Árboles de búsqueda

- Un árbol de búsqueda es un árbol binario tal que el elemento almacenado en su raíz es mayor que todos los elementos del hijo izquierdo y menor que todos los elementos del hijo derecho; además tanto el hijo izquierdo como el hijo derecho son a su vez árboles binarios de búsqueda.
- Los elementos deben tener definida una relación de orden (operador $<$)
- No permitimos la aparición de elementos repetidos; existen otras variaciones de estos árboles que sí lo permiten.
- Cumplen que su recorrido en inorden está ordenado en orden estrictamente creciente.

- Las operaciones de buscar, insertar y eliminar un elemento del árbol tienen coste del orden de la altura del árbol.
- Cada nodo del árbol almacena dos valores: la *clave* que se utiliza para ordenar los nodos y el *valor* que es la información adicional.
- El TAD estará parametrizado *por dos tipos distintos*: el tipo de la clave (que debe poderse ordenar) y el tipo del valor.

- Operaciones del TAD `Arbus`:
  - `ArbusVacio`: generadora. Construye un árbol de búsqueda vacío.
  - `Inserta`: generadora. Añade una nueva clave con su valor asociado. Si la clave ya existía en el árbol se sobreescribe el valor.
  - `borra`: modificadora, borra del árbol una clave y su valor asociado.
  - `esta`: observadora. Permite preguntar si una clave aparece almacenada en el árbol.
  - `consulta`: observadora. Dada una clave cualquiera devuelve su valor asociado. Es parcial (falla si la clave no `esta`).
  - `esVacio`: observadora que permite ver si el árbol guarda algo de información.
- Las operaciones generadoras permiten construir el mismo árbol de diferentes formas (no son *libres*).

Implementación

- Tipo representante: Cada nodo del árbol será representado como un nodo en memoria que contendrá cuatro atributos: la clave y el valor del elemento almacenado y punteros al hijo izquierdo y al hijo derecho.
  Un *puntero* a la raíz de la *estructura jerárquica de nodos* que representan el árbol.
- Invariante de la representación: exige lo mismo que en los árboles binarios (añadiendo que se cumpla el invariante de la representación tanto de la clave como del valor), y que se mantenga el orden de las claves.
- Relación de equivalencia: dos árboles de búsqueda son equivalentes si almacenan los mismos elementos.
- Las operaciones generales de liberación de memoria, copia, etc. de los árboles binarios se extienden para que tengan en cuenta la existencia de dos valores.

```
template <class Clave, class Valor>
class Arbus {
public:    ...
private:
 class Nodo {
 public:
  Nodo() : _iz(NULL), _dr(NULL) {}
  Nodo(const Clave &clave, const Valor &valor) :
   _clave(clave), _valor(valor), _iz(NULL), _dr(NULL) {}
  Nodo(Nodo *iz, const Clave &clave,
      const Valor &valor, Nodo *dr) :
      _clave(clave), _valor(valor), _iz(iz), _dr(dr)
   {}
  Clave _clave;
  Valor _valor;
  Nodo *_iz;
  Nodo *_dr;
 };
 Nodo *_ra;
};
```

Implementación de las operaciones públicas

```
/** Constructor; operacion ArbolVacio */
Arbus() : _ra(NULL) { }

// Observadora que devuelve si el árbol
// es vacío (no contiene elementos) o no.
bool esVacio() const {
    return _ra == NULL;
}
```

Las operaciones de búsqueda de una clave y consulta del valor asociado se basan en un método recursivo auxiliar que trabaja con la estructura de nodos y busca el nodo que contiene una clave dada.

```
// Método protegido/privado
// Busca una clave en la estructura jerárquica de nodos
// @param p Puntero a la raíz de la estructura de nodos
// @param clave Clave a buscar
static Nodo *buscaAux(Nodo *p, const Clave &clave) {
    if (p == NULL) return NULL;
    if (p->_clave == clave)  return p;
    if (clave < p->_clave)
        return buscaAux(p->_iz, clave);
    else
        return buscaAux(p->_dr, clave);
}
```

```
// Devuelve el valor asociado a una clave.
// @param clave Clave por la que se pregunta.
const Valor &consulta(const Clave &clave) {
    Nodo *p = buscaAux(_ra, clave);
    if (p == NULL) throw EClaveErronea();
    return p->_valor;
}

// Averigua si una clave está en el abb.
// @param clave Clave por la que se pregunta.
bool esta(const Clave &clave) {
    return buscaAux(_ra, clave) != NULL;
}
```

