Todo sobre Estructura de Datos No Lineales

Hecho por: Jose Luis Venega Sánchez

Índice general

1. Introducción a Árboles

En este tema vamos a ver todo sobre los árboles que se usan en programación.

Encontraremos varios tipos de árboles (binarios, generales, ABB, APOs, entre otros), que nos facilitarán el trabajo a la hora de resolver algunos problemas.

Para poder entender todos los conceptos que vamos a dar sobre los árboles, primero debemos de entender qué es un árbol.

Los árboles ofrecen dos ventajas frente a las estructura de datos lineales (listas, vectores, colas, etc..):

- Permiten representar **jerarquías**.
- En algunos casos podemos realizar búsquedas de **orden logarítmicas**: $O(log\ n)$.

1.1. Definición de árbol

Un árbol en el ámbito de la programación es un conjunto de elementos de un tipo determinado, donde cada elemento se almacena en un **nodo**.
En los árboles vamos a encontrar una *relación de paternidad* (padre - hijo) entre los diferentes nodos que conforman el árbol, como veremos en la *Figura 1.1*.

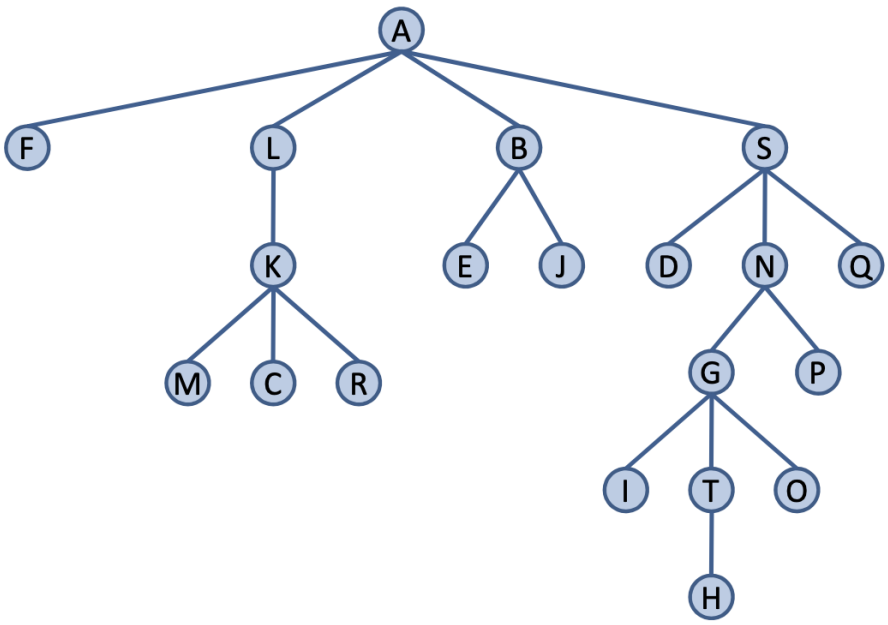


Figura 1.1: Ejemplo de árbol

Cada letra sería un nodo diferente.
Por ejemplo: el nodo A es padre de los nodos F, L, B y S.

Una definición más formal sería:

Si n es un **nodo** y A_1, A_2, ..., A_k son árboles con raíces n_1, n_2, ..., n_k y además se define una relación de paternidad (siendo hijos de n y hermanos entre sí), tenemos como resultado un árbol (*Figura 1.2:Definición de árbol*).

Si solamente tenemos un único nodo, en nuestro caso n, diremos que tenemos un **árbol vacío**.

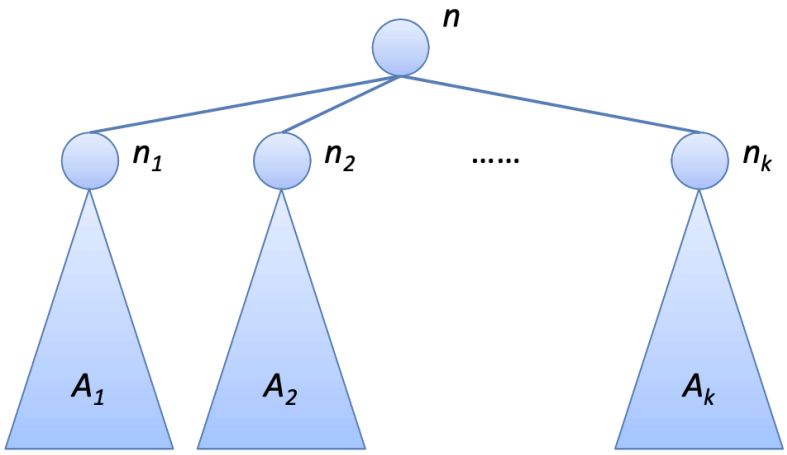


Figura 1.2: Definición de árbol

Vemos que A_1, A_2, ..., A_k son subárboles cuyas raíces son los nodos n_1, n_2, ..., n_k que a su vez son hijos del nodo n (raíz del árbol).

1.2. Definiciones

Vamos a ver varias definiciones sobre algunos conceptos de los árboles como (grado, camino, longitud, etc).

Grado: El grado es el número de nodos hijos de un nodo en concreto. El grado del árbol es el máximo de los grados de los nodos que lo componen.

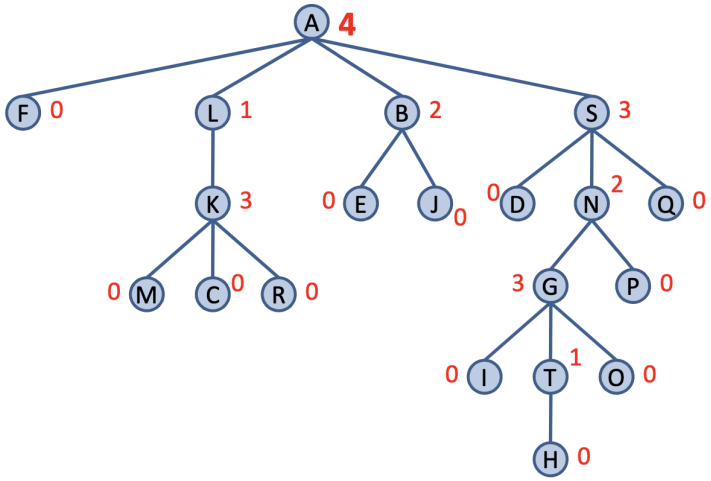


Figura 1.3: Grado de un árbol

Nota: El grado de un árbol no tiene porque ser siempre igual al grado del nodo raíz.

Camino y Longitud: El camino es una sucesión de nodos del árbol n_1, n_2, ..., n_k donde n_1 es el padre de n_i+1. El camino no tiene porque empezar en la raíz del árbol, es decir, se puede realizar un camino desde cualquier nodo.

La longitud de un árbol es el número de nodos - 1.

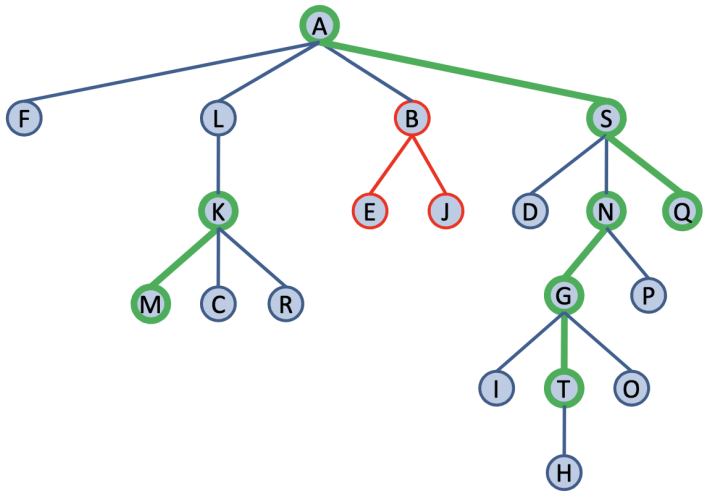


Figura 1.4: Camino y Longitud de un árbol
En verde encontramos tres ejemplos de caminos diferentes.

Ancestros y Descendientes: Si existe un camino entre dos nodos a y b cualesquiera, el nodo a será **antecesor/ancestro** del nodo b y el nodo b será **descendiente** del nodo a.

Si ese ancestro o descendiente son su padre o hijo(s)(respectivamente), se denominarán **ancestro propio** ó **descendiente propios**.

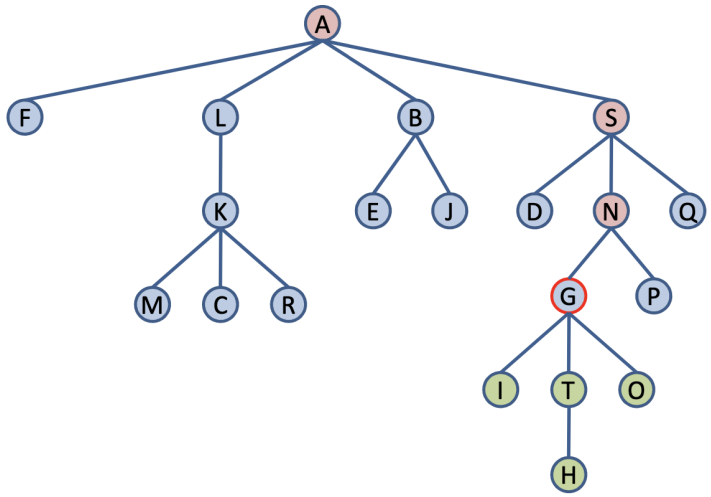


Figura 1.5: Ancestros y Descendientes de un nodo
Vemos que el nodo A tiene como descendientes propios los nodos (F, L, B, S) y como descendientes los hijos de estos anteriores.

1.3. Componentes de un árbol

Los árboles se componen de nodos raiz, hojas, subárboles, ramas, etcétera.

Raíz: Es el único nodo del árbol que no tiene ancestro.

Hoja: Es aquel nodo que no tiene ningún descendiente propio y por ende no tiene descendientes. **Hoja $\neq$ árbol nulo**.

Subárbol: Conjunto formado por un nodo cualquiera y sus descendientes.

Rama: Es el camino que termina en un nodo que es hoja.

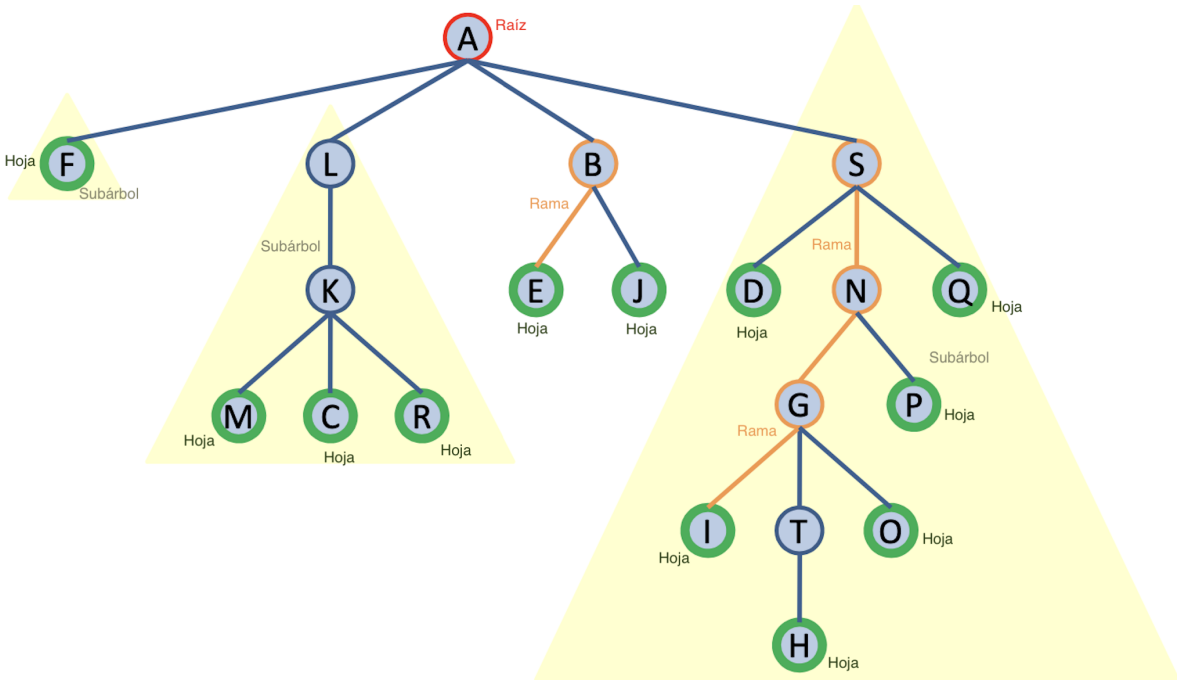


Figura 1.6: Componentes de un árbol

En el nodo F, tenemos dos puntos de vista:

- Respecto al súbarbol, es raíz sin descendientes propios (hoja).
- Respecto al árbol, es una hoja.

