Todo sobre Estructura de Datos No Lineales

Índice general

1. Introducción a Árboles **4**
1.1. Definición de árbol 4
1.2. Definiciones 5
1.3. Componentes de un árbol 7

2. Árboles Binarios **9**
2.1. Especificación del TAD árbol binario 9
2.2. Implementaciones del TAD árbol binario 11
2.2.1. Implementación mediante celdas enlazadas 11
2.2.2. Implementación vectorial del TAD árbol binario 14
2.2.3. Implementación mediante vector de posiciones relativas 16

3. Árboles Generales **18**
3.1. Especificación del TAD árbol general 18
3.2. Implementaciones del TAD árbol general 20
3.2.1. Implementación vectorial mediante lista de hijos 20
3.2.2. Implementación mediante celdas enlazadas 21

4. Árboles Binario de Búsqueda **22**
4.1. Ejemplo de búsqueda de un elemento en un ABB 22
4.2. Especificación del TAD árbol binario de búsqueda 23
4.3. Implementación del TAD árbol binario de búsqueda 24
4.3.1. Implementación de un ABB mediante una estructura dinámica recursiva 24

5. Árboles de Búsqueda Equilibrados **28**
5.1. Árboles AVL 28
5.2. Árboles ARN 29
5.3. AVL vs ARN 30

6. Árboles parcialmente ordenados **31**
6.1. Operaciones básicas 31
6.2. Especificación del TAD árbol parcialmente ordenado 31
6.3. Implementación del TAD árbol parcialmente ordenado 33

7. Tablas de Dispersión **37**
7.1. Funciones Hash 37
7.2. Resolución de colisiones 38
7.3. Hashing Cerrado 38
7.4. Hashing abierto 40
7.5. Encadenamiento mezclado 40
7.6. Eficiencia 40
7.7. Comparación entre métodos 40
7.8. Redimensionamiento y Rehashing 41
7.9. Comparación con árboles de búsqueda 42

8. Introducción a los grafos **43**
8.1. Definición de grafo 43

8.2. Definiciones 44
8.3. Representaciones de grafos 45

9. Caminos de coste mínimo **47**
9.1. Algoritmo de Dijkstra 47
9.1.1. Ejemplo algoritmo Dijkstra 47
9.1.2. Código del algoritmo Dijkstra 50
9.2. Algoritmo de Floyd 51
9.2.1. Código del algoritmo de Floyd 52
9.3. Algoritmo de Warshall 54
9.3.1. Código del algoritmo de Warshall 54

10. Árboles generadores de coste mínimo **56**
10.1. TAD Partición 56
10.1.1. Especificación del TAD Partición 57
10.1.2. Implementación del TAD Partición 57
10.2. Algoritmo de Kruskal 64
10.2.1. Ejemplo algoritmo Kruskal 64
10.3. Algoritmo de Prim 66