Implementación de la inserción

- La inserción debe garantizar que:
  - Si la clave ya aparecía en el árbol, el valor antiguo se sustituye por el nuevo.
  - Tras la inserción, el árbol de búsqueda sigue cumpliendo el invariante de la representación, es decir, sigue estando ordenado por claves.
- La implementación debe "encontrar el hueco" en el que se debe crear el nuevo nodo, y si por el camino descubre que la clave ya existía, sustituir su valor.
- Los nuevos nodos se crean siempre como hojas del árbol (raíz de un árbol vacío).
- La implementación se realiza con un método recursivo auxiliar.

```
// Método protegido/privado
// Inserta una pareja (clave, valor) en la estructura jerárquica que comienza
// en el puntero pasado como parámetro.
// Ese puntero se admite que sea NULL, por lo que se creará un nuevo nodo que
// pasará a ser la nueva raíz de esa estructura jerárquica. El método devuelve
// un puntero a la raíz de la estructura modificada. En condiciones normales
// coincidirá con el parámetro recibido; sólo cambiará si la estructura era vacía.
// @param clave Clave a insertar. Si ya aparecía en la estructura de nodos,
            se sobreescribe el valor.
// @param valor Valor a insertar.
// @param p Puntero al nodo raíz donde insertar la pareja.
// @return Nueva raíz (o p si no cambia).
```

```
static Nodo *insertaAux(const Clave &clave,
                        const Valor &valor, Nodo *p) {
  if (p == NULL) {  return new Nodo(clave, valor); }
  else  {
    if (p->_clave == clave)  p->_valor = valor;
    else if (clave < p->_clave)
        p->_iz = insertaAux(clave, valor, p->_iz);
    else  // (clave > p->_clave)
        p->_dr = insertaAux(clave, valor, p->_dr);
    return p;
    }
}
```

```
// Añadir una nueva clave/valor a un abb.
// @param clave Clave nueva.
// @param valor Valor asociado a esa clave.
void inserta(const Clave &clave, const Valor &valor) {
    _ra = insertaAux(clave, valor, _ra);
}
```

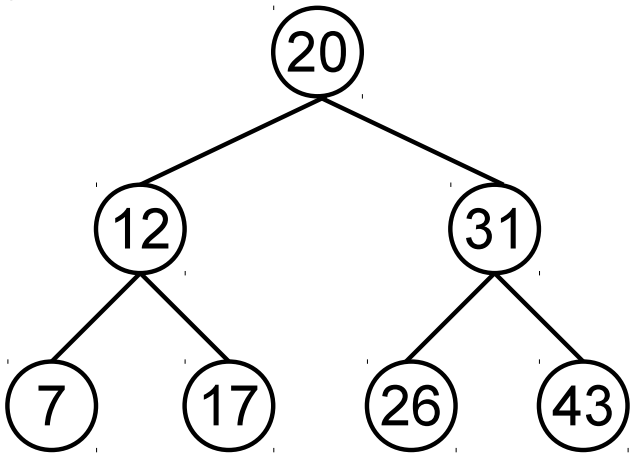
- Si suponemos que el árbol está equilibrado, la complejidad del método anterior es logarítmica pues en cada llamada recursiva se divide el tamaño de los datos entre dos.

Implementación del borrado

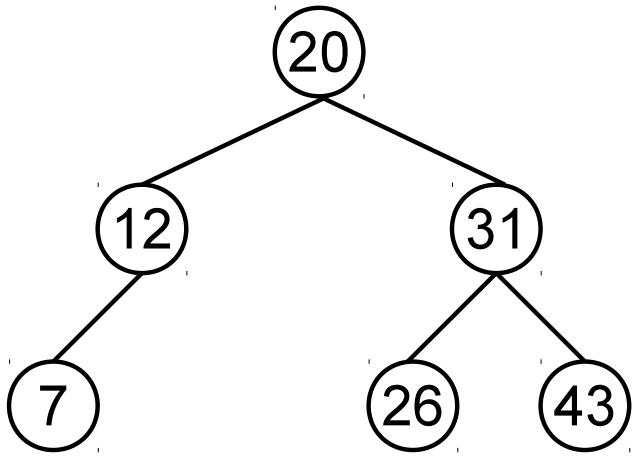
La operación de borrado puede exigir reestructurar los nodos para mantener la estructura ordenada por claves.
Si nos piden eliminar la clave c, se busca el nodo que la contiene y:

- Si la clave no está, se termina sin modificar el árbol.
- Si la búsqueda tiene éxito:
  - Si es una hoja, se elimina actualizando el puntero del padre.
  - Si tiene un sólo hijo, se elimina el nodo y se coloca en su lugar el subárbol hijo cuya raíz quedará en el lugar del nodo.
  - Si tiene dos hijos lo cambiamos por el elemento más pequeño de su hijo derecho (que no tendrá hijo izquierdo, pues de otra forma no sería el más pequeño):
    - Se busca el nodo α más pequeño del hijo derecho.
    - Si ese nodo tiene hijo derecho, éste pasa a ocupar su lugar.
    - El nodo α pasa a ocupar el lugar del nodo a borrar, de forma que su hijo izquierdo y derecho cambian a los hijos izquierdo y derecho del nodo a borrar.

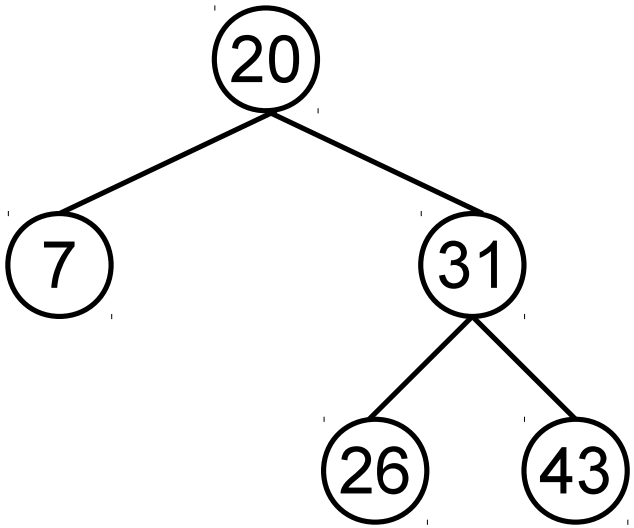
Ejemplo



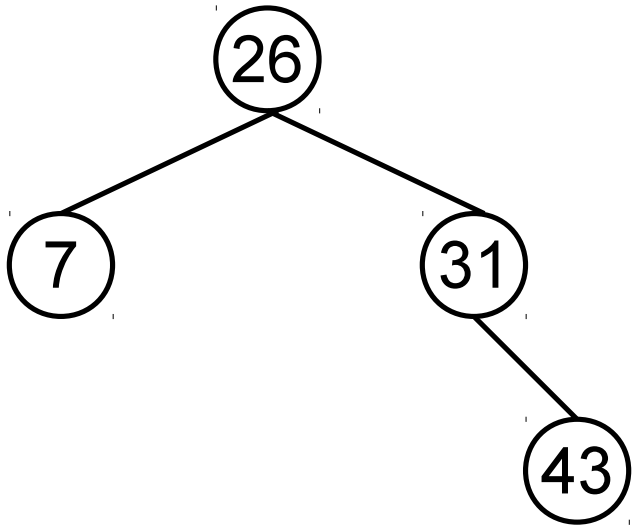
Árbol inicial



a.borrar(17)



a.borrar(12)



a.borrar(20)

```
// Elimina una clave del árbol.
// Si la clave no existía la operación no tiene efecto.
//@param clave Clave a eliminar.
```

```
void borra(const Clave &clave) {
    _ra = borraAux(_ra, clave);
}
```

```
private:
// Elimina (si existe) la clave/valor de la estructura jerárquica de nodos
// apuntada por p. Si la clave aparecía en la propia raíz, ésta cambiará,
// por lo que se devuelve la nueva raíz. Si no cambia se devuelve p.
// @param p Raíz de la estructura jerárquica donde borrar la clave.
// @param clave Clave a borrar.
// @return Nueva raíz de la estructura, tras el borrado. Si la raíz
//  no cambia, se devuelve el propio p.
```

```
static Nodo *borraAux(Nodo *p, const Clave &clave) {
    if (p == NULL)  return NULL;
    if (clave == p->_clave) return borraRaiz(p);
    else {
       if (clave < p->_clave)
            p->_iz = borraAux(p->_iz, clave);
       else  p->_dr = borraAux(p->_dr, clave);
      return p;  }
}
```

```
// Borra la raíz de la estructura jerárquica de nodos y devuelve el puntero
// a la nueva raíz que garantiza que la estructura sigue siendo válida para
//un árbol de búsqueda (claves ordenadas).
```

```
static Nodo *borraRaiz(Nodo *p) {
    Nodo *aux;
    // No hay hijo izquierdo
        if (p->_iz == NULL) {
        aux = p->_dr;  delete p;  return aux;
    } else
    // No hay hijo derecho
    if (p->_dr == NULL) {
        aux = p->_iz; delete p;  return aux;
    } else {
    // Convertimos el elemento más pequeño del hijo
    // derecho en la raíz.
        return mueveMinYBorra(p);
    }
}
```

```
// Método auxiliar; recibe un puntero a la raíz a borrar.
// Busca el elemento más pequeño del hijo derecho que se convertirá en la
 raíz (que devolverá), borra la antigua raíz (p) y "cose" todos los punteros:
   – El mínimo pasa a ser la raíz, cuyo hijo izquierdo y derecho eran los
     hijos izquierdo y derecho de la raíz antigua.
   – El hijo izquierdo del padre del elemento más pequeño pasa a ser el
     antiguo hijo derecho de ese mínimo.

static Nodo *mueveMinYBorra(Nodo *p) {
   // Vamos bajando hasta que encontramos el elemento más pequeño
    // Vamos guardando también el padre (que será null si el hijo
    // derecho es directamente el elemento más pequeño).
    Nodo *padre = NULL; Nodo *aux = p->_dr;
    while (aux->_iz != NULL) {
        padre = aux; aux = aux->_iz;
    }
    // aux apunta al elemento más pequeño.
    // padre apunta a su padre (si el nodo es hijo izquierdo)
    // Arreglamos los punteros del padre
    if (padre != NULL) {
        padre->_iz = [redacted] aux->_dr;
    }
    // Y ahora los del nodo que pasa a ser la raíz
    aux->_iz = p->_iz;
    if (padre != NULL) {    // Si no tenía padre, el hijo derecho
        aux->_dr = p->_dr; // de la nueva raíz coincide con el hijo
    }                      // derecho del nodo más pequeño
    delete p;
    return aux;
}
```

