Lección 10: Árboles

- Árboles
- Árboles binarios
- Recorridos
- Árboles binarios de búsqueda
- Inserción y borrado en los ABB
- Consideraciones finales

Motivación

Mymail es una nueva compañía que ofrece cuentas de correo electrónico a sus usuarios. Al cabo de varias semanas ya cuenta con miles de usuarios, y a este ritmo se esperan que sean millones. Cómo manejamos dicha cuentas de modo eficiente?

...	maripili@ mymail.com	piluca@ mymail.com	lucasin@ mymail.com	josefo@ mymail.com	nachete@ mymail.com	...

Los vectores no son eficientes:
- Si están ordenados cuesta O(n) insertar nuevas cuentas.
- Si no están ordenados cuesta O(n) buscar un alias.

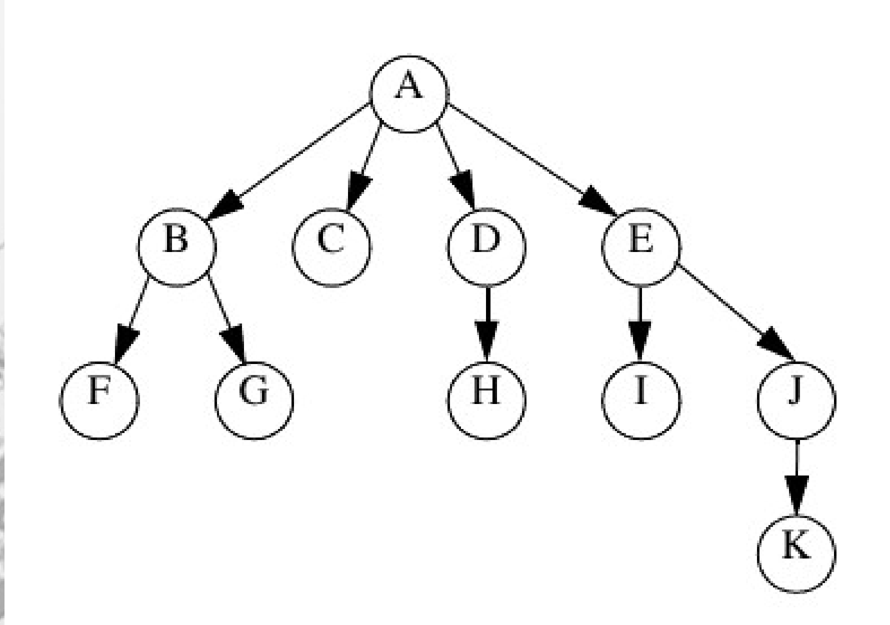
Motivación

- Las estructuras de datos estudiadas hasta el momento son EEDD lineales con acceso directo o secuencial.
- Para este problema se necesita una estructura de datos de acceso por clave o **contenedor asociativo** que permita búsquedas eficientes.
- Por ejemplo, para que la búsqueda sea logarítmica (en base 2), en cada paso debemos descartar la mitad de los datos pendientes de procesar, como lo hace la búsqueda dicotómica.
- Algunos **árboles** funcionan como contenedores asociativos que permiten resolver la búsqueda de forma eficiente.

Definiciones

Un árbol es una jerarquía de nodos enlazados que cumple:

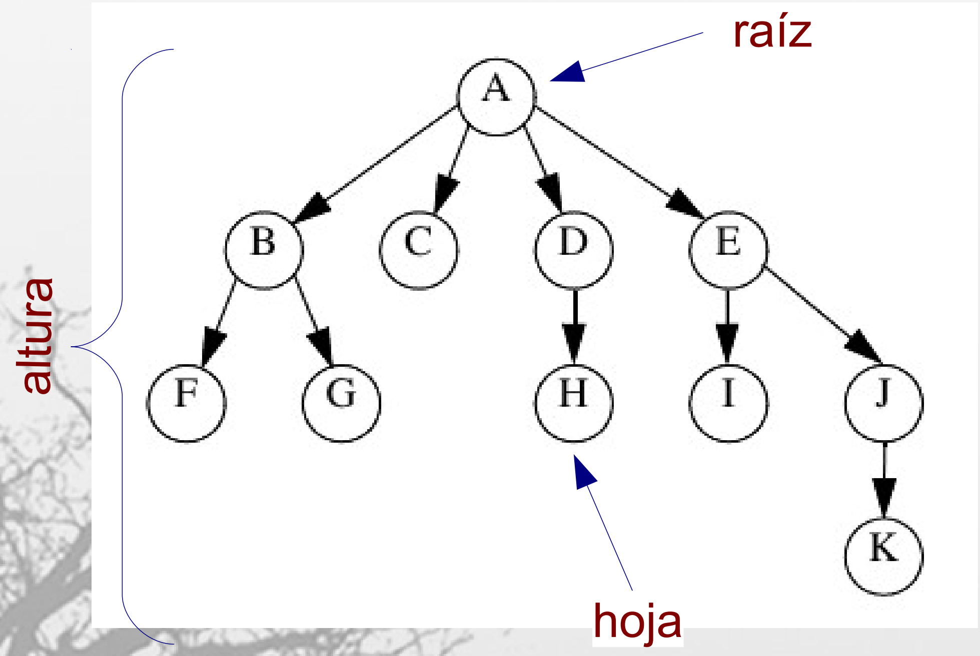
- Cada nodo tiene cero o más hijos
- Cada nodo tiene como como máximo un padre.