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 suceción 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);
}
```

<u>Ejemplo:</u> 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 quedaría:

- **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 quedaría:

- **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 quedaría:

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 suceción 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** $\rightarrow$ 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 <u>sobreyectiva</u>(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

Un **grafo** 'G = (V,A)', consta de un conjunto de **vértices** o **nodos** (V) y de un conjunto de **aristas** o **arcos** (A) que define una relación banaria en V. Por tanto, podemos definir como que una arista es un par de vértices cualesquiera $(v, w) \in A$.

Si cada arista $(v, w) \in A$ es un par ordenado, es decir si, (v, w) y w, v no son equivalentes, entonces el grado es **dirigido** y la arista resultante del par (v, w) se representa con una *flecha* indicando la dirección de v a w (*Figura 8.1: Grafo dirigido*).

Figura 8.1: Grafo dirigido

NOTA: Decimos que el vértice v es **incidente** sobre el vértice w y que w es **adyacente** a v.

Por otro lado, si (v, w) y w, v son **equivalentes** o **simétrico**, el grafo será **no dirigido**, y por tanto, la arista se representa como un *segmento* entre v y w, véase en la (*Figura: 8.2 Grafo no dirigido*).

Figura 8.2: Grafo no dirigido

Cada arista del grafo puede tener asociada un valor asociado, llamado*peso*, que indica el **coste** (tiempo, distancia, etc) de ir desde el vértice v al vértice w. Un grafo el cual tenga un peso asociado a cada arista se le denomina **grafo ponderado**.

Figura 8.3: Grafo ponderado

8.2. Definiciones

Vamos a ver varias definiciones que debemos de saber cuando vayamos a trabajar con grafos.

Grado: El grado en un grafo no dirigido es el **número de arcos**. Si el grafo es dirigido se obtiene a partir de la diferencia entre **grado de entrada** y **grado de salida**, es decir, *arcos incidentes* − *arcos adyacentes*.

Camino: Es una suceción de vértices de un grafo $n_1, n_2, ..., n_k$, tal que (n_i, n_{i+1}) es una arista para $1 \leq i < k$. La **longitud** del camino es el número de arcos que comprende, es decir $k-1$, si el grafo no es ponderado. Si el grafo es ponderado, la longitud será la *suma* de todos los pesos de las arista que lo componen.

Camino simple: Es el camino cuyos arcos son todos distintos, además si todos los vértices son distintos → **camino elemental**.

Ciclo: Es el camino en el que coincide los vértices inicio y final. Si el camino es simple → **ciclo simple** y si el camino es elemental → **ciclo elemental**.

Grafo Conexo: Es un grafo **no dirigido** en el que al menos hay un camino entre cualquier par de vértice.

Figura 8.4: Diferencia entre grafo conexo y no conexo

<u>Grafo fuertemente conexo:</u> Es un grafo **dirigido** en el que hay la menos un camino entre cualquier par de vértices. Si un grafo dirigido no es fuertemente conexo, pero el grafo no dirigido *subyacente* es conexo, diremos que el grafo es **débilmente conexo**.

Figura 8.5: Ejemplos de grafos

<u>Grafo completo:</u> Es aquel que existe una arista entre cualquier par de vértices, si es en ambos sentido (segmento), es diremos que es completo y dirigido.

Figura 8.6: Ejemplos de grafo completo y dirigido

<u>Subgrafo:</u> Dado un grafo $G = (V, A)$, diremos que el subgrafo $G^{`} = (V^{`}, A^{`})$ si $A^{`}$ sólo contiene todas las aristas de A que unen los vértices de $V^{`}$.

8.3. Representaciones de grafos

1. **Matriz de adyacencia**

   Dado un grafo $G = (V, A)$ con n vértices, se define la matriz de adyacencia asociada a G como una matriz M_{nxn}, donde $M_{i,j} = 1$ si $(i, j) \in A$ y $M_{i,j} = 0$ si $(i, j) \notin A$.

   Si G es un grafo no dirigido, M será una matriz simétrica ya que $(i, j) = (j, i)$ para cualesquiera vértices i, j.

2. **Matriz de costes**

   Dado un grafo $G = (V, A)$ con n vértices, se define la matriz de coste asociada a G como una matriz C_{nxn}, donde

$Ci, j = p$ si $(i, j) \in A$, donde p = peso asociado a (i, j)
$Ci, j = peso_ilegal$ si $(i, j) \notin A$, (peso_ilegal = valor no válido como peso de un arco). Si no hay camino directo entre los dos vértices, se indicará con el valor **infinito**.

3. **Listas de adyacencia**

   Asociamos a cada vértice i del grafo una lista que almacena todos los vértices adyacentes al mismo (i).

Ventajas e inconvenientes

- Las **matrices de adyacencia** y **costes** son muy eficientes para comprobar si existe una arista entre dos vértices.
- Pueden desaprovechar una gran cantidad de memoria si el grafo no es completo.
- La representación mediante **listas de adyacencia** aprovechan mejor el espacio de memoria y son *poco eficientes* para determinar si existe una arista entre dos vértices del grafo.
- Estas estructuras de datos **no admiten** la inserción y eliminación de vértices, pero la estructura **lista de listas de adyacencia** si se permite la adición y eliminación de vértices.

9. Caminos de coste mínimo

En este capítulo veremos algunos de los algoritmos que trabajan con grafos para obtener el o los caminos cuyo coste sean mínimo.

Encontraremos los algoritmos de Dijkstra, Floyd y Warshall, donde veremos que hacen y como realizan el cálculo de los caminos con coste mínimos.

9.1. Algoritmo de Dijkstra

El *algoritmo de Dijkstra* calcula los caminos de coste mínimo entre un **origen** y **todos los demás vértices** del grafo G.

```
template <typename tCoste>
vector<tCoste>
  Dijkstra(const GrafoP<tCoste>& G, typename GrafoP<tCