la propiedad recursivamente, es decir, son también BST.
- Ejemplo de dos BSTs equivalentes (tienen los mismos datos):



Árboles Binarios de Búsqueda

Aplicación:

- Permite implementar colecciones de datos con búsquedas, inserciones y borrados en $\mathcal{O}(log(n))$.
- Además su recorrido inorden sería un recorrido ordenado.
- Servirá por tanto como representación interna para los TADs `set` y `map` (este último se verá en el tema 7).

El TAD set usando un BST

```
template <class T, class Comp = std::less<T>>
class set {
  /*
  Nodo que almacena el elemento (de tipo T) y punteros al hijo
  izquierdo y derecho (nullptr si el hijo es vacío).
  */
  struct TreeNode {
    T elem;
    Link iz, dr;
    TreeNode(T const& e, Link i = nullptr, Link d = nullptr) :
             elem(e), iz(i), dr(d) {}
  };

  using Link = TreeNode*; // alias por claridad

  Link raiz; // puntero a la raíz del árbol

  int nelems; // número de elementos (cardinal del conjunto)

  // objeto función que compara elementos (orden total estricto)
  Comp menor;
```

El TAD set usando un BST

```
public:

// constructor (conjunto vacío)
set(Comp c = Comp()) : raiz(nullptr), nelems(0), menor(c) {}

// consulta si el árbol es vacío. O(1)
bool empty() const {
  return raiz == nullptr;
}

// devuelve cardinalidad del conjunto. O(1)
int size() const {
  return nelems;
}
```

- La contrucción por copia, asignación y eliminación sería igual que en la clase `bintree`.

El TAD set usando un BST- Búsqueda

- La pertenencia se implementa mediante la operación `count` (como en la STL) que devuelve 0 o 1.

```
public:
  // Consulta si e pertenece al conjunto
  int count(T const& e) const {
    return pertenece(e, raiz) ? 1 : 0;
  }

protected:
  bool pertenece(T const& e, Link a) const {
    if (a == nullptr) return false;
    else if (menor(e, a->elem)) {
      return pertenece(e, a->iz);
    }
    else if (menor(a->elem, e)) {
      return pertenece(e, a->dr);
    }
    else { // e == a->elem
      return true;
    }
  }
```

El TAD set usando un BST- Inserción

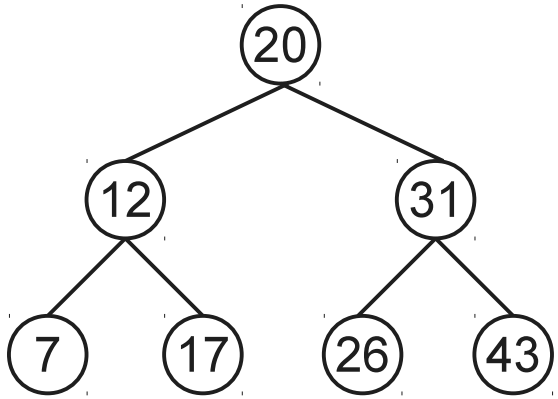
```
public:
  // inserta e en el conjunto (devuelve false si ya estaba)
  bool insert(T const& e) {
    return inserta(e, raiz);
  }

protected:
  bool inserta(T const& e, Link& a) {
    if (a == nullptr) {
      a = new TreeNode(e);
      ++nelems;
      return true;
    }
    else if (menor(e, a->elem)) {
      return inserta(e, a->iz);
    }
    else if (menor(a->elem, e)) {
      return inserta(e, a->dr);
    }
    else // el elemento e ya está en el árbol
      return false;
  }
```

El TAD set usando un BST- Eliminación

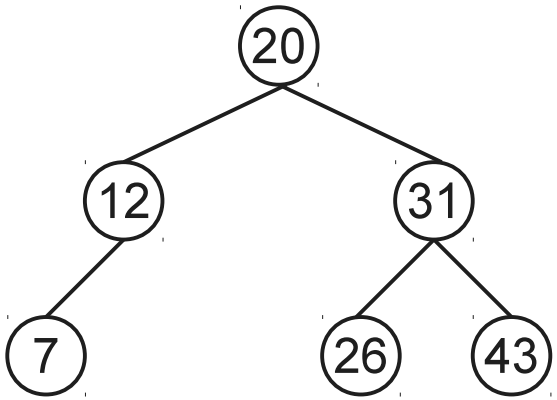
```
public: // elimina e del conjunto
  bool erase(T const& e) { return borra(e, raiz); }

protected:
  bool borra(T const& e, Link& a)  {
    if (a == nullptr) return false;
    else {
      if (menor(e, a->elem)) return borra(e, a->iz);
      else if (menor(a->elem, e)) return borra(e, a->dr);
      else { // e == a->elem
        if (a->iz == nullptr || a->dr == nullptr) {
          Link aux = a;
          a = (a->iz == nullptr) ? a->dr : a->iz;
          --nelems;
          delete aux;
        }
        else { // tiene dos hijos
          subirMenorHD(a);
          --nelems;
        }
        return true;
      }
    }
  }
```