- A es el padre de D
- H es el hijo de D
- H no tiene hijos
- A no tiene padre

Definiciones

- El nodo del árbol que no tiene padre es la **raíz**
- Los nodos que no tienen hijos son **hojas**

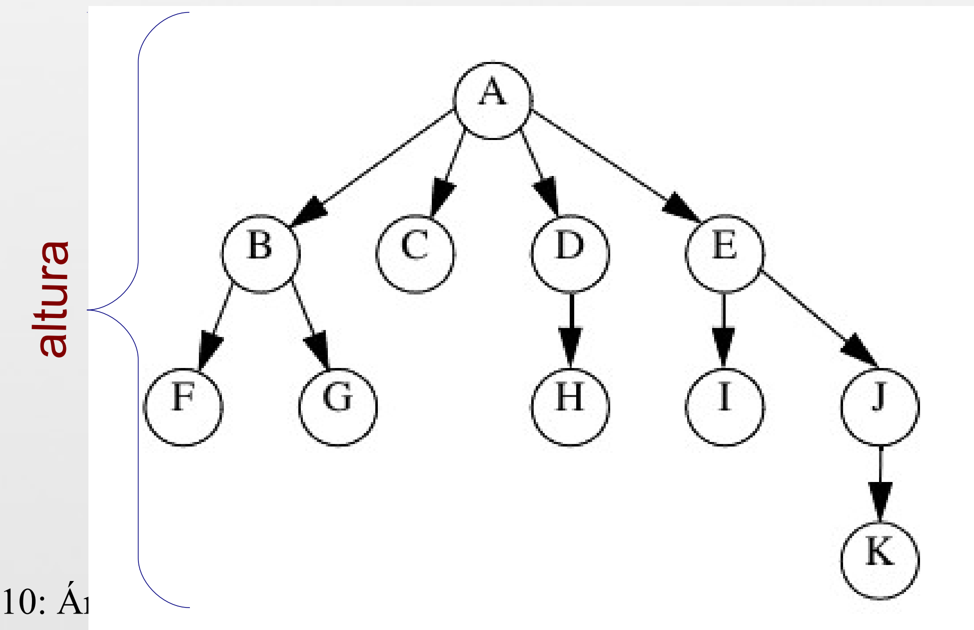


- A es el nodo raíz
- H es una hoja

Definiciones

- La **altura del árbol** es el camino más largo que se recorre desde la raíz hasta una hoja.
- La **altura de un nodo** es la longitud del camino más largo de dicho nodo a una hoja.
- La **profundidad** de un nodo es la longitud del camino desde la raíz a dicho nodo.

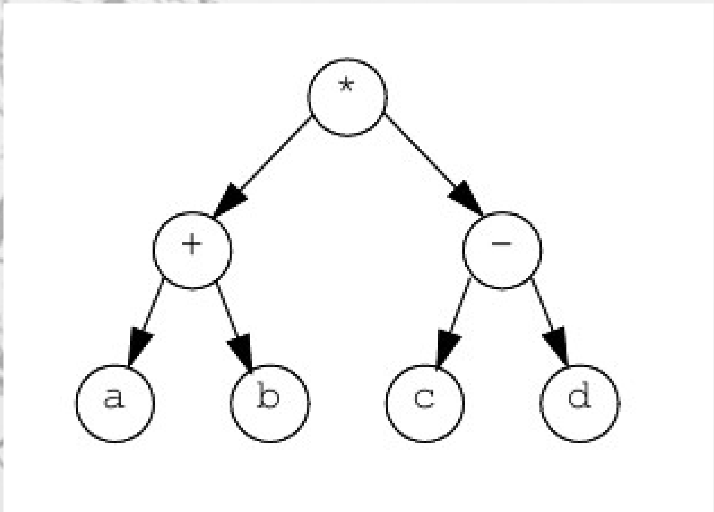
- La altura del árbol es 3
- La altura del nodo E es 2
- La altura del nodo F es 0
- La profundidad de J es 2



Aplicaciones

Los árboles son una estructura de datos extremadamente versátil que permite representar:

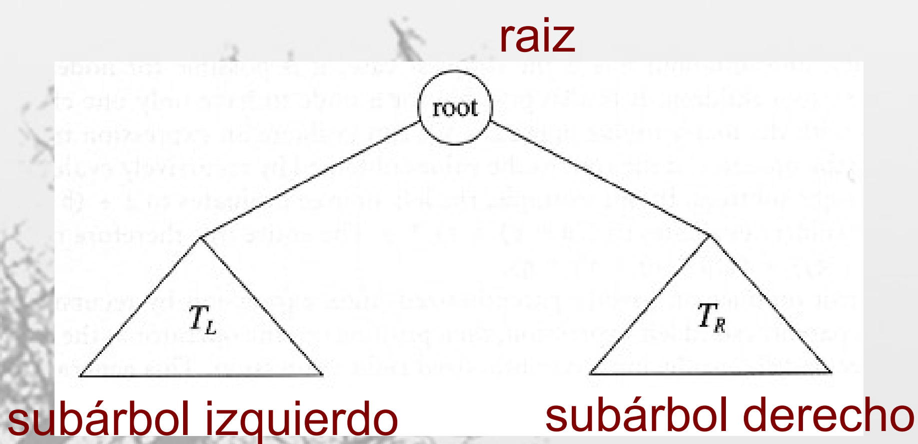
- Genealogías u organigramas,
- Circuitos eléctricos,
- Estructuras de directorios, o fórmulas matemáticas
- Y por supuesto contenedores asociativos



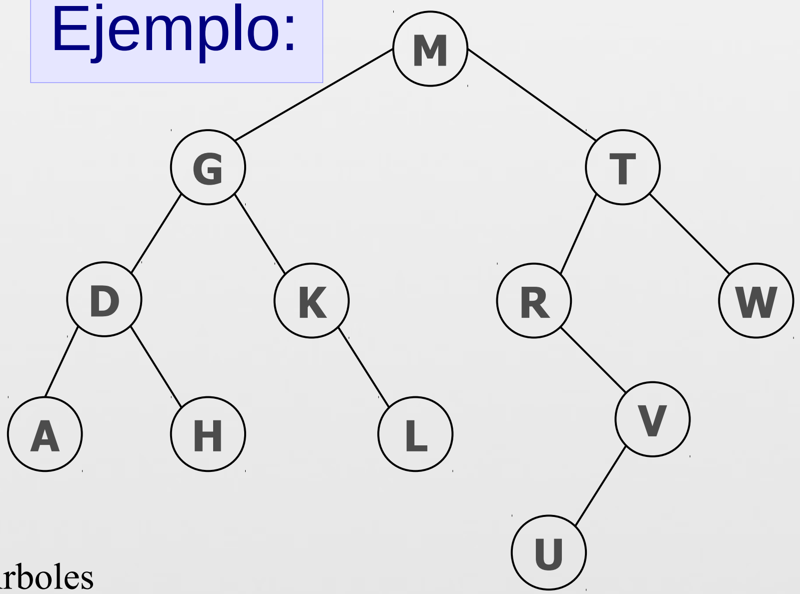
(a+b)*(c-d)

Árboles binarios

- El **orden** de un árbol es el máximo número de nodos que puede tener un nodo.
- Un **árbol binario** tiene orden 2.
- Los árboles binarios son los más utilizados como contenedores de datos en memoria.