Recorrido de los elementos mediante un iterador

- Extendemos el TAD con un iterador que lo recorre en inorden.
- El iterador permite acceder a la clave y el valor del elemento.
- *No* permite cambiar las claves porque el árbol podría dejar de estar ordenado.
- Siguiente elemento a un nodo:
  - Si el nodo visitado tiene hijo derecho, el siguiente nodo será el elemento más pequeño del hijo derecho (elemento mas a la izquierda).
  - Sino el siguiente elemento a visitar es el *ascendiente más cercano* que aún no ha sido visitado. El iterador mantendrá una *pila* con todos los ascendientes que aún quedan por recorrer. Al descender por una rama, se van apilando todos los descendientes para poder visitarlos después.
- El recorrido termina cuando el nodo actual no tiene hijo derecho y la pila queda vacía. En ese caso se cambia el puntero interno a `NULL` para indicar que estamos "fuera" del recorrido.

```
// Clase interna que implementa un iterador sobre
// la lista que permite recorrer la lista e incluso
// alterar el valor de sus elementos.
class Iterador {
public:
    const Clave &clave() const {
        if (_act == NULL) throw EAccesoInvalido();
        return _act->_clave;
    }

    const Valor &valor() const {
        if (_act == NULL) throw EAccesoInvalido();
        return _act->_valor;
    }
```

```
void avanza() {
    if (_act == NULL) throw EAccesoInvalido();
    // Si hay hijo derecho, saltamos al primero
    // en inorden del hijo derecho
    if (_act->_dr) _act=primeroInOrden(_act->_dr);
    else {
        // Si no, vamos al primer ascendiente
        // no visitado. Para eso consultamos
        // la pila; si ya está vacía, no quedan
        // ascendientes por visitar
        if (_ascendientes.esVacia()) _act = NULL;
        else {
            _act = _ascendientes.cima();
            _ascendientes.desapila();
        }
    }
}
```

```
    bool operator==(const Iterador &other) const {
        return _act == other._act;
    }

    bool operator!=(const Iterador &other) const {
        return !(this->operator==(other));
    }
protected:
    // Para que pueda construir objetos del
    // tipo iterador
    friend class Arbus;
    Iterador() : _act(NULL) {}
    Iterador(Nodo *act) {
        _act = primeroInOrden(act);
    }
```

```
/**
 Busca el primer elemento en inorden de
 la estructura jerárquica de nodos pasada
 como parámetro; va apilando sus ascendientes
 para poder "ir hacia atrás" cuando sea necesario.
 @param p Puntero a la raíz de la subestructura.
 */
Nodo *primeroInOrden(Nodo *p) {
    if (p == NULL)
        return NULL;

    while (p->_iz != NULL) {
        _ascendientes.apila(p);
        p = p->_iz;
    }
    return p;
}
```

```
    // Puntero al nodo actual del recorrido
    // NULL si hemos llegado al final.
    Nodo *_act;

    // Ascendientes del nodo actual
    // aún por visitar
    Pila<Nodo*> _ascendientes;
};

//Devuelve el iterador al principio de la lista.
Iterador principio() {
    return Iterador(_ra);
}

// @return Devuelve un iterador al final del recorrido
Iterador final() const {
    return Iterador(NULL);
}
```

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.

Para terminar...

- Podemos ver un texto como una secuencia (lista) de palabras, donde cada una de ellas es un `string`.
- El problema de las *concordancias* consiste en contar el número de veces que aparece cada palabra en ese texto.
- Implementar una función que reciba un texto como lista de las palabras (en forma de cadena) y escriba una línea por cada palabra distinta del texto donde se indique la palabra y el número de veces que aparece.
- La lista debe aparecer ordenada alfabéticamente.
- La implementación utiliza un árbol de búsqueda como variable local que va almacenando, para cada palabra encontrada en el texto, el número de veces que ha aparecido.

```
void refsCruzadas(Lista<string> &texto) {
  Lista<string>::Iterador it(texto.principio());
  Arbus<string, int> refs;
  while (it != texto.final()) {
    if (!refs.esta(it.elem()))
      refs.inserta(it.elem(), 1);
    else
      refs.inserta(it.elem(),1+refs.consulta(it.elem()))
    it.avanza();
  }
  // Y ahora escribimos
  Arbus<string, int>::Iterador ita = refs.principio();
  while (ita != refs.final()) {
    cout << ita.clave() << " " << ita.valor() << endl;
    ita.avanza();
  }
}
```