Altura y Profundidad:

- La Altura es la longitud de la rama más larga, la altura del árbol se corresponde con la altura del nodo raíz.
- La Profundidad es la longitud desde el único camino desde la raíz hasta ese nodo. También se denomina *nivel de nodo*.

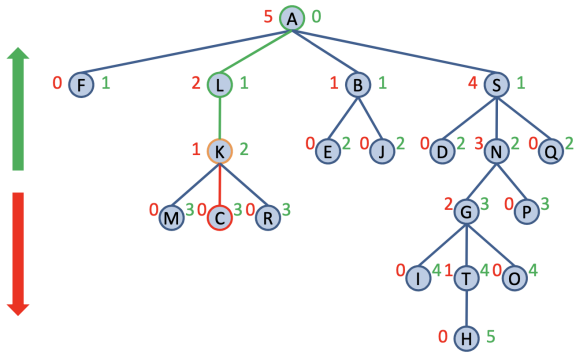


Figura 1.7: Ejemplo de altura y profundidad

En rojo vemos la altura y en verde la profundidad.

2. Árboles Binarios

Un **árbol binario** es aquel que cuyos nodos son como máximo de grado 2, es decir, pueden tener 0, 1 ó 2 hijos.

Sabiendo esto vamos a construir nuestros árboles binarios a partir de un árbol completamente vacío, insertando el nodo raíz del mismo y a partir de ahí iremos construyendo el árbol.

Para ello, tenemos que conocer y saber cuales son y que hacen las operaciones que permiten la construcción de árboles binarios, es decir, la **especificación del TAD árbol binario**.

2.1. Especificación del TAD árbol binario

Sea la especificación de operaciones del TAD Abin (árbol binario):

Constructor del árbol binario

Postcondición: Crea un árbol completamente vacío.

```
Abin();
```

Inserción de nodos

- **Inserción del nodo raíz**:
  Precondición: Árbol vacío.
  Postcondición: Inserta el nodo raíz cuyo contenido es 'e'.
  ```
  void insertarRaiz(const T& e);
  ```
- **Inserción de los nodos hijos**:
  Precondición: n es un nodo que existe en el árbol y no tiene hijo izquierdo/derecho.
  Postcondición: Inserta el nodo hijo(izquierdo o derecho) cuyo contenido es 'e'.
  ```
  void insertarHijoIzqdo(nodo n, const T& e);
  void insertarHijoDrcho(nodo n, const T& e);
  ```

Eliminación de nodos

Para poder eliminar un nodo éste debe de ser un **nodo hoja**, es decir, no tiene descendientes. Si no hacemos esto, no estaríamos cumpliendo la especificación del TAD.

Además no nos podemos eliminar a nosotros mismo, por tanto, esta operación solamente se podrá realizar a los hijos de nodo, si ese nodo no tiene hijos(es hoja) para eliminarlo tendremos que llamar a su padre y eliminarlo.

- **Eliminación del nodo raíz**:
  Precondición: Árbol no vacío y el nodo raiz es una hoja.
  Postcondición: Elimina el nodo raíz y deja el árbol vacío.
  ```
  void eliminarRaiz();
  ```

- **Eliminación de los nodos hijos**:
  Precondición: Árbol no vacío y el nodo n no es una hoja.
  Postcondición: Elimina el nodo hijo izquierdo o derecho del nodo n.
  ```
  void eliminarHijoIzqdo(nodo n);
  void eliminarHijoDrcho(nodo n);
  ```

Métodos observadores

- **Árbol vacío**:
  Postcondición: Devuelve `TRUE` si el árbol está vacío, si no devuelve `FALSE`.
  ```
  bool arbolVacio();
  ```
- **Obtener el elemento de un nodo**:
  Precondición: n es un nodo que existe en el árbol.
  Postcondición: Devuelve el elemento de ese nodo.
  ```
  const T& elemento(nodo n) const;
  T& elemento(nodo n);
  ```
- **Obtener el elemento de la raíz**:
  Postcondición: Devuelve el nodo raíz, si el árbol está vacío devuelve `NODO_NULO`.
  ```
  nodo raiz()const;
  ```
- **Obtener nodo padre de un nodo cualquiera**:
  Precondición: n es un nodo que existe en el árbol.
  Postcondición: Devuelve el padre del nodo n, si este no tiene devuelve `NODO_NULO`.
  ```
  nodo padre(nodo n)const;
  ```
- **Obtener Hijo Izquierdo de un nodo**:
  Precondición: n es un nodo que existe en el árbol.
  Postcondición: Devuelve el hijo izquierdo del nodo n, si este no tiene devuelve `NODO_NULO`.
  ```
  nodo hijoIzqdo(nodo n)const;
  ```
- **Obtener Hijo Derecho de un nodo**:
  Precondición: n es un nodo que existe en el árbol.
  Postcondición: Devuelve el hijo derecho del nodo n, si este no tiene devuelve `NODO_NULO`.
  ```
  nodo hijoDrcho(nodo n)const;
  ```

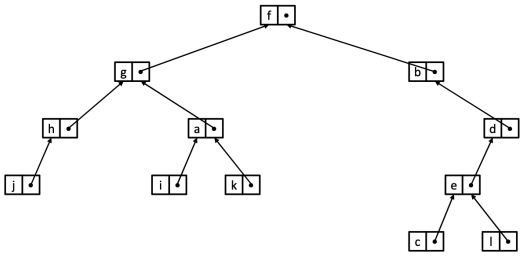
NOTA: `NODO_NULO` no es un elemento, si no que indica la no existencia de un nodo.

2.2. Implementaciones del TAD árbol binario

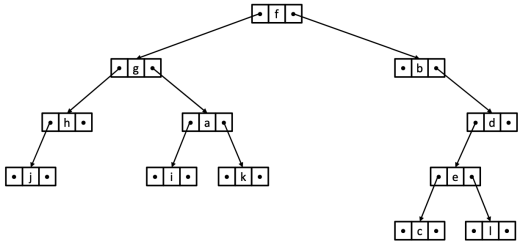
2.2.1. Implementación mediante celdas enlazadas

En esta implementación hacemos uso de memoria dinámica ya que no conocemos el número máximo de nodos que puede contener el árbol.

Vamos a ver dos propuestas de dicha implementación que son correctas pero muy ineficientes:

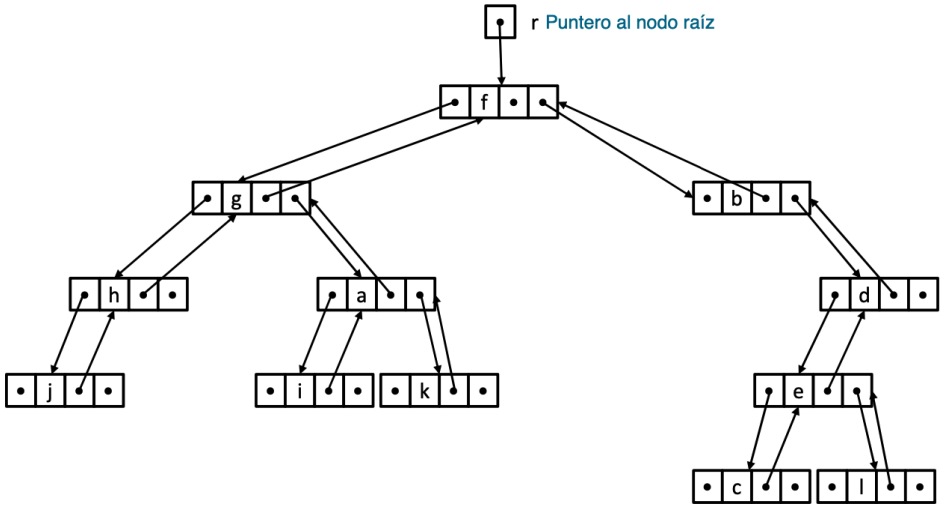


Esta implementación es muy costosa ya que es de orden $O(n)$ y además al no tener acceso directo al nodo padre ni a los hijos tenemos que acceder desde las hojas, por tanto, al tener que acceder desde las hoja tenemos que recorrer el árbol entero.



Esta implementación soluciona parcialmente el problema del acceso a los nodos (ya que si podemos acceder a los hijos) pero seguimos sin tener acceso al padre, por lo que su coste sigue siendo $O(n)$.

Nuestra implementación del TAD árbol binario mediante celdas enlazadas



Esta será nuestra implementación de dicho TAD, debido a que se corrigen todos los problemas descritos anteriormente, ahora tenemos acceso directo al nodo raíz y podemos acceder a cualquier nodo del árbol.

Finalmente, vemos que gracias a esta implementación todas las operaciones del TAD pasan a ser de orden $O(1)$.

En memoria se almacenará celdas cuyo contenido son los nodos (padre, hijo izquierdo y derecho) y el elemento del mismo (si ese nodo es nulo, no hay celda), por tanto, en la parte privada del TAD tendremos:

```
template <typename T> class Abin{
  public:
    //Métodos vistos en la especificación del TAD
  private:
    struct celda{
      T elto; //elemento del nodo
      nodo padre, hizq, hder; //nodos
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
      hizq(NODO_NULO),hder(NODO_NULO){}
    };
    nodo r; //nodo raíz del árbol.
};
```

Notas sobre el código de la implementación del TAD

En la parte pública de la clase encontramos:
`static const nodo NODO_NULO;`, que nos va a permitir indicar explicitamente la no existencia de un nodo.

A la hora de construir un Abin, este se debe de crear vacío, es decir, el nodo raíz es un nodo nulo:

```
template <typename T> inline Abin<T>::Abin():r(NODO_NULO){}
```

Cuando vayamos a insertar el nodo que será la raíz del árbol solo le pasamos el contenido del mismo, ya que al tener un puntero a la raíz no nos hace falta pasarle el nodo:

```
template <typename T> inline void Abin<T>::insertarRaiz(const T& e){
  assert(r == NODO_NULO); //Comprobamos la Precondición
  r = new celda(e); //Inserción del nodo raíz.
}
```

Si queremos insertar ya sea un nodo hijo izquierdo o derecho, tenemos que pasarle el nodo n (padre de ese nuevo nodo) y el elemento del nodo a insertar.

```
template <typename T> inline void Abin<T>::insertarHijoIzqdo(nodo n , const T& e){
  //Comprobamos Precondiciones
  assert(n == NODO_NULO);
  assert (n->hizq ==NODO_NULO);
  n->hizq = new celda(n,e); //Inserción del nodo hijo izquierdo de n.
}
```

NOTA: Análogamente para la inserción del hijo derecho de un nodo.

Si queremos eliminar un nodo, este debe ser una hoja. Por tanto, primero debemos comprobar si el nodo es una hoja. Si lo es, lo eliminamos dejándolo como un nodo nulo. Esto asegura que el nodo quede en un estado válido y que la dirección de memoria que ocupa la celda se libere correctamente.

```
template <typename T> inline void Abin<T>::eliminarHijoIzqdo(nodo n){
  //Comprobamos Precondiciones
```

```
  assert(n==NODO_NULO);
  assert(n->hizq!=NODO_NULO);//existe hijo izquierdo
  assert(n->hizq->hizq==NODO_NULO && n->hizq->hder==NODO_NULO); //Es Hoja

  //Lo explicado anteriormente
  delete n->hizq; //eliminamos
  n->hizq = NODO_NULO; //lo dejamos en estado válido
}
```

NOTA: No podemos hacer estos dos últimos paso a la inversa debido a que si lo dejamos como estado válido (nodo nulo), no podremos eliminar el puntero ni la dirección de memoria. *NOTA:* Análogamente para la eliminación del hijo derecho de un nodo.

Por otro lado, encontramos si queremos eliminar la raíz de un árbol, esta debe de ser una hoja, y por ende al ser eliminado dicho nodo el árbol quedaría vacío.

```
template <typename T> inline void Abin<T>::eliminarRaiz(){
  //Comprobamos Precondiciones
  assert(r == NODO_NULO);
  assert(r->hizq == NODO_NULO && r->hder == NODO_NULO);
  delete r;
  r = NODO_NULO;
}
```

Dentro de la parte privada del TAD encontramos dos métodos:

- `DestruirNodos(nodo& n)`; donde dado un nodo n elimina todos los descendientes del mismo, para poder eliminar al mismo. Esto nos simplifica mucho el destructor de la clase ya que al ser un método recursivo se ejecuta tantas veces hasta que llega a la condición de parada 'NODO_NULO'.

  ```
  template <typename T> void Abin<T>::destruirNodos(nodo& n){
   if(n!= NODO_NULO){
    destruirNodos(n->hizq);
    destruirNodos(n->hder);
    delete n;
    n=NODO_NULO;
   }//Para cuando n = NODO_NULO
  }
  ```

- `Copiar(nodo n)`; dado un nodo n cualquiera realiza una copia de dicho nodo y todos sus descendientes, donde ese nodo n es la raíz del nuevo árbol creado.

  ```
  template <typename T> typename Abin<T>::nodo Abin<T>::copiar(nodo n){
   nodo m = NODO_NULO; //creamos un nodo m aux
   if(n!= NODO_NULO){
    m = new celda(n->elto); //copiamos el padre (n).
    m->hizq = copiar(n->hizq); //copiamos subárbol izquierdo recursivamente
    if(m->hizq != NODO_NULO) m->hizq->padre = m;
    m->hder = copiar(n->hder); //copiamos subárbol derecho recursivamente
    if(m->hder != NODO_NULO) m->hder->pader = m;
   }
   return m;
  ```

```
}
```

2.2.2. Implementación vectorial del TAD árbol binario

A diferencia de la implementación mediante celdas enlazadas aquí si conocemos el número máximo de nodos del árbol, además haremos uso de un vector (como bien dice su nombre) de celdas que también almacenará el elemento y los nodos padre, hijo izquierdo y derecho. Véase *Figura 2.1: Representación de la implementación vectorial.*

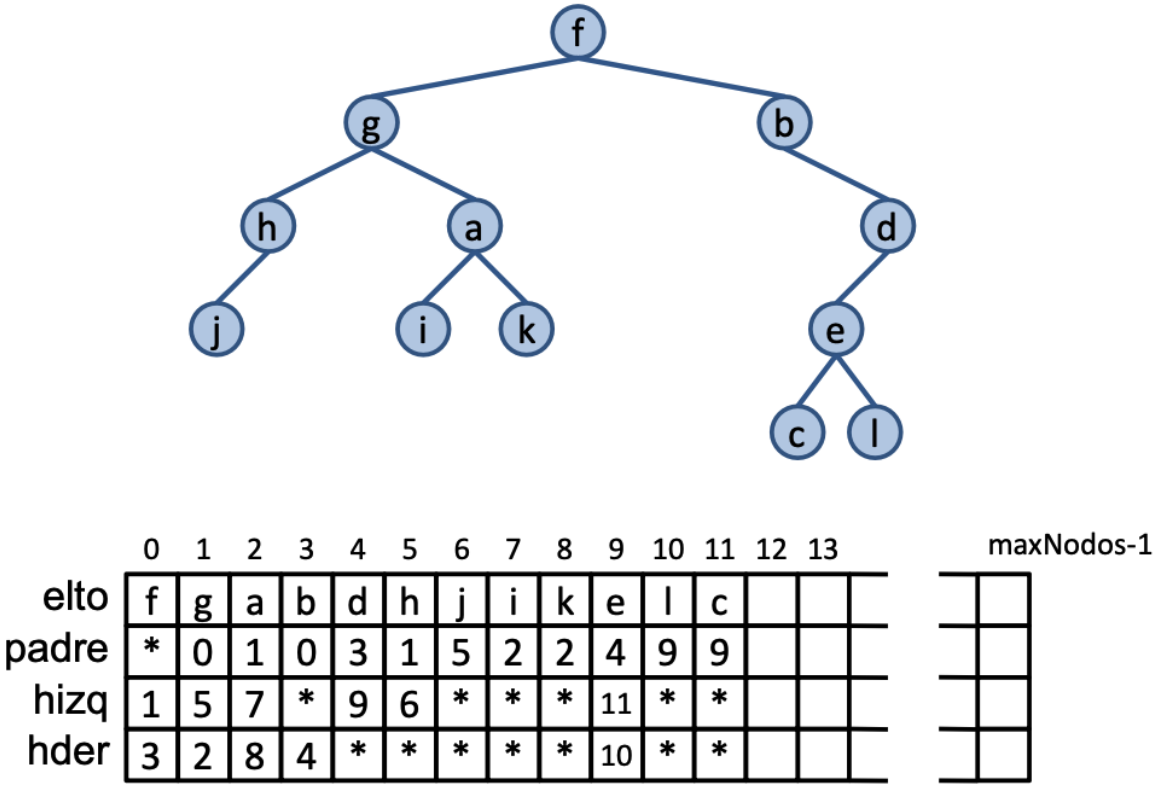


Figura 2.1: Representación de la implementación vectorial.

Ahora en la parte privada del TAD encontramos dos variables `numNodos`(número actual de nodos del árbol) y `maxNodos`(tamaño del árbol), ahora tenemos que tener en cuenta al primero para poder insertar y eliminar en coste unitario ($O(1)$), el segundo nos hará falta para poder crear el árbol ya que en este caso el constructor recibe por parámetro el numero máximo de nodos, es decir, `maxNodos` → `Abin(size_t maxNodos);`

Para que la eliminación de nodos sea de coste unitario debemos de mover el último nodo insertado a la última posición del vector.

Implementación de Inserción y Eliminación de los nodos

- **Inserción**: En la inserción de un nodo tenemos que tener en cuenta el número actual de nodos que hay en el árbol ya que este tiene que ser menor a `maxNodos`, esto nos ayudará a que la inserción sea de coste unitario, ya que accedemos a la última posición del vector (sin tener que recorrer todo el árbol):

```
template <typename T>
inline void Abin<T>::insertarHijoIzqdo(nodo n, const T& e){
  //Precondiciones
  assert(n>=0 && n <numNodos); //nodo valido
  assert(n->hizq == NODO_NULO); //n no tiene hijo izquierdo previo
  assert(numNodos < maxNodos); //árbol no lleno
  //Inserción del nodo
  nodos[numNodos].hizq = numNodos; //el nuevo nodo se inserta al final
  nodos[numNodos].elto = e; //almacenamos el elemento
  nodo[numNodos].hizq = NODO_NULO; //no tiene hijo izquierdo
  nodo[numNodos].hder= NODO_NULO; //no tiene hijo derecho
  numNodos ++; //incrementamos el número de nodos actuales en el vector.
}
```

- **Eliminación**: La eliminación de un nodo siempre va a ser de coste unitario, debido a que se elimina el hijo de un nodo n 'padre' (el cual tenemos acceso directo).

```
template <typename T> inline void Abin<T>::eliminarHijoIzqdo(nodo n){
  nodo hizqdo; //creamos un nodo auxiliar
  //Precondiciones
  assert(n>=0 && n < numNodos);
  hizqdo = nodos[n].hizq; //guardamos el hijo izquierdo del nodo n
  assert(hizqdo != NODO_NULO); //existe hijo izquierdo de n
  assert(nodos[hizqdo].hizq == NODO_NULO &&
    nodos[hizqdo].hder == NODO_NULO); // es Hoja

  //Comprobamos si el hijo izquierdo de n no es el último nodo insertado.
  if(hizqdo != numNodos - 1){
    //Movemos el último nodo a la posición del hijo izqdo
    nodos[hizqdo] = nodos[numNodos-1];
    //Actualizamos la posición del hijo en la padre del nodo movido(hizqdo)
    if(nodos[nodos[hizqdo].padre].hizq == numNodos - 1){
      nodos[nodo[hizqdo].padre].hizq = hizqdo;
    }
    else{
      nodos[nodos[hizqdo].padre].hder = hizqdo;
    }
    //Si el movido tiene hijos, solamente cambiamos el padre
    if(nodos[hizqdo].hizq!=NODO_NULO) //Si tiene hijo izquierdo
      nodos[nodo[hizqdo].hizq].padre = hizqdo;
    if(nodos[hizqdo].hder != NODO_ NULO)
      nodos[nodos[hizqdo].hder].padre = hizqdo;
  }
  nodos[n].hizq = NODO_NULO;
  --numNodos;
}
```

Es decir, movemos el último nodo insertado a la posición del nodo que vamos a eliminar para que podamos insertar siempre por el final (última posición del vector) haciendo que sea de coste unitario.

NOTA: Análogamente para el hijo derecho.

2.2.3. Implementación mediante vector de posiciones relativas

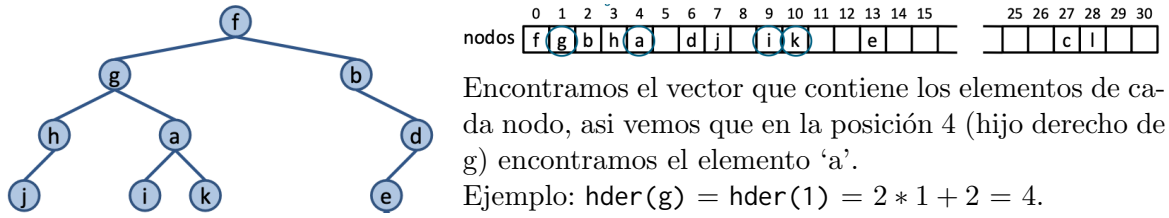
En esta implementación no nos hará falta almacenar punteros para poder acceder a los nodos Hijos y padre, es decir, vamos a poder acceder a ellos sin que estén almacenados.

Ahora la variable `maxNodos` se calculará mediante una fórmula donde estará implicada la altura h que coincide con el nivel de nodo.
Sea la formula: `maxNodos` $= \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$.

También podemos calcular las posiciones de los nodos:

- `padre(i)` $= (i-1)/2$ (división entera).
- `hizq(i)` $= 2i + 1$, los impares son hijos izquierdos.
- `hder(i)` $= 2i + 2$, los pares son hijos derechos.

Gracias a esto, solamente almacenamos un vector cuyo contenido es el elemento de cada nodo y cada posición del vector corresponde al nodo (calculado mediante las fórmulas anteriores). Véase (*Figura 2.2: Ejemplo de la representación con un vector de posiciones relativas*).
Para un árbol binario:

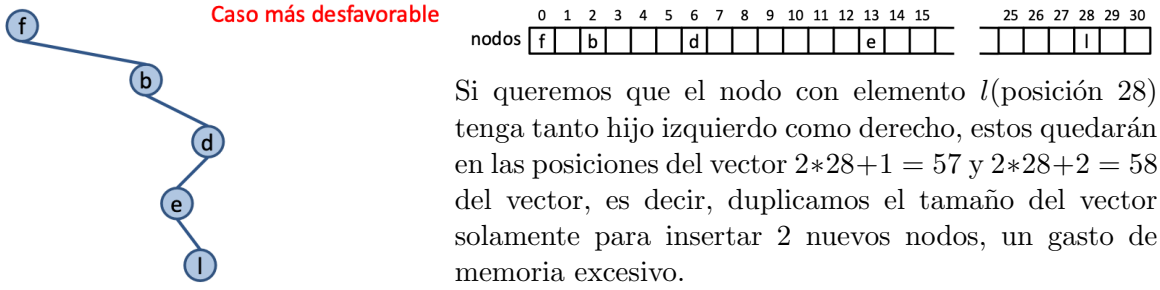


Encontramos el vector que contiene los elementos de cada nodo, asi vemos que en la posición 4 (hijo derecho de g) encontramos el elemento 'a'.
Ejemplo: `hder(g)` $=$ `hder(1)` $= 2 * 1 + 2 = 4$.

Figura 2.2: Ejemplo de la representación con un vector de posiciones relativas

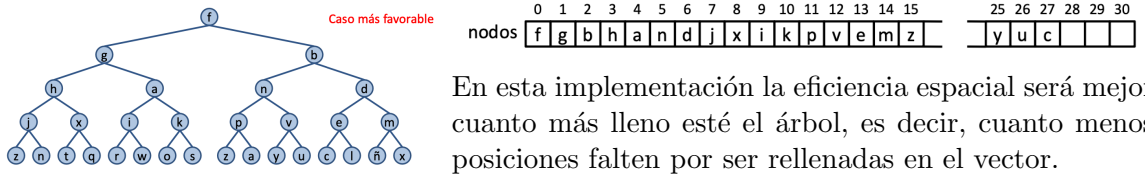
Esta implementación tiene una parte mala y es que encontramos un caso muy desfavorable, esto es debido a que el tamaño del vector depende de la altura del árbol, por tanto, si encontramos la ausencia de nodo en un nivel $n \leq h$ provocará $2^{h-n+1} - 1$ posiciones libres en el vector.

Veámoslo con un ejemplo:



Si queremos que el nodo con elemento l(posición 28) tenga tanto hijo izquierdo como derecho, estos quedarán en las posiciones del vector $2*28+1 = 57$ y $2*28+2 = 58$ del vector, es decir, duplicamos el tamaño del vector solamente para insertar 2 nuevos nodos, un gasto de memoria excesivo.

Por otro lado, tenemos un caso muy favorable y sucede cuando tenemos el árbol casi completo, es decir, no hay ausencia de nodos por nivel (todos los niveles tienen todos los nodos que les corresponde), lo denominamos como **árbol completo**.



En esta implementación la eficiencia espacial será mejor cuanto más lleno esté el árbol, es decir, cuanto menos posiciones falten por ser rellenadas en el vector.

Finalmente, hacemos uso del concepto de **elemento nulo** `T_ELTO_NULO`, no es un elemento o contenido del nodo, si no que al igual que NODO_NULO, indica la no existencia de elemento en el vector, es decir, una **flag** que nos indica las posiciones libres del vector (*Figura 2.3: Vector nodos con el uso de elemento nulo*).

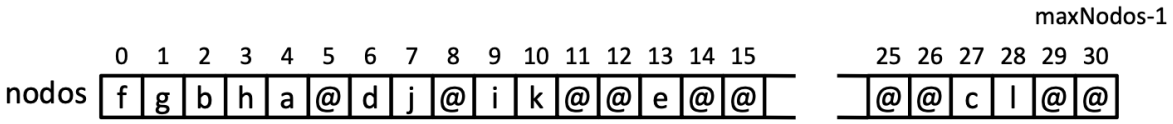


Figura 2.3: Vector nodos con el uso de elemento nulo

3. Árboles Generales

Un árbol general es aquel que cuyos nodos son de cualquier grado, es decir, pueden tener un número cualquiera de hijos, a diferencia de los árboles binarios que son de grado (0,1 ó 2).

Otra gran diferencia de los árboles generales frente a los árboles binarios es que en nuestro TAD no hablamos de hijos derechos, si no de hermanos derechos de un nodo, por tanto, tendremos nodos que son padre, hijo izquierdo y hermanos derechos, véase en (*Figura 3.1: Estructura de un árbol general*).

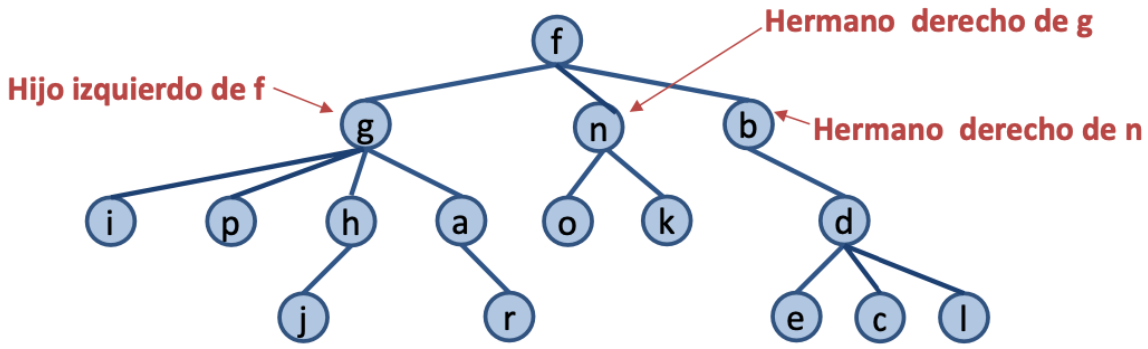


Figura 3.1: Estructura de un árbol general

La creación de un árbol general es igual que en los árboles binarios, partimos de un árbol general vacío e insertamos la raíz y a partir de ahí construimos el árbol insertando hijos izquierdos y hermanos derechos de esos nodos.

Al igual que en el caso de los árboles binarios vamos a ver cuales son y que hacen las operaciones mediante la **especificación del TAD árbol general**.

3.1. Especificación del TAD árbol general

Sea la especificación de las operaciones del TAD árbol general (Agen):

Constructor del árbol general

Postcondición: Crea y devuelve un árbol general vacío.

```
Agen();
```

Inserción de nodos

- **Inserción del nodo raíz**:
  Precondición: Árbol vacío.
  Postcondición: Inserta el nodo raíz cuyo contenido es 'e'.

```
void insertarRaiz(const T& e);
```

- **Inserción de los nodos hijos**:
  Precondición: n es un nodo que existe en el árbol.
  Postcondición: Inserta el elemento 'e' como hijo izquierdo de n, si ya tiene un hijo izquierdo el nodo n, este lo inserta y el anterior pasa a ser hermano derecho del nuevo.
  ```
  void insertarHijoIzqdo(nodo n, const T& e);
  ```
- **Inserción de los nodos hermanos**:
  Precondición: n es un nodo que existe en el árbol y no es raíz.
  Postcondición: Inserta el elemento 'e' como hermano derecho del nodo n.
  ```
  void insertarHermDrcho(nodo n,const T& e);
  ```

Eliminación de nodos

Al igual que en los árboles binarios, tenemos que comprobar que el nodo que vamos a eliminar sea una hoja, ya sea un hijo izquierdo o un hermano derecho de un nodo cualquiera.

- **Eliminación del nodo raíz**:
  Precondición: Árbol no vacío y nodo raíz es una hoja.
  Postcondición: Elimina el nodo raíz y deja el árbol vacío.
  ```
  void eliminarRaiz();
  ```
- **Eliminación de nodo hijo izquierdo**:
  Precondición: n es un nodo del árbol, tiene hijo izquierdo y este es Hoja.
  Postcondición: Elimina el hijo izquierdo del nodo n y su hermano derecho pasa a ser el nuevo hijo izquierdo.
  ```
  void eliminarHijoIzqdo(nodo n);
  ```
- **Eliminación de nodo hermano derecho**:
  Precondición: n es un nodo del árbol, tiene hermano derecho de n y este es Hoja.
  Postcondición: Elimina el hermano derecho de nodo n.
  ```
  void eliminarHermDrcho(nodo n);
  ```

Métodos observadores

- **Consultar elemento de un nodo**:
  Precondición: n es un nodo del árbol.
  Postcondición: Devuelve el contenido del nodo n.
  ```
  const T& elemento(nodo n)const;
  T& elemento(nodo n);
  ```
- **Consultar la raíz del árbol**:
  Postcondición: Devuelve el contenido del nodo raíz, no tiene, devuelve NODO_NULO.
  ```
  nodo raiz()const;
  ```
- **Consultar el padre de un nodo**:
  Precondición: n es un nodo del árbol.
  Postcondición: Devuelve el padre del nodo n, si no tiene, devuelve NODO_NULO.
  ```
  nodo padre(nodo n)const;
  ```
- **Consultar el hijo izquierdo de un nodo**:
  Precondición: n es un nodo del árbol.
  Postcondición: Devuelve el hijo izqdo del nodo n, si no tiene, devuelve NODO_NULO.
  ```
  nodo hijoIzqdo(nodo n)const;
  ```

- **Consultar el hermano derecho de un nodo**:
  Precondición: n es un nodo del árbol.
  Postcondición: Devuelve el hermano derecho del nodo n, si no existe, devuelve NODO_NULO.
  `nodo hermDrcho(nodo n)const;`

3.2. Implementaciones del TAD árbol general

3.2.1. Implementación vectorial mediante lista de hijos

En la implementación vectorial mediante una lista de hijos, cada nodo tiene almacenado una lista de nodos llamados hijos, la cual podemos recorrer para poder consultar sus diferentes hijos (mediante un bucle `while`).

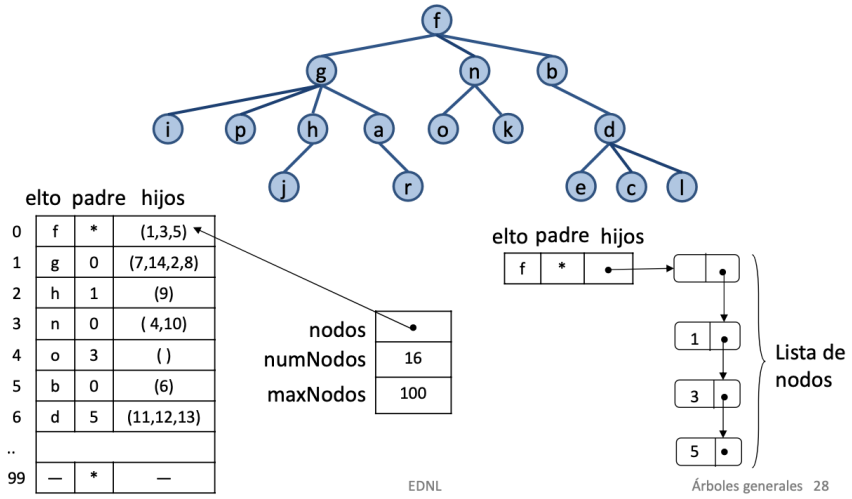


Figura 3.2: Representación de la implementación vectorial mediante lista de hijos.

En la parte privada de la implementación de dicho TAD vamos a encontrar el elementos que se almacena, el nodo padre y la lista de los diferentes hijos de cada nodo, todo almacenado en celdas:

```
template <typename T> class Agen{
  public:
    //Métodos vistos en la especificación del TAD
  private:
    struct celda{
      T elto; //elemento
      nodo padre
      Lista<nodo>hijos;
    };
    celda *nodos; //Vector de nodos
    size_t maxNodos; //Tamaño de vector (árbol)
    size_t numNodos; //Número de nodos en el árbol
};
```

3.2.2. Implementación mediante celdas enlazadas

Esta implementación es muy parecida a la implementación mediante celdas enlazadas en árboles binarios, la única diferencia es que en la parte privada no almacenamos hijos derechos, si no que almacenamos hermanos derechos, ya que en nuestro TAD, el concepto de hijo derecho ya no tiene sentido.

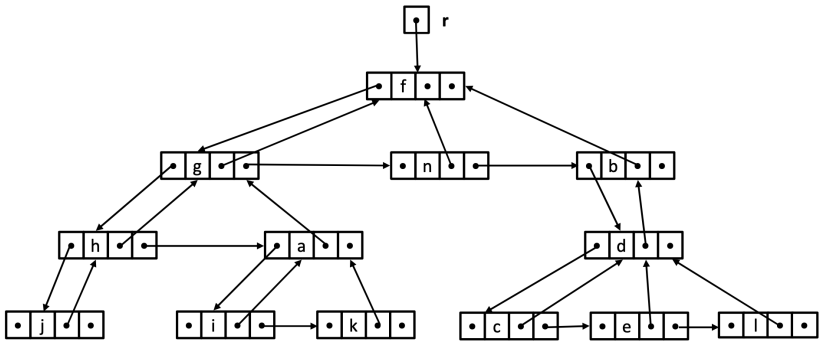


Figura 3.3: Representación de la implementación mediante celdas enlazadas

Por tanto, la parte privada del TAD nos quedaría:

```
template <typename T> class Agen{
  public:
    //Métodos vistos en la especificación del TAD
  private:
    struct celda{
      T elto;
      nodo padre,hizq,heder;
      celda(const T& e, nodo p = NODO_NULO):elto(e),padre(p),
        hizq(NODO_NULO),heder(NODO_NULO){}
    };
    nodo r; //nodo raíz del árbol
};
```

4. Árboles Binario de Búsqueda

Un **Árbol Binario de Búsqueda** o **ABB** es aquel que los nodos del subárbol izquierdo contiene elementos menores a su padre y el subárbol derecho contiene elementos mayores a su padre (*Figura 4.1: Representación de un árbol binario de búsqueda*).

Es importante saber que no pueden haber valores repetidos en este tipo de árboles, debido a que si no, no podremos comparar si esos elementos son menores o mayores (estrictos).

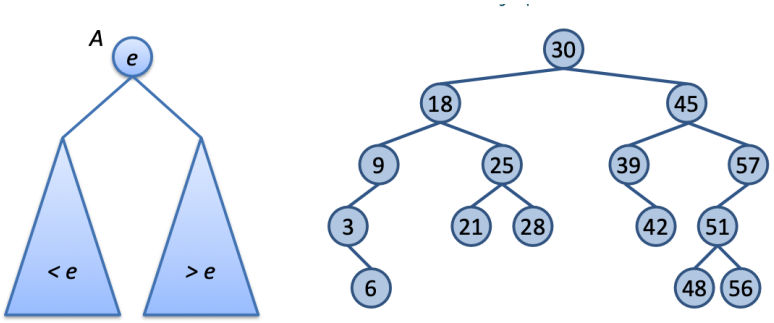
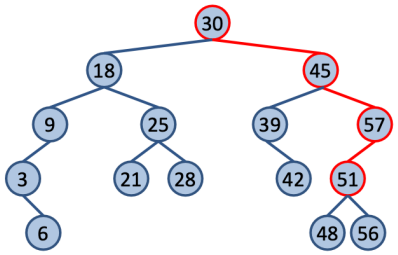


Figura 4.1: Representación de un árbol binario de búsqueda

Vemos que para un elemento 'e' (18), el subárbol izquierdo va a contener elementos menores a él (9,3,6) y el derecho elementos mayores (25,21,28).

Ahora, al saber que vamos a tener elementos 'ordenados' por su valor a la hora de buscar un elemento en el árbol pasamos de un coste $O(n)$ a un coste $O(log_2\ n)$ debido a que al buscar un elemento (comparando si es mayor o menor que el padre) nos quitamos todo el subárbol que no cumple esa condición, es decir, o vamos por el hijo izquierdo o por el hijo derecho de ese padre, por los dos no.

4.1. Ejemplo de búsqueda de un elemento en un ABB



Queremos buscar el nodo cuyo elemento es '51', para ello hacemos:
$51 > 30 \rightarrow$ Hijo derecho de 30 $\rightarrow$ subárbol derecho
$51 > 45 \rightarrow$ Hijo derecho de 45 $\rightarrow$ subárbol derecho
$51 < 57 \rightarrow$ Hijo izquierdo de 57 $\rightarrow$ subárbol izquierdo
$51 = 51 \rightarrow$ hemos encontrado el nodo cuyo elemento es 51.

Es importante saber que el tiempo de búsqueda depende de la estructura de ramificación del propio árbol. En este ejemplo, vemos que es un coste de $O(log_2\ n)$, pero se puede dar el caso que el ABB es una lista de elementos por tanto, el coste sería $O(n)$.

En este caso, vemos que tenemos un árbol cuya ramificación se asemeja a una lista de elementos, por tanto, al no poder descartar ningún subárbol a la hora de buscar (por ejemplo el nodo con elemento 18) tendremos que recorrer toda la rama hasta llegar a ese nodo.

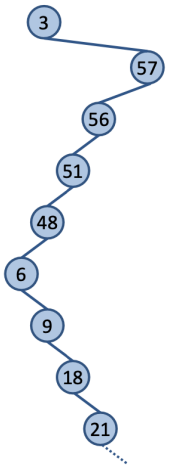


Figura 4.2: Ejemplo de ABB de coste $O(n)$

.

4.2. Especificación del TAD árbol binario de búsqueda

Anteriormente, tanto en los árboles binarios como generales trabajábamos con nodos. Ahora en los árboles binarios de búsqueda vamos a trabajar con subárboles dependiendo de un valor 'e'.

Constructor del árbol binario de búsqueda

Postcondición: Crea y devuelve un ABB vacío.

```
Abb();
```

Inserción de elementos en el ABB

Postcondición: Si 'e' no pertence al árbol, lo inserta, en caso contrario el árbol no se modifica.

```
void insertar(const T& e);
```

Eliminación de elementos del ABB

Postcondición: Si 'e' se encuentra en el árbol, lo elimina, en caso contrario, el árbol no se modifica.

```
void eliminar(const T& e);
```

Métodos observadores de un ABB

- **Obtener el subárbol**:
  Postcondición: Si 'e' pertence al árbol devuelve el subárbol (cuya raíz es 'e'), si no, devuelve un subárbol vacío.

  ```
  const Abb& buscar(const T& e)const;
  ```

- **Estado vacío**:
  Postcondición: Devuelve `True` si el árbol está vacío, si no, `False`.
  ```
  bool vacio()const;
  ```
- **Obtener subárbol izquierdo**:
  Precondición: Árbol no vacío.
  Postcondición: Devuelve el subárbol izquierdo.
  ```
  const Abb& izqdo()const;
  ```
- **Obtener subárbol derecho**:
  Precondición: Árbol no vacío.
  Postcondición: Devuelve el subárbol derecho.
  ```
  const Abb& drcho()const;
  ```

4.3. Implementación del TAD árbol binario de búsqueda

4.3.1. Implementación de un ABB mediante una estructura dinámica recursiva

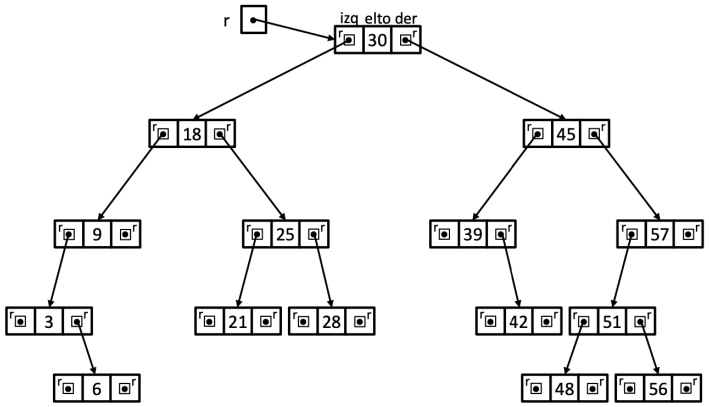


Figura 4.3: Representación de la implementación dinámica recursiva

Vamos a hacer uso de esta implementación debido a que el método más 'importante' de este TAD es el método `buscar()` (que nos devuelve el subárbol donde el elemento 'e' es la raíz). Este método se realiza mediante llamadas recursiva y veremos como lo hace.

En la parte privada de TAD vamos a tener que almacenar el elemento y los dos subárboles (izquierdo y derecho).

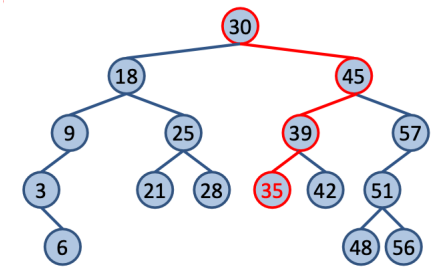
```
template <typename T> class Abb{
  public:
    //Métodos vistos en la especificación del TAD
  private:
    struct arbol{
      T elto; //elemento
      Abb izq, der; //subárboles
      arbol(const T& e):elto(e),izq{},der{} {}
    };
    arbol *r;
};
```

Implementación de la obtención del subárbol

Ahora, vamos a ver como se implementa y que hace el método `buscar()`, mencionado anteriormente.

```
template <typename T>
const Abb<T>& Abb<T>::buscar(const T& e)const{
  if(r == nullptr)
    return *this; //Árbol vacio, no existe elemento
  else if(e < r->elto) //si es menor que el de la raiz, subárbol izquierdo
    return r->izq.buscar(e);
  else if(r->elto < e) //si es mayor, subárbol derecho
    return r->der.buscar(e);
  else
    return *this; //encontrado en la raiz
}
```

Inserción de un elemento en un ABB




```
template <typename T>
void Abb<T>::insertar(const T& e){
  if(r==nullptr) //árbol vacio
       r = new arbol(e);
//se inserta en el subárbol izq.
  else if(e < r->elto)
       r->izq.insertar(e);
//se inserta en el subárbol drch.
  else if(r->elto < e)
       r->der.insertar(e);
}
```

Ejemplo: Queremos insertar en el árbol anterior el elemento '35' (indicado en rojo), para ello la traza a seguir (según el código dado) sería:

$35 > 30 \rightarrow$ Hijo derecho de 30 $\rightarrow$ subárbol derecho.
$35 < 45 \rightarrow$ Hijo izquierdo de 45 $\rightarrow$ subárbol izquierdo.
$35 < 39 \rightarrow$ Hijo izquierdo de 39 $\rightarrow$ subárbol izquierdo.

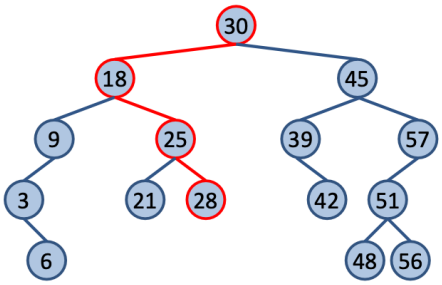
Como el nodo cuyo valor es '39' no tiene subárbol izquierdo y 35 es menor que él, insertamos dicho valor como 'hijo izquierdo' del nodo con valor 39.

NOTA: Si encontramos que dicho valor ya se encuentra en el árbol no hacemos nada, ya que no podemos tener valores repetidos en el árbol.

Eliminación de un elemento en un ABB

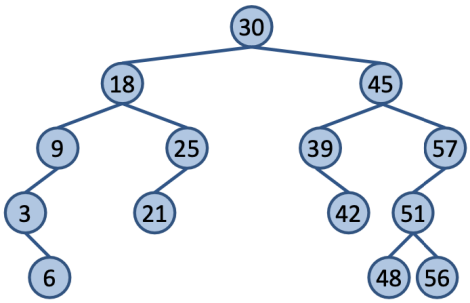
A la hora de eliminar un elemento de un ABB no tenemos como precondición que el nodo que contiene a ese elemento sea una hoja, por tanto, encontramos 3 casos(cuando es hoja, tiene un hijo y tiene ambos hijos):

- **Caso 1: Eliminación de un nodo hoja:**

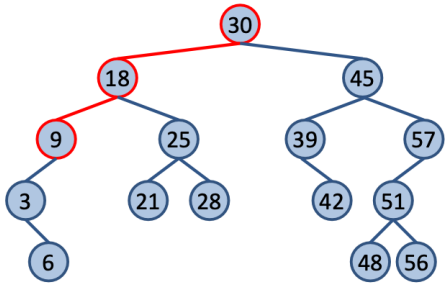


Queremos eliminar el nodo con elemento '28'(indicado en rojo), para ello:
$28 < 30 \rightarrow$ Hijo izquierdo de 30 $\rightarrow$ subárbol izquierdo.
$18 < 28 \rightarrow$ Hijo derecho de 18 $\rightarrow$ subárbol derecho.
$25 < 28 \rightarrow$ Hijo derecho de 25 $\rightarrow$ subárbol derecho.
Como $28 = 28$, lo eliminamos.

Como resultado quaría:

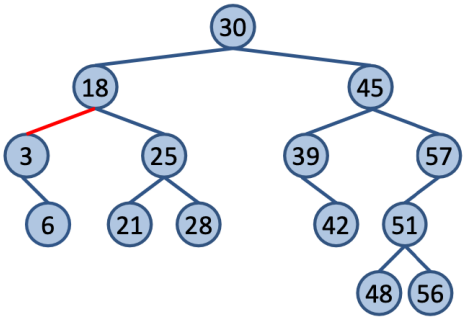


- **Caso 2: Eliminación de un nodo con un hijo:**
Este caso es un poco diferente, ya que el nodo a eliminar va a tener o hijo izquierdo o derecho.



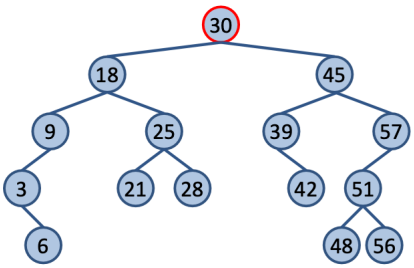
Queremos eliminar el nodo con elemento '9'(indicado en rojo), para ello:
$9 < 30 \rightarrow$ Hijo izquierdo de 30 $\rightarrow$ subárbol izquierdo.
$9 < 18 \rightarrow$ Hijo derecho de 18 $\rightarrow$ subárbol derecho.
Como $9 = 9$, lo eliminamos y su hijo pasa a ser el nuevo hijo izquierdo del 'padre de 9', es decir el nodo con valor '18'.

Como resultado quaría:



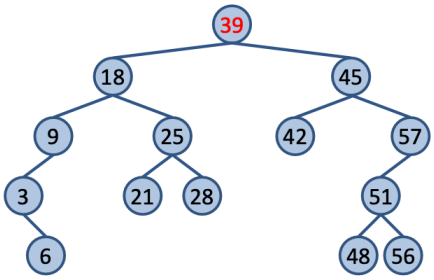
- **Caso 3: Eliminación de un nodo con ambos hijos:**

  Este caso es parecido al caso 2, con la única diferencia que ahora el nodo a eliminar tiene tanto hijo izquierdo como derecho. Esto se soluciona buscando a los candidatos para ocupar su posición en el árbol, estos pueden ser el **mayor elemento del subárbol izquierdo** o el **menor elemento del subárbol derecho**.



Como vemos, queremos eliminar el nodo con valor '30' (tiene dos hijos 18 y 45). Por tanto, los candidatos válidos son el nodo con mayor valor del subárbol que tiene como raíz '18' ó el nodo con menor valor del subárbol que tiene como raíz '45'.
Hemos escogido el nodo con menor valor del subárbol derecho que es '39', pero este tiene un hijo derecho '42' que pasará a ser hijo izquierdo del nodo con valor '45'.

Como resultado quaría:



Código de la eliminación de elementos de un ABB:

```
template <typename T> void Abb<T>::eliminar(const T& e){
  if(r!=nullptr){ //Caso 1: es hoja
    if(e < r->elto)
      r->izq.eliminar(e);
    else if(r->elto < e)
      r->der.eliminar(e);
  }
  else if(!r->izq.r && !r->der.r) //raíz es hoja
    delete r;
    r == nullptr;
  else if(!r->der.r) //Caso 2: Solo tiene un hijo (izquierdo)
    arbol *a = r->izq.r;
    r->izq.r = nullptr;
    delete r;
    r = a;
  else if(!r->izq.r) //Caso 2: Ánalogo para el derecho
    arbol *a = r->der.r;
    r->der.r = nullptr;
    delete r;
    r = a;
  else //Caso 3: Tiene ambos hijos
    r->elto = r->der.borrarMin();
}
```

5. Árboles de Búsqueda Equilibrados

La inserción como la eliminación de elementos de un ABB determinan su grado de equilibrio, por ende la sucesión inserción o eliminación de elementos del mismo puede alterar el equilibrio del árbol.

Sabemos que el coste de las operaciones `buscar()`, `insertar()` y `eliminar()` depende de la altura del árbol (el caso más desfavorable vimos que era $O(n)$ 'lista').

Este equilibrio nos permite garantizar un coste proporcional a la mínima altura posible del árbol, consiguiendo un coste mínimo de $O(log_2\ n)$, por tanto, es muy importante que a la hora de insertar o eliminar elementos mantengamos el árbol lo más equilibrado posible.

Encontramos dos tipos de árboles equilibrados los **árboles AVL** y los **árboles ARN**.

5.1. Árboles AVL

Un **árbol AVL** es un tipo de ABB equilibrado, la resta de la altura del subárbol derecho menos la altura del subárbol izquierdo del mismo nodo se denomina **factor de equilibrio de nodo** (esto garantiza el coste logarítmico).

Decimos que un árbol está equilibrado cuando su factor de equilibrio es -1, 0 ó 1 en todos sus nodos.

Como hemos comentado previamente, la inserción y la eliminación de elementos pueden verificar la condición de equilibrio del árbol y si es necesario, reequilibrarlo mediante la rotación de nodos (intercambios).

En el peor caso, el tiempo de inserción, eliminación y búsqueda están en coste $O(log\ n)$.

Diferencia entre AVL y No AVL

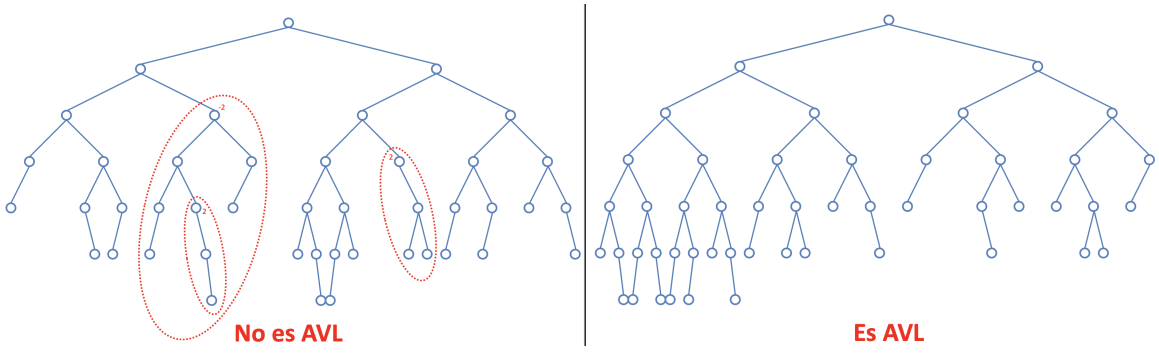
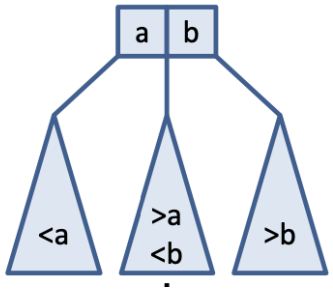


Figura 5.1: Diferencia entre No AVL y AVL

5.2. Árboles ARN

Un **árbol ARN** ó **árbol rojinegro** es un ABB que representa un *árbol B* de orden 3 (árbol 2-3), el cual está equilibrado (tiene todas las hojas en el mismo nivel).



Si tenemos k claves, entonces tenemos $k + 1$caminos, por tanto, si tenemos un árbol 2-3, tendremos 2 clave y 3 caminos.
Si tenemos 2 claves el desequilibrio va a ser 1.
Siempre va a existir un nodo negro (valor más grande del par) pero puede existir o no un nodo rojo (valor más pequeño del par).

Esta representación permite una implementación más sencilla de inserciones y eliminaciones, ya que solamente comparamos los valores de cada nodo y no tenemos que hacer uso de estructuras complejas como los punteros.

Todo nodo **raíz** se denomina nodo negro, mientras que todo **hijo izquierdo** de esa raíz se denomina nodo rojo. Otra cosa a tener en cuenta es que todos los hijos de un nodo **rojo** son nodos **negros**, véase en la (*Figura 5.2: Ejemplo de árbol rojinegro*).

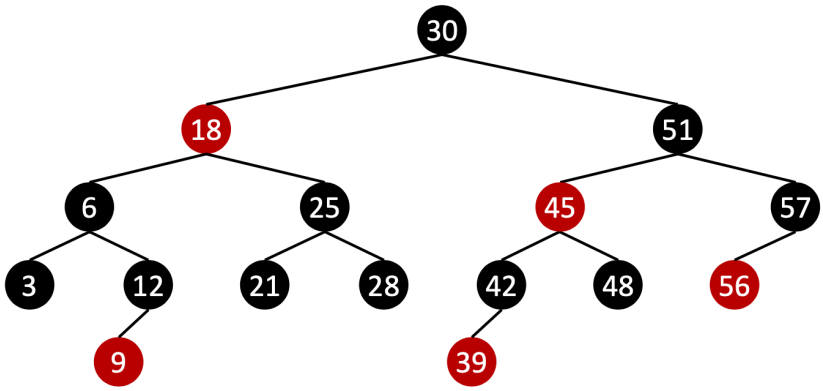


Figura 5.2: Ejemplo de árbol rojinegro

Inserción y Eliminación

Se realizan igual que en un ABB (Árbol Binario de Búsqueda) pero deben de representar correctamente el árbol 2-3 subyacente, es decir, tenemos que realizar cambios de color y rotaciones (operaciones internas) en los nodos.

Operaciones internas

Encontramos las operaciones de rotaciones y repintado las cuales tendremos que usar a la hora de insertar y eliminar nodos:

- **Rotación** → Recolocan los nodos del árbol conservando la propiedad de orden/búsqueda.

- **Repintado** → Cambia de color a los nodos.

La efiencia en los árboles rojinegro dependen de la altura del mismo, es decir, para un árbol rojinegro con n nodos tiene una **altura** $h \leq 2log_2\ n$ y dado que las operaciones de **inserción**, **búsqueda** y **eliminación** tienen un peor tiempo de ejecución proporcional a la altura del árbol, entonces estás operaciones tienen un coste de $O(log\ n)$.

5.3. AVL vs ARN

Los árboles AVL están *más equilibrados* que los árboles ARN, ya que el desequilibrio máximo de los AVL es 1, además que en los AVL las búsqueda de elemento son *más rápidas*.
Por otro lado, los árboles ARN son *más flexibles* por lo que las inserciones y eliminaciones de elementos son más rápidas.

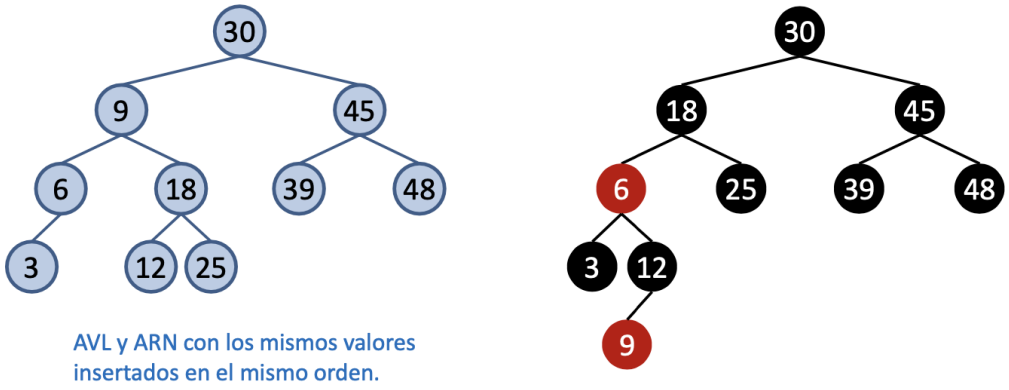
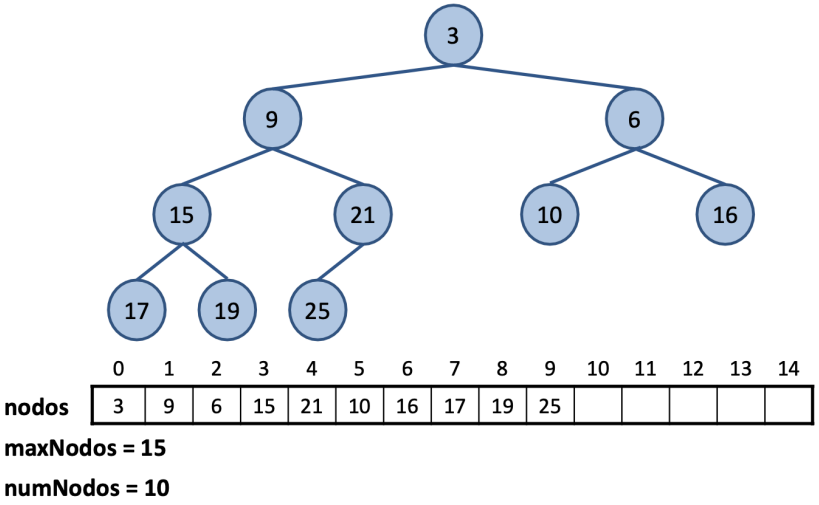


Figura 5.3: Diferencia entre AVL y ARN

Sin embargo, estas diferencias comentadas, en la práctica respecto al rendimiento global es inapreciable.

6. Árboles parcialmente ordenados

Un **árbol parcialmente ordenado** o **APO**, es un árbol completo (árbol que tiene todos sus niveles llenos menos el último que faltan nodos por la derecha) en el que el *valor de cualquier nodo es menor o igual que el de todos sus descendientes.*



6.1. Operaciones básicas

Cuando trabajamos con APOs podemos realizar varias operaciones como **acceso al mínimo valor**, **inserción** y **eliminación**.

- El mínimo valor se puede obtener en un tiempo de$O(1)$, debido a que el mínimo valor siempre será la **raíz** del propio árbol.
- Encontramos la **propiedad de completitud** que implica que la altura menor posible del APO de n nodos es de $h = log_2\ n$.
- También existe la **propiedad de orden** del APO que permite efectuar inserciones y eliminaciones en el mismo en un tiempo $O(h)$ (en el peor caso), y en un tiempo $O(log_2\ n)$ en el mejor y caso promedio.

6.2. Especificación del TAD árbol parcialmente ordenado

En los árboles parcialmente ordenados vamos a encontrar dos métodos `hundir()` y `flotar()`.

Constructor del árbol parcialmente ordenado

Precondición: maxNodos > 0
Postcondición: Crea y devuelve un APO de tamaño MaxNodos vacío.

```
Apo(size_t maxNodos);
```

Inserción de elementos en el APO

Precondición: APO no lleno.
Postcondición: Inserta el elemento en su posición correcta.
`void insertar(const T& e);`

Eliminación de elementos en el APO

Precondición: APO no vacío.
Postcondición: Elimina la raíz reordenando el APO. `void suprimir();`

Métodos observadores de un APO

- **Obtener la raíz**:
  Precondición: APO no vacío.
  Postcondición: Devuelve el elemento que está en la raíz.
  `const T& cima()const;`
- **Obtener estado del árbol**:
  Postcondición: Devuelve `True` si el árbol está vacío, si no, `False`. `bool vacio()const;`
- **Obtener el padre**:
  Precondición: APO no vacío.
  Postcondición: Devuelve el padre del nodo n.
  `nodo padre(nodo n)const;`
- **Obtener el hijo izquierdo**:
  Precondición: APO no vacío.
  Postcondición: Devuelve el hijo izquierdo del nodo n.
  `nodo hIzq(nodo n)const;`
- **Obtener el hijo derecho**:
  Precondición: APO no vacío.
  Postcondición: Devuelve el hijo derecho del nodo n.
  `nodo hDer(nodo n)const;`

Métodos hundir y flotar de un APO

- **Hundir un nodo**:
  Este método irá de la mano a la hora de eliminar el nodo raíz, reordenando el árbol hasta que dicho nodo sea una hoja y lo podamos eliminar.

  Podemos tener 3 casos a la hora de eliminar la raíz del APO:

  - **Caso 1: Un nodo** → Al tener un solo nodo (raíz), si la eliminamos el árbol quedará **vacío**.
  - **Caso 2: Dos nodo** → Ahora tenemos tanto la raíz como su hijo izquierdo, por tanto, si eliminamos la raíz, su hijo izquierdo pasará a ser la nueva raíz del APO.

- **Caso 3: Más de dos nodo** → Si queremos eliminar una raíz que tiene más de dos nodos, el último nodo insertado será el que ocupe la posición del raíz y luego hundimos dicho nodo para poder cumplir la propiedad de orden.

```
void hundir(nodo n);
```

- **Flotar un nodo**:
  A la hora de insertar un nodo, éste se añade a la última posición del vector y luego para que se cumpla la propiedad de orden tenemos que ir 'subiendo' dicho nodo hasta que encuentre su sitio.

```
void flotar(nodo n);
```

6.3. Implementación del TAD árbol parcialmente ordenado

Vamos a hacer uso de la implementación mediante un **vector de posiciones relativas** ya que estos son muy útiles y eficientes cuando trabajamos con árboles completos.

La parte privada del TAD quedaría:

```
template <typename T> class Apo{
  public:
    //Métodos vistos en la Especificación del TAD.
  private:
    typedef size_t nodo; //Indice del vector.
    size_t maxNodos; //Tamaño del vector (árbol).
    size_t numNodos; //Número de nodos (último nodo del árbol)
    T* nodos; //Vector de nodos
    //Métodos privados de la clase
    nodo padre(nodo n)const;
    nodo hIzq(nodo n)const;
    nodo hDer(nodo n)const;
    void hundir(nodo n);
    void floar(nodo n);
};
```

Inserción de elementos en un APO

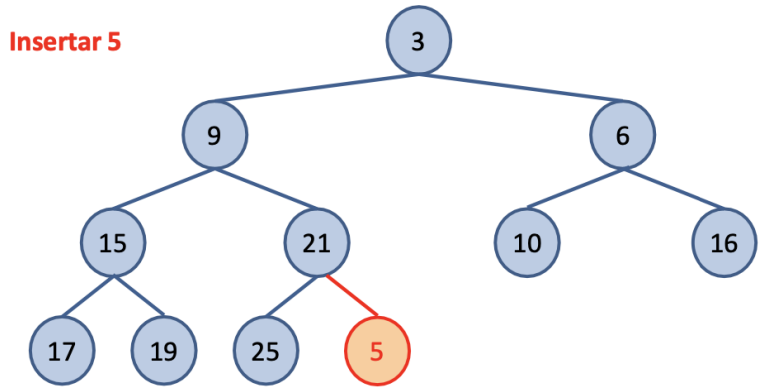


Figura 6.1: Ejemplo de inserción 1.

Queremos insertar el valor '5' en nuestro APO el cual no está vacío, por tanto, vamos a insertarlo en la última posición del vector de posiciones relativas que equivale al hijo derecho del nodo con valor '21'. A continuación vamos flotando dicho nodo hasta que se cumpla la propiedad de ordenación. Tenemos como resultado que el nodo con valor '5' es el nuevo hijo

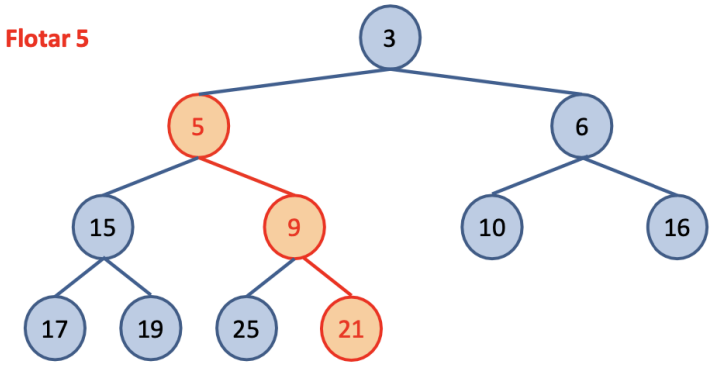


Figura 6.2: Ejemplo de inserción 2.

izquierdo del nodo raíz, y por tanto, vemos que todos los nodos están ordenados.

Código de Inserción y Flotar

Código de insertar:

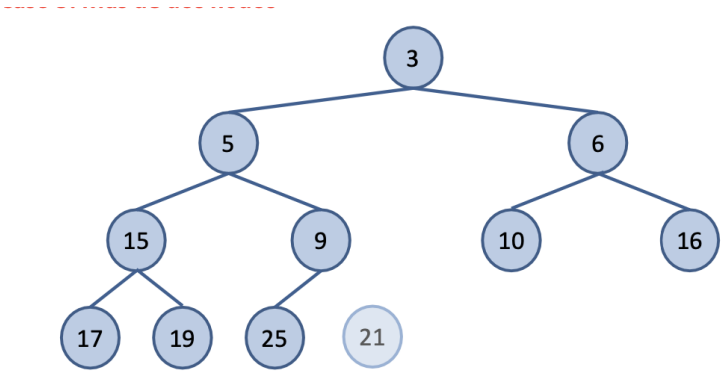
```
template <typename T>
void Apo<T>::insertar(const T& e){
  assert(numNodos < maxNodos);
  //insertamos en la última posición.
  nodos[numNodos] = e;
  //Llamamos al método flotar
  if(numNodos > 1)
    flotar(numNodos-1);
```

```
}
```

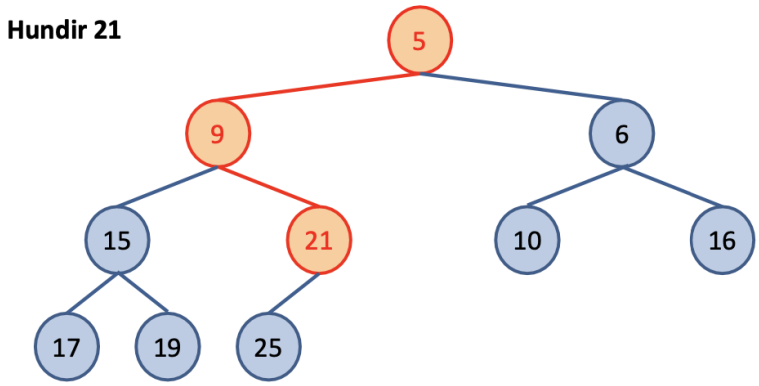
Código de flotar:

```
template <typename T>
void Apo<T>::flotar(nodo n){
  //Guardamos el contenido del nodo
  T e = nodos[n];
  //recorremos los nodos intercambiandolos
  while(n>0 && e < nodos[padre(n)]){
    nodos[n] = nodos[padre(n)];
    n = padre(n);
  }
  nodos[n] = e;
}
```

Eliminación de elementos en un APO



Vemos que queremos realizar la eliminación del nodo raíz con contenido '3', por tanto, el último nodo del árbol '21', ocupará la posición del nodo raíz sobrescribiendo su contenido y por ende tendremos que hundir dicho nodo para que el APO siga ordenado.



COmo resultado tenemos que el nodo con valor '21' ahora es padre del nodo con valor '25' y el nuevo nodo raíz es el nodo con valor '5' que previamente era el hijo izquierdo del nodo con valor '3' (la raíz anterior).

Código de suprimir:

```
template <typename T>
void Apo<T>::suprimir(){
  assert(numNodos > 0);
  if(--numNodos > 0){
    nodos[0]=nodos[numNodos];
    if(numNodos > 1)
      hundir(0);
  }
}
```

Código de hundir:

```
template <typename T>
void Apo<T>::hundir(nodo n){
 bool fin = false;
 T e = nodos[i];
 while (hIzq(i) < numNodos && !fin){
 nodo hMin;
 if (hDer(i) < numNodos && nodos[hDer(i)]
      < nodos[hIzq(i)])
 hMin = hDer(i);
 else
 hMin = hIzq(i);
 if (nodos[hMin] < e){
 nodos[i] = nodos[hMin];
 i = hMin;
}
 else
   fin = true;
}
 nodos[i] = e;
}
```

Finalmente, como conclusión sacamos de que un **APO** no es un **AVL**, ya que ambos tienen propositos diferentes, siendo el objetivo principal de los AVLs la búsqueda de elementos lo más rapido posible y en los APOs su uso en algoritmos de ordenación y en el uso de colas con prioridad.

7. Tablas de Dispersión

Las **tablas de dispersión** o **tablas hash** son aquellas que almacenan elementos de un conjunto mediante el uso de una **función hash**, que dado un elemento devuelve su posición exacta en la tabla que se almacena.

El tiempo de búsqueda de dicho elemento es de coste $O(1)$, ya que tenemos acceso directo a dichos elementos de un conjunto, esto es así ya que el tiempo que se tarda en calcular la función hash es constante.

Una idea propuesta es que cada elemento que se almacena se compone de una clave, estas claves son números entero y dependen del tamaño del vector M, donde el rango de las mismas es $[0, M-1]$, por tanto, si queremos acceder al elemento 'i', será de la manera $h(i) = i$.

También podemos encontrar problemas con dicha idea, ya que pueden que no estén acotadas (demasiado grandes para el tamaño del vector), la claves no son de tipo entero..

7.1. Funciones Hash

Las **funciones hash** transforman una clave de un cierto tipo (número, cadena, fecha, etc) en una **dirección** o **índice** de la propia tabla hash, $h : C \rightarrow [0, M-1]$, donde C es el conjunto de posibles claves.

Debido a esto encontramos las **colisiones** entre claves que se produce cuando a dos claves diferentes le corresponden como resultado el mismo valor de función hash. Si esto ocurre decimos que ambas son **claves sinónimas**.

Por tanto, una **función hash perfecta** es aquella que no produce ninguna colisión entre todas las claves posibles $C \leq M$ (el cardinal de C es menor al tamaño del vector).

Pero esto es algo teórico, ya que en la práctica el número de claves posibles es mayor al tamaño del vector $C > M$, por tanto, es inevitable que se produzcan colisiones entre claves.

Propiedades que encontramos en las buenas funciones hash:

- Eficiencia temporal → Rapida de calcular.
- Eficiencia espacial → Función sobreyectiva(todas las posciones de la tabla corresponden a una clave).
- Distribución uniforme → Reduce la posibilidad de colisiones entre claves.

Esto no es algo sencillo y presenta algunas dificultades.

Una función hash la podemos construir en dos partes:

1. Definir una función que hash **H(x)** que transformen las claves a números enteros.
2. Trasladar cualquier número a una dirección obtenida mediante el calculo de su módulo respecto al tamaño del vector → $h(x) = H(x)\ mod\ M$.

7.2. Resolución de colisiones

Como hemos comentado anteriormente, en la práctica las colisiones son algo *inevitable*, para hacerles frente podemos definir diferentes tipos de hashings (hashing cerrado, abierto o encadenamiento mezclado).

Tenemos que diferenciar que la **sinonímia** $\rightarrow$ **colisión**, pero la **colisión** $\nrightarrow$ **sinonímia**.

7.3. Hashing Cerrado

Cuando se produce una colisión, se busca una nueva posción libre en la tabla. Aquí encontramos una **secuencia de exploración** para poder localizar dicha posición libre de la tabla. Esto se realiza mediante una serie de **M funciones hash** $\{h_0, h_1, h_2, ..., h_{M-1}\}$, donde h_0 es la función hash original.

Exploración lineal

Tenemos que $h_i = (h_0 \ + \ i \ * \ \alpha) \ mod \ M$, donde la distancia entre celdas $\alpha \in \mathbb{N}$ y es primo relativo con M.

Este tipo de hash produce **agrupamientos primarios** $\rightarrow$ posiciones ocupadas $\alpha = 1$, esto aumenta el tiempo de posteriores colisiones debido a que se puede producir un gran bloque de claves, lo que denominamos **degradación muy rápida de la eficiencia de las operaciones**.

Exploración cuadrática

Ahora tenemos $h_i = (h_0 \ + \ i^2) \ mod \ M$, ahora la distancia entre celdas aumenta exponencialmente.

Ahora encontramos que se producen **agrupamientos secundarios** de claves sinónimas debido a que la secuencia de exploración desde h_0(+1, +4, +9, +16,..), además estos agrupamientos secundarios se entrelanzan pero no se unen para formar otros mayores, es decir, son menos perjudiciales para la eficiencia temporal que los agrupamientos primarios.

Exploración aleatoria

Este es otro tipo de hahing cerrado, donde $h_i = (h_0 \ + \ x_i) \ mod \ M$, donde x_i se obtiene mediante una generador de números aleatorios (GNA).

El GNA tiene que generar dichos números cuya distribución debe de ser uniforme, es decir, que tengan la misma probabilidad para permitir recorrer todas las posciones.

Este tipo de hashing cerrado evita los agrupamientos de claves sinónimas si usamos una semilla diferente para cada clave, de modo que tendremos secuencias de exploración diferentes.

Hashing doble

En este tipo de hashing encontramos una segunda función hash $h`$, quedando la función hash como $h_i = (h_0 + i * h`) \ mod \ M$.

Este método es parecido a la *exploración lineal*, pero con una diferencia: la distancia entre celdas exploradas es variable, debido a que ahora depende de incrementos de $h`$ posiciones.

Como ahora cada clave tiene una secuencia de exploración diferente, se **evita la colisión de claves sinónimas**.

Esta nueva función hash secundaria tiene que cumplir ciertas condiciones:

- $h`(x) \neq 0$, para cualquier clave 'x', de lo contrario no sirve para resolver colisiones, ya que si no todas las funciones de exploración coincidirían con la función hash original h_0.
- $h` \neq h_0$. Si ambas son iguales, la secuencia de exploración sería la misma, por ende, habría colisiones entre claves.
- $h`(x)$ y M deben de ser primos relativos para garantizar que se recorre toda la tabla.

Es importante saber distinguir entre casillas libres y ocupadas en la tabla, así como las borradas.

Operaciones

- **Búsqueda:** Seguir con la secuencia de exploración hasta:
  - Encontramos la cable buscada (búsqueda con éxito).
  - Encontramos una casilla *libre* (búsqueda sin éxito).
  - Se ha recorrido todas las casillas (búsqueda sin éxito).
- **Inserción:** Seguir con la secuencia de exploración hasta:
  - Encontramos una casilla que está *libre* y realizar la inserción.
  - Encontramos la clave a insertar (ya existe en la tabla).
  - Se ha recorrido toda la tabla (concluimos con que está llena).
- **Eliminación:** Seguir con la secuencia de exploración hasta:
  - Encontramos la clave a eliminar y realizamos la eliminación (marcar la casilla como *borrada*).
  - Encontrar una casilla libre o se ha recorrido toda la tabla (clave no existe).

La eficiencia de las operaciones se degrada rápidamente a medida de que se va llenando de claves la tabla.

7.4. Hashing abierto

Una **tabla hash abierta** (también denominadas con *direccionamiento cerrado* o *encadenamiento separado*), cada posición de la misma es un cubículo con una estructura que permite alojar varios elementos cuyas claves son **sinónimas**.

La estructura más sencilla es '**tabla de M lista enlazada**', cuya longitud media es n/M, donde n es el número de claves insertadas. Esto determina el tiempo medio de búsqueda y la función hash distrubuirá uniformemente las claves para que no se acumulen en pocos cubículos.

Si seleccionamos una M demasiado grande, las listas serán bastante cortas y, en consecuencia, **no supone una mejora importante de la eficiencia de la búsqueda mantener las listas ordenadas** o **sustituirlas por árboles de búsqueda**.

7.5. Encadenamiento mezclado

Es una *combinación del hashin cerrado y abierto*, donde las claves se insertan en casillas libres de la tabla (hashing cerrado), pero aquellas que la función hash lleva a casillas ocupadas se enlazan formando listas dentro de la tabla (hashin abierto).

Estas listas pueden contener una mezcla de **claves sinónimas** y **no sinónimas**, a diferencia del hashin abierto.

La búsqueda con esta combinación **es más rápida** que las búsquedas que se realizan en el *hashing cerrado*, puesto a que solo hay que buscar el fragmento de la lista que comienza en la casilla $h(k)$, para localizar la clave k.

Por el contrario, la búsqueda en el *encadenamiento mezclado* no son tan rápidas como en el *hashing abierto*, ya que estas listas contienen tanto claves no sinónimas como sinónimas, por ende, son más grandes.

7.6. Eficiencia

- **Mejor caso**: No hay colisiones, hay una clave en cada casilla de la tabla, por tanto, las búsquedas son de orden $O(1)$.
- **Peor caso**: Todas la claves de la tabla tienen colisiones, debido a esto las búsquedas son de orden $O(n)$.
- **Caso promedio**: Si asumimos que la distribución de claves es uniforme, la búsqueda depende del factor de carga ($\alpha = n/M$) y la distribución de las propias claves.

7.7. Comparación entre métodos

Si el factor de carga se mantiene por **debajo del umbral**, el número de comparaciones medio estará acotado por una constante y esto hace que la **búsqueda** tenga un coste promedio de $O(1)$.

El *hashing cerrado* requiere prever el número máximo de elementos a almacenar (degradación muy rápida de la eficiencia y rendimiento), mientras que *hashing abierto* requiere un mayor tamaño de la tabla (aunque el uso de memoria es menor, ya que no usamos punteros).

La exploración lineal es el método **menos eficiente**, con una curva exponencial (*véase en la Figura 7.1: Gráfica comparativa entre diferentes tipos de hashing*). Aunque esta aplicación puede ser útil en algunos casos, ya que su implementación no es complicada.

La exploración cuadrática tiene una eficiencia equiparable a las exploración aleatoria o *hashing doble*, esto confirma que los **agrupamientos secundarios** no afectan tanto en la eficiencia como los **agrupamientos primarios**.

Como vemos el rendimiento del *Encadenamiento mezclado* se aproxima al del *hashing abierto*, el cual es el **más eficiente** peor acosta de consumir más memoria debido al uso de punteros de las listas enlazadas.

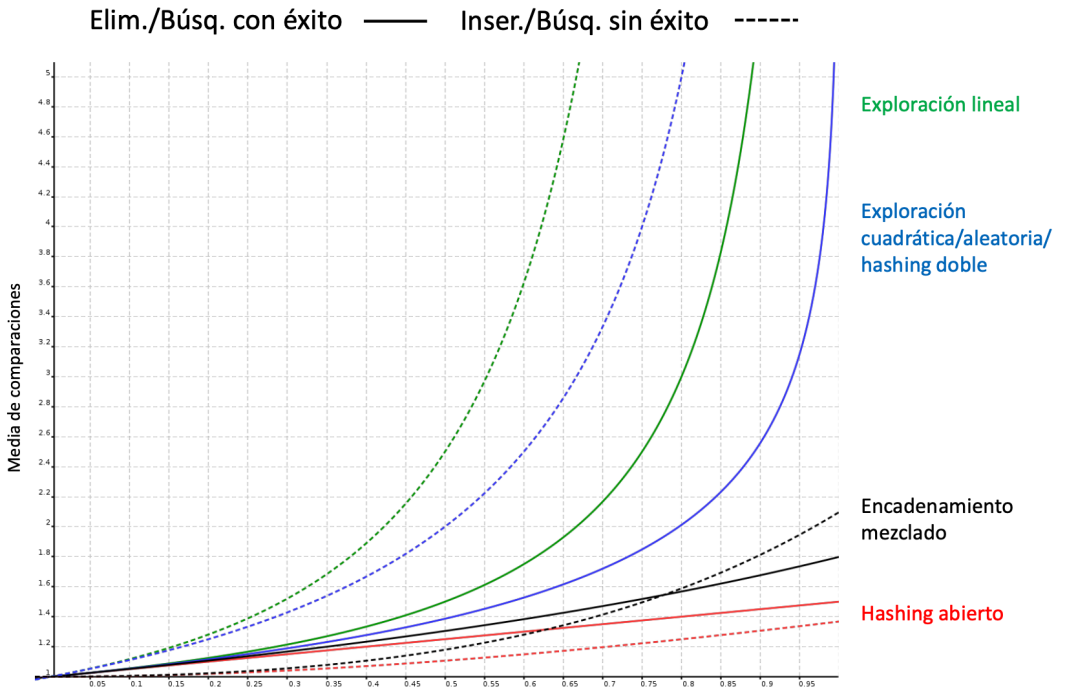


Figura 7.1: Gráfica comparativa entre diferentes tipos de hashing

7.8. Redimensionamiento y Rehashing

Cuando una tabla tiene un factor de carga alto, es decir, está muy llena, se realiza un **redimensionamiento** de la misma (se crea una tabla más grande y se reinsercionan de nuevo los elementos de la anterior) o un **rehashing** (se reinsercionan los elementos en la misma tabla para eliminar las casillas que están marcadas como borradas).

Esto hace que se restaure la eficiencia de la tabla.

7.9. Comparación con árboles de búsqueda

Sabemos que las tablas hash ofrecen un tiempo de búsqueda del orden $O(1)$ si controlamos el facto de carga, pero un coste de $O(n)$ para el peor caso.

Como vimos en el tema de árboles de búsqueda, estos ofrecen un tiempo de búsqueda del orden $O(log\ n)$, con la ventaja de que estos árboles son dinámicos, es decir, no tenemos que prever el número máximo de elementos a almacenar.

Otra ventaja es que los **árboles de búsqueda** ofrecen un conjunto de operaciones más amplios, donde destaca el *acceso a los elementos en orden*, gracias a esto podemos acceder los mínimos y máximos valores respecto a otro o buscar el minimo valor o máximo del árbol.

Como conclusión tenemos que si queremos realizar búsquedas rápidas la elección correcta es el uso de **tablas de dispersión o hash** cuando se elige una buena función hash y controlamos adecuadamente el factor de carga. Sin embargo, si necesitamos tener los elementos ordenados, haremos uso de un **árbol de búsqueda**.

8. Introducción a los grafos

En este tema vamos a ver los diferentes tipos de grafos que existen para resolver problemas. Veremos sobre los grafos (*Dijkstra, Floyd, Warshall, Kruskal, Prim*, etc), donde cada uno tiene un proposito diferente, es decir, resuelven diferentes problemas y es importante saber cual resuelve cual a la hora de trabajar con ellos.

Para ello, primero vamos a ver que es un grafo y de que se compone.

8.1. Definición de grafo