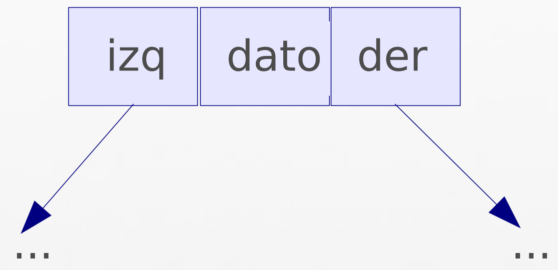


Ejemplo:



Definiendo árboles binarios en C++

```
template <class T>
class Nodo
{
   Nodo<T> *izq;
   Nodo<T> *der;
   T dato;
public:
   Nodo(): izq(0), der(0){}
   Nodo(T &ele): izq(0), der(0), dato(ele){}
};
```



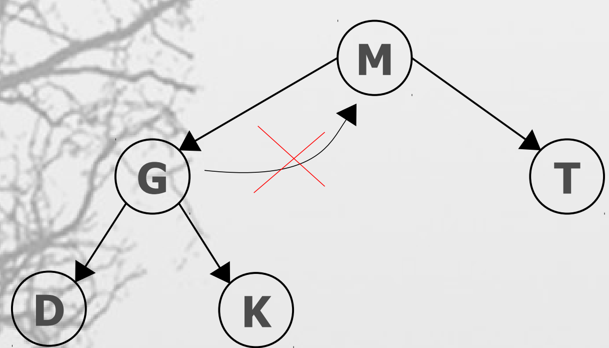
La definición de nodo es recursiva

```
template <class T>
class Abb {
    Nodo<T> *raiz;
public:
    Abb();
    ...
};
```

Un árbol es un puntero a un nodo, el nodo raíz

Recorridos de árboles

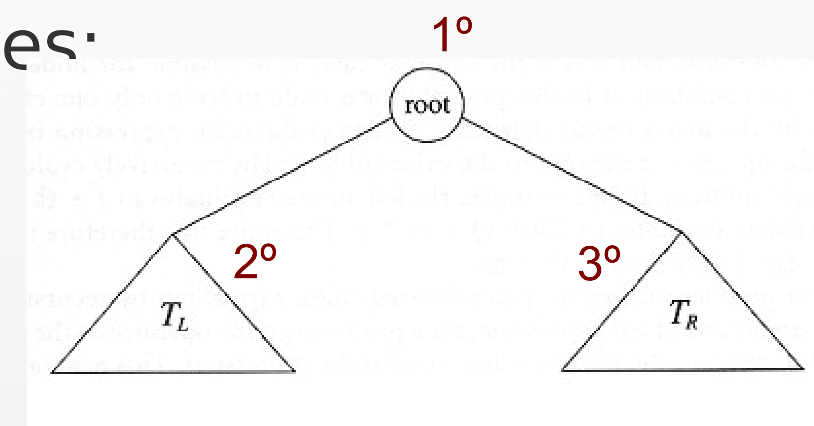
- El mecanismo para acceder a los datos de un árbol se parece al de la lista enlazada: de un nodo padre se pasa a un nodo hijo (el izquierdo o derecho).
- Pero no al revés, desde un nodo hijo no tenemos acceso al nodo padre.
- Los métodos más simples de recorrido son los siguientes métodos recursivos: **preorden, inorden** y **postorden**.

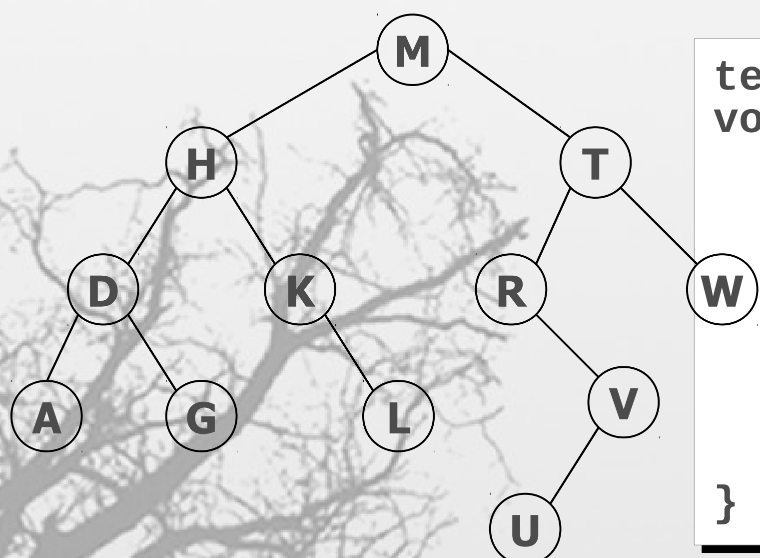


Recorridos: preorden

En Preorden el orden de visita es:

1. Nodo raíz
2. Nodo izquierda (recursivo)
3. Nodo derecha (recursivo)





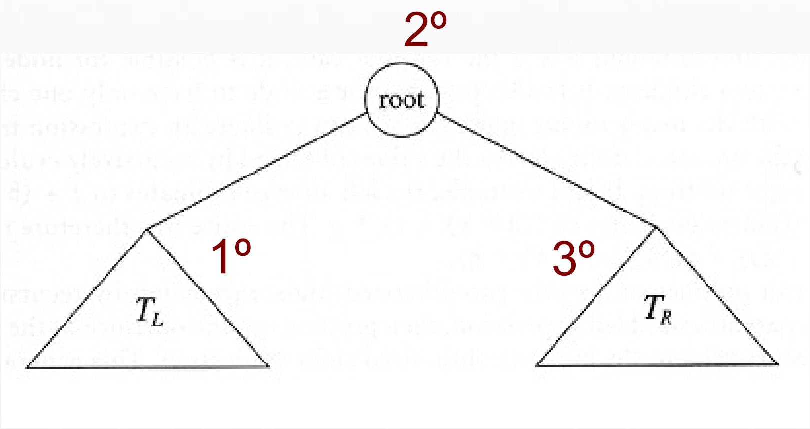

```
template <class T>
void Abb<T>::preorden (Nodo<T> *p, int nivel){
    if (p){
         // Sustituir por procesamiento ---
        cout << "Procesando nodo "<< p->dato ;
        cout << "en el nivel " << nivel << endl;
        // -----------
        preorden (p->izq, nivel+1);
        preorden (p->der, nivel+1);
    }
}
```

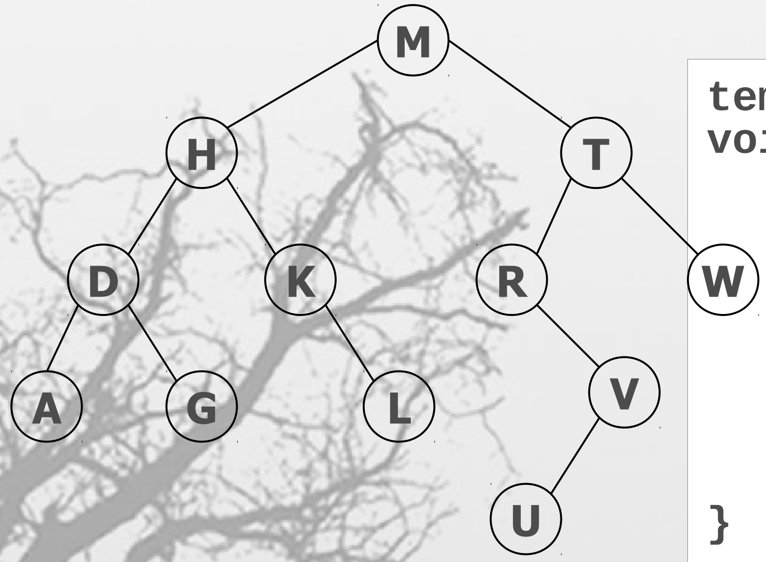
M H D A G K L T R V U W

Recorridos: inorden

En Inorden orden de visita es:

1. Nodo izquierda (recursivo)
2. Nodo raíz
3. Nodo derecha (recursivo)





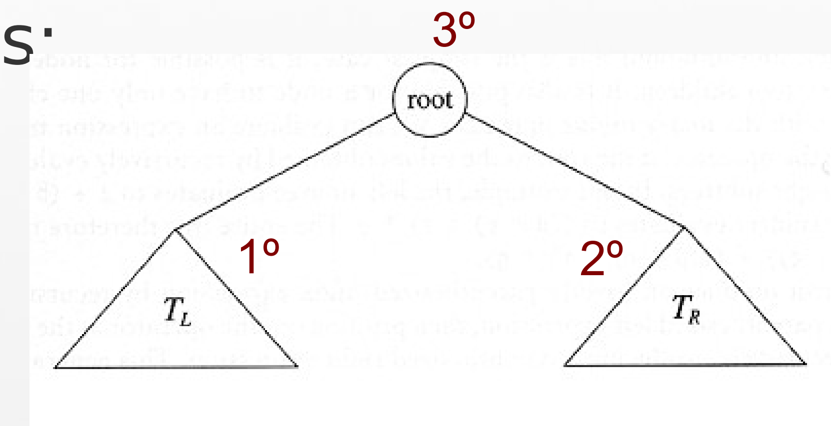

```
template <class T>
void Abb<T>::inorden (Nodo<T> *p, int nivel){
    if (p){
        inorden (p->izq, nivel+1);
        // Sustituir por procesamiento ----
        cout << "Procesando nodo " << p->dato;
        cout << "en el nivel " << nivel << endl;
        // -----------
        inorden (p->der, nivel+1);
    }
}
```

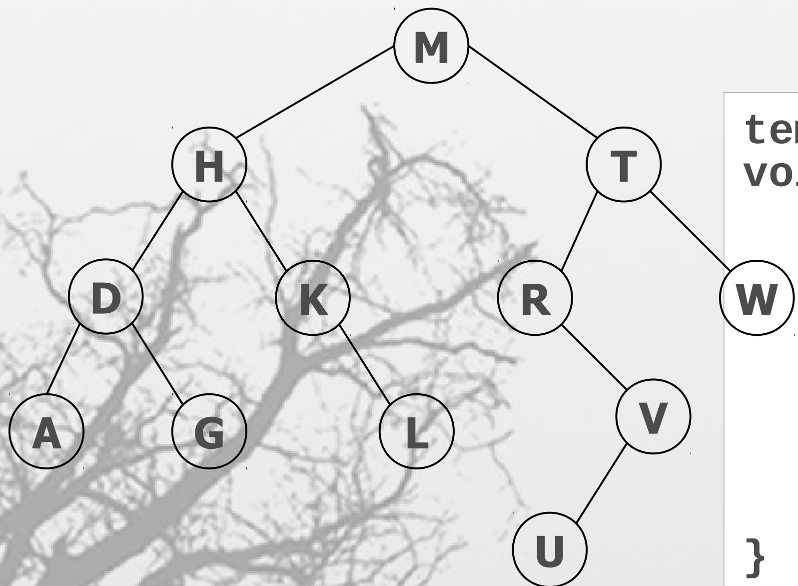
A D G H K L M R U V T W

Recorridos: postorden

En Inorden el orden de visita es:

1. Nodo izquierda (recursivo)
2. Nodo derecha (recursivo)
3. Nodo raíz






```
template <class T>
void Abb<T>::postorden (Nodo<T> *p, int nivel){
    if (p){
        postorden (p->izq, nivel+1);
        postorden (p->der, nivel+1);
        // Sustituir por procesamiento ----
        cout << "Procesando nodo "<< p->dato;
        cout << "en el nivel " << nivel << endl;
        // -----------
    }
}
```

A G D L K H U V R W T M

Recorridos

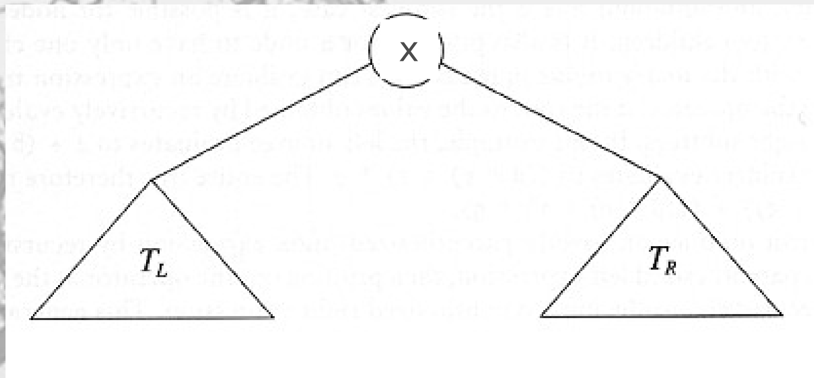
- Las funciones anteriores *preorden()*, *inorden()* y *postorden()* son recursivas, y deben ser privadas.
- Otras funciones públicas deben invocarlas.
- Existen versiones iterativas para realizar los recorridos, pero son más complejas.

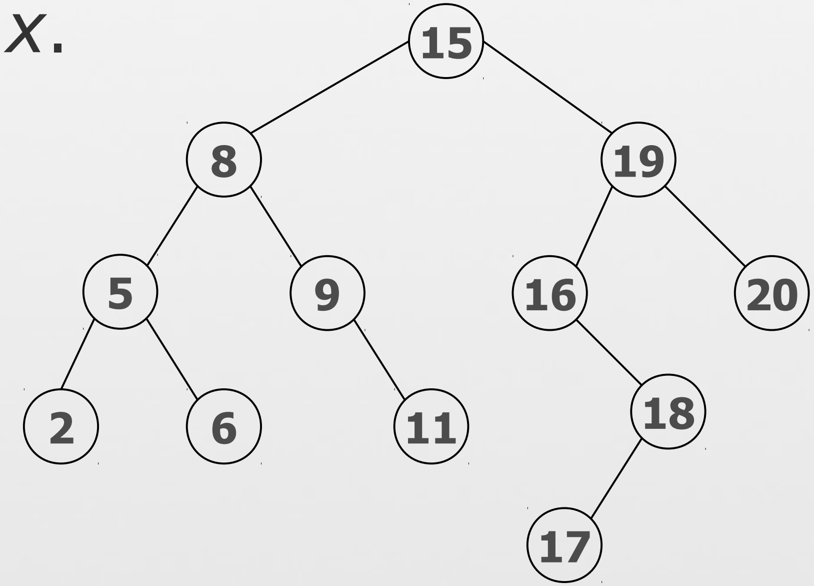
```
template <class T>
class Abb {
    Nodo<T> *raiz;
    void preorden(Nodo<T> *p, int nivel);
    void inorden(Nodo<T> *p, int nivel);
    void postorden(Nodo<T> *p, int nivel);
public:
    Abb(void);
    void recorrePreorden() { preorden(raiz,0); }
    void recorreInorden() { inorden(raiz,0); }
    void recorrePostorden() { postorden(raiz,0); }
…
}
```

Árboles binarios de búsqueda

En un árbol binario de búsqueda se debe cumplir la siguiente regla:

- Para cada nodo *x*, todas las claves en el subárbol izquierdo T_L deben ser menores que el valor de *x*, y todas las claves del subárbol derecho T_R deben ser mayores que el valor de *x*.

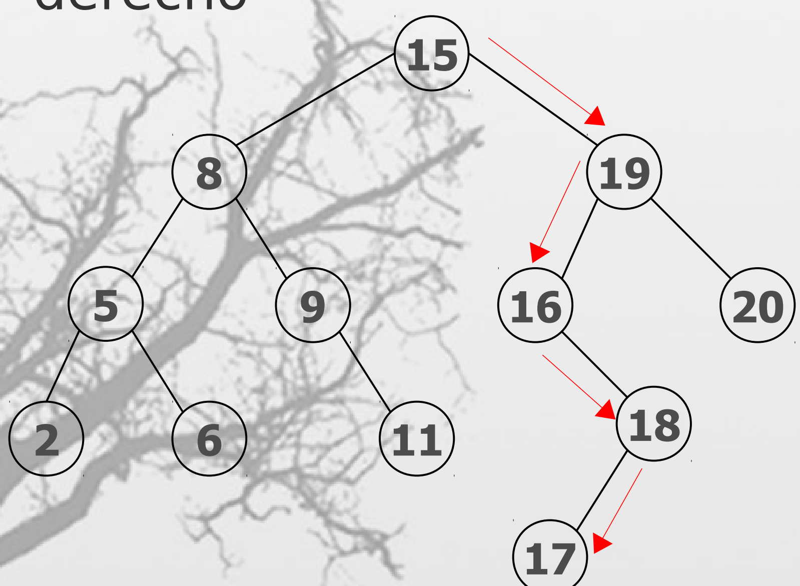




Búsqueda en los ABB

Los árboles binarios de búsqueda (ABB) son interesantes porque agilizan la búsqueda del dato *q*:

- Si q es igual al dato en la raíz → encontrado
- Si q menor que el dato en la raíz → continuar por subárbol izquierdo
- Si q es mayor que el dato en la raíz → continuar por subárbol derecho



Buscamos el 17:
17 > 15 → derecha
17 < 19 → izquierda
17 > 16 → derecha
17 < 18 → izquierda
17 == 17 → encontrado

Búsqueda en los ABB

- Si un árbol está equilibrado la búsqueda es O(*log*n)

```
template <class T>
Nodo<T> *Abb<T>::buscaClave (T &ele, Nodo<T> *p){
    if (!p)
        return 0;
    else if (ele < p->dato)
        return buscaClave (ele, p->izq);
    else if (ele > p-> dato)
        return buscaClave (ele, p->der);
    else return p;
}

template <class T>
bool Abb<T>::buscar (T &ele, T &result){
    Nodo<T> *p = buscaClave (ele, raiz);
    if (p) {
        result = p->dato;
        return true;
    }
    return false;
}
```

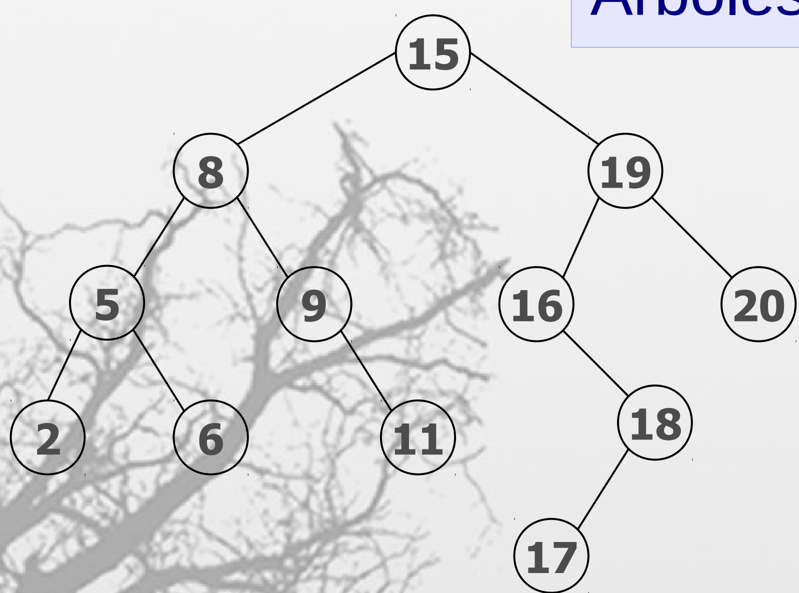
Función privada

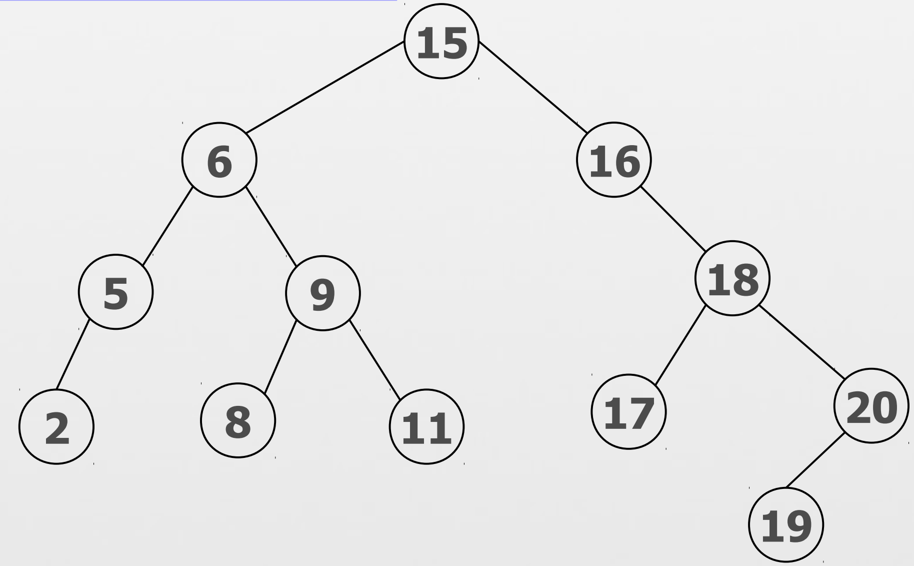
Función pública

Insertar en un ABB

- Un nuevo nodo siempre queda ubicado en una hoja.
- El orden de inserción determina la forma del árbol

Árboles equivalentes





Insertar en un ABB

Este código asume que no se repiten claves

```
template <class T>
Nodo<T> *Abb<T>::insertaDato(T &ele, Nodo<T>* &p){
    if (!p){
        p = new Nodo<T>(ele);
    } else {
        if (ele <= p->dato)
            p->izq = insertaDato(ele, p->izq);
        else
            p->der = insertaDato(ele, p->der);
    }
    return p;
}
```

Función privada

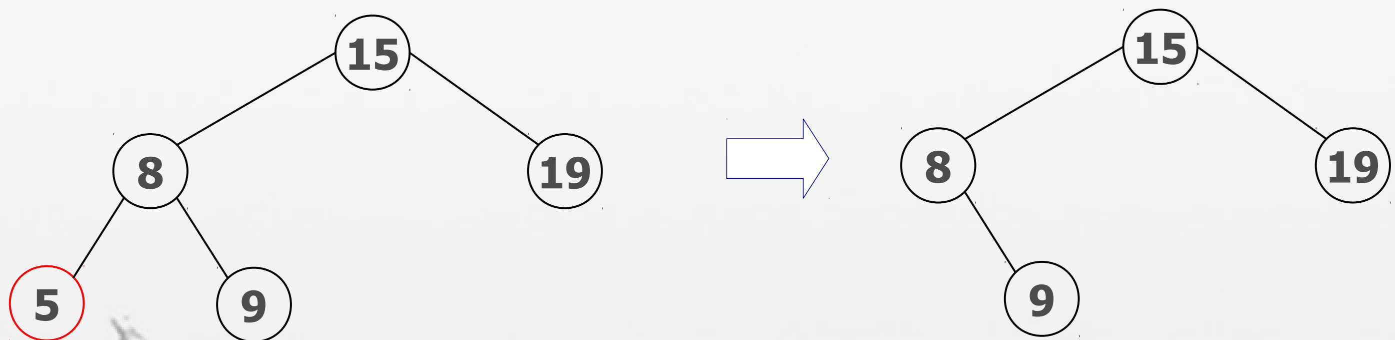
```
template <class T>
bool Abb<T>::insertar(T &ele){
    bool encontrado = buscar(ele);
    if (!encontrado){
        insertaDato(ele, raiz);
        return true;
    }
    return false;
}
```

Función pública

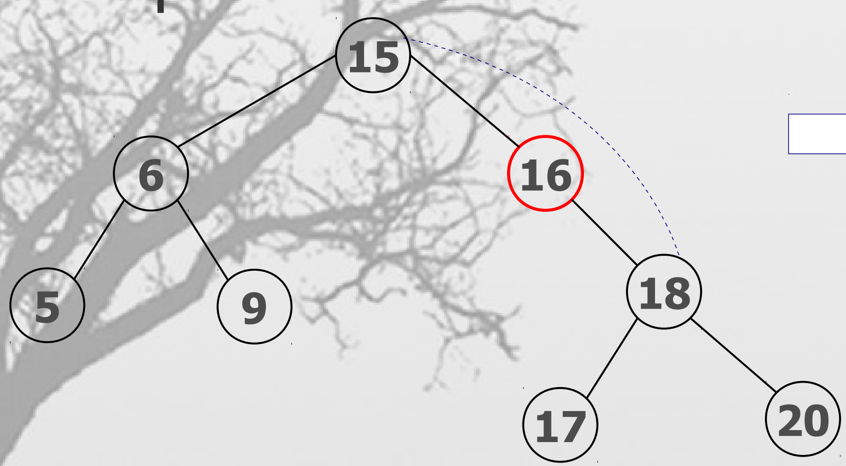
Borrar un nodo de un ABB

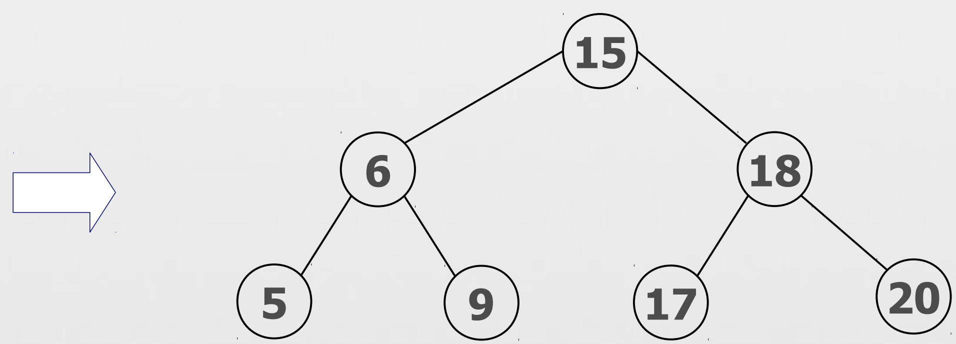
Depende del número de hijos:

- Si es una hoja: se elimina directamente el nodo



- Si el nodo *e* tiene sólo un hijo *h*: se engancha el padre de *e* con h

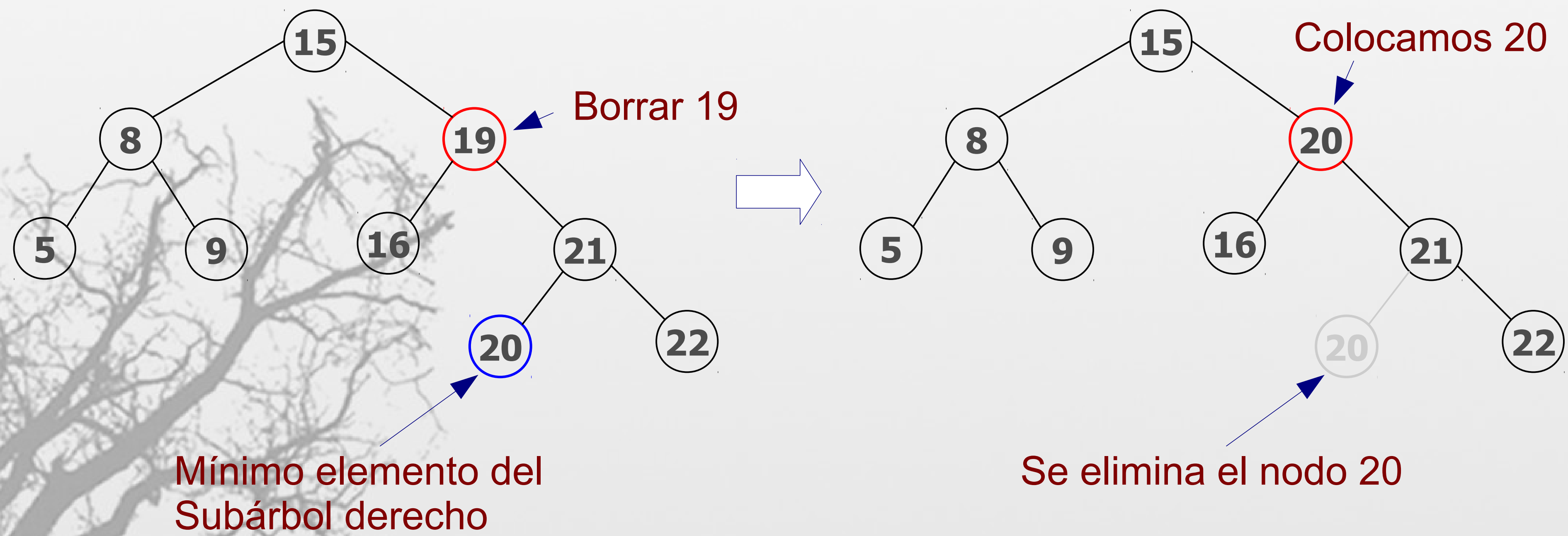




Borrar un nodo de un ABB

Si el nodo e tiene dos hijos:

- Encontrar *f*, el mínimo elemento en el subárbol derecho (*f* es una hoja)
- Colocar *f* en la posición de *e* y luego eliminar *f*



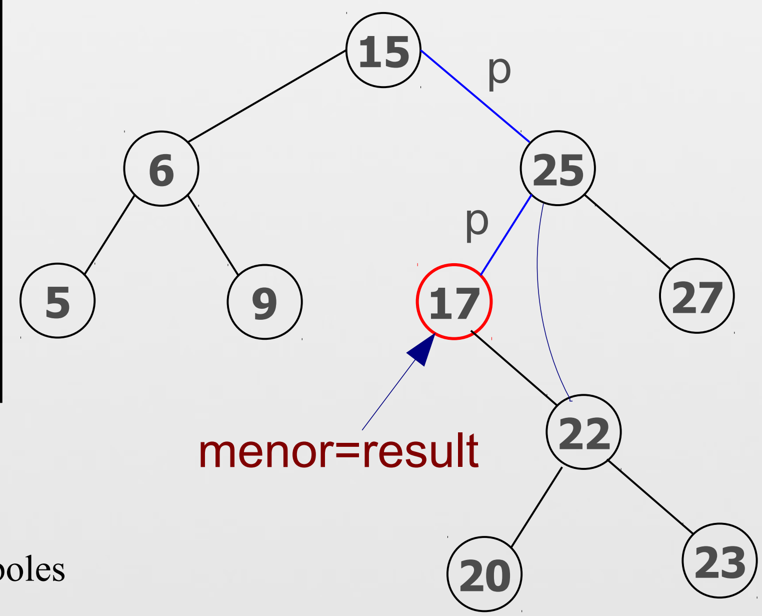
Borrar un nodo de un ABB

```
template <class T>
bool Abb<T>::eliminar(T &ele){
    Nodo<T> *result = borraDato(ele, raiz);
    if (result) return true;
    return false;
}
```

Función pública

```
template <class T>
Nodo<T> *Abb<T>::borraMin(Nodo<T>* &p){
    Nodo<T> *result;
    if (p){
        if (p->izq)
            return borraMin(p->izq);
        else {
            result = p;
            p = p->der;
            return result;
        }
    }
}
```

Función que encuentra el mínimo de un subárbol y lo deja aislado apuntado por *result*



Borrar un nodo de un ABB

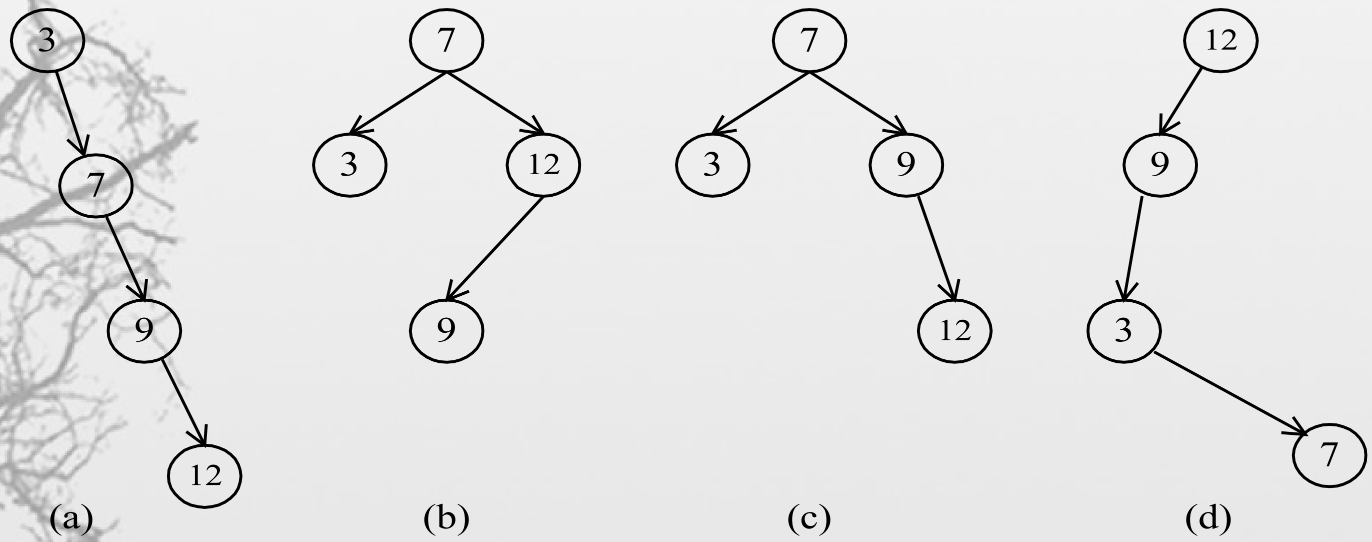
Función privada

```
template<class T>
Nodo<T> *Abb<T>::borraDato(T &ele, Nodo<T>* &p){
    if (p) {
      if (ele < p->dato)
          borraDato(ele, p->izq);
      else
          if (ele > p->dato)
              borraDato(ele, p->der);
          else {            //encontrado
              Nodo<T> *temp = p;
              if (!p->izq)  //tiene hijo a la decha
                  p = p->der;
              else
                  if (!p->der) //tiene hijo a la izda
                      p = p->izq;
                  else if (p->izq && p->der){ //tiene ambos hijos
                      temp = borraMin(p->der);
                      p->dato = temp->dato;
                  }
              delete temp; //borrar en todos los casos (también hoja)
              temp = 0;
          }
    }
    return p;
}
```

Consideraciones finales

Los ABB mejoran el tiempo de búsqueda pero:

- En el peor de los casos la búsqueda es lineal.
- Tan sólo se consigue tiempo logarítmico *esperado*
- La estructura es muy dependiente del orden de entrada de los datos.



Consideraciones finales

La solución a estos problemas pasa por conseguir que el árbol permanezca equilibrado tras las inserciones y borrados

- Un **arbol es equilibrado** si para cada nodo, el número de niveles de sus subárboles izquierdo y derecho no difiere en más de una unidad.
- Las búsquedas en estos casos sí son O(n*log*n).