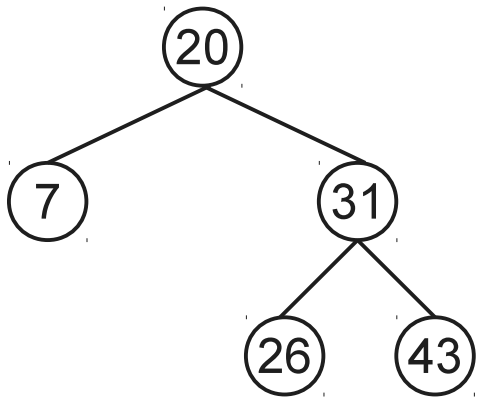
El TAD set usando un BST- Eliminación

- Ejemplos de eliminación: Sea el siguiente árbol



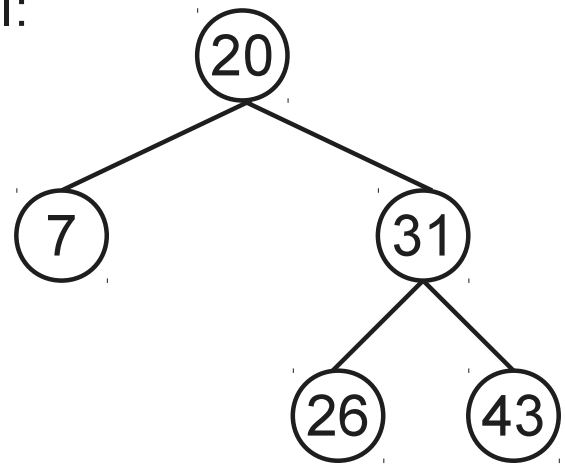
- si hacemos `a.erase(17)` y `a.erase(12)` tendríamos resp.



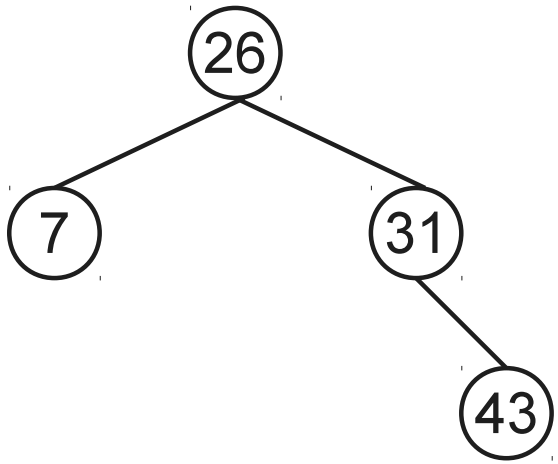


El TAD set usando un BST- Eliminación

- El caso difícil es cuando el nodo a eliminar tiene los dos hijos, por ej. el nodo 20 de este árbol:



- La estrategia va a ser sustituirle por un nodo (sin dos hijos) que pueda ocupar su lugar $\Rightarrow$ El menor de su hijo derecho



El TAD set usando un BST- Eliminación

- Esta función busca el menor del hijo derecho de a y lo coloca en su lugar.
- El menor del hijo derecho se puede quitar fácilmente del sitio que ocupa pues no puede tener hijo izquierdo.

```
void subirMenorHD(Link& a) {
  Link b = a->dr, padre = nullptr;
  while (b->iz != nullptr) {
    padre = b;
    b = b->iz;
  }
  if (padre != nullptr) {
    padre->iz = b->dr;
    b->dr = a->dr;
  }
  b->iz = a->iz;
  delete a;
  a = b;
}
```

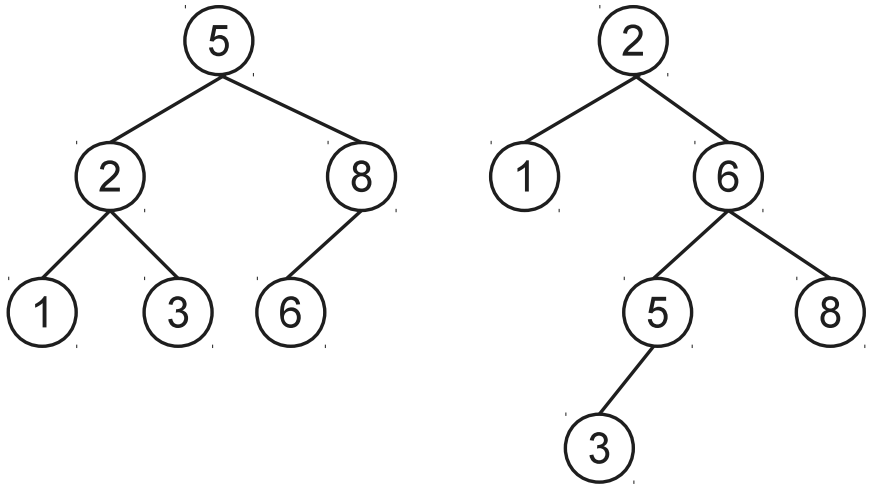
El TAD set usando un BST- Análisis de complejidad

- Las tres operaciones importantes, pertenencia, inserción y eliminación, tendrían complejidad lineal en la profundidad del árbol.
- Si el árbol está bien *equilibrado* (e.d., no hay mucha diferencia entre unas ramas y otras) esto sería $\mathcal{O}(log(n))$ con $n =$ nº de elementos.
- Si el árbol queda muy desequilibrado (por ejemplo si se insertan todos sus elementos en orden creciente) entonces la complejidad sería $\mathcal{O}(n)$.
- Hay técnicas para asegurar que los árboles binarios se mantienen equilibrados $\Rightarrow$ *árboles AVL* o *árboles rojo-negros* (ver enlaces).
- El tipo `set` de la STL asegura árboles equilibrados, por tanto:

Los métodos `count`, `insert` y `erase` del TAD `set` son $\mathcal{O}(log(n))$

El TAD set usando un BST- Recorrido con iterador

El recorrido inorden de un BST es un recorrido ordenado creciente!



- Implementaremos por tanto un iterador que haga el recorrido inorden.
- Se podría generar el recorrido en un vector y dejarlo guardado en el iterador $\Rightarrow$ Tendría coste $\mathcal{O}(n)$ en espacio.
- Para evitarlo el iterador debe ser capaz de ir *dando saltos*:
  - Se bajará todo a la izqda para encontrar el primer nodo de un subárbol.
  - Necesitaremos una pila con los antecesores no visitados para volver a ellos $\Rightarrow$ $\mathcal{O}(log(n))$ en espacio.

El TAD set usando un BST- Recorrido con iterador

```
class const_iterator {
protected:
  friend class set;
  Link act; // puntero al nodo apuntado por el iterador
  std::stack<Link> ancestros;  // antecesores no visitados

  // construye el iterador al primero
  const_iterator(Link raiz) { act = first(raiz); }

  // construye el iterador al último
  const_iterator() : act(nullptr) {}

  Link first(Link ptr) { //encuentra 1er nodo de subárbol ptr
    if (ptr == nullptr) return nullptr;
    else { // buscamos el nodo más a la izquierda
      while (ptr->iz != nullptr) {
        ancestros.push(ptr); // guarda antecesores no visitados
        ptr = ptr->iz;
      }
      return ptr;
    }
  }
```

El TAD set usando un BST- Recorrido con iterador

```
public: // class const_iterator
  T const& operator*() const {
    if (act == nullptr)
      throw std::out_of_range("No hay elemento a consultar");
    return act->elem;
  }

  const_iterator& operator++() {  // ++ prefijo
    next();
    return *this;
  }

  bool operator==(const_iterator const& that) const {
    return act == that.act;
  }

  bool operator!=(const_iterator const& that) const {
    return !(this->operator==(that));
  }
```

El TAD set usando un BST- Recorrido con iterador

```
protected: // class const_iterator
  void next() { // avanza puntero act a su siguiente
    if (act == nullptr) {
      throw std::out_of_range("El iterador no puede avanzar");
    } else if (act->dr != nullptr) { // primero del hijo derecho
      act = first(act->dr);
    } else if (ancestros.empty()) { // hemos llegado al final
      act = nullptr;
    } else { // podemos retroceder
      act = ancestros.top();
      ancestros.pop();
    }
  }
};// class const_iterator

const_iterator begin() const {
  return const_iterator(raiz);
}

const_iterator end() const {
  return const_iterator();
}
```

El TAD set usando un BST- Recorrido con iterador

- Los métodos `first` y `next` tienen complejidad aislada $\mathcal{O}(log(n))$, y por tanto también `begin` y `++it`.
- Un análisis *basto* de un recorrido como este diría que su complejidad es $\mathcal{O}(n * log(n))$:

```
set<int> s;
...
for (set<int>::const_iterator it = s.begin(); it != l.end(); ++it){
    cout << *it << endl;
}
```

- Sin embargo si hacemos un análisis más fino podemos ver que el recorrido completo no pasa por ningún nodo más de dos veces.

El recorrido de un set es $\mathcal{O}(n)$

El TAD set usando un BST- Equivalencia

- Por último, ahora podemos implementar la relación de equivalencia de la clase `set` (`operator==`) fácilmente recorriendo los árboles con sus iteradores:

```
bool operator==(set<T>& other) const {
  if (this->size() != other.size()) return false;
  bool eq = true;
  for (const_iterator it = this->begin(), itOther = other.begin();
                      it != this->end() && eq; ++it, ++itOther){
    if (*it != *itOther) eq = false;
  }
  return eq;
```

- Su complejidad sería $\mathcal{O}(n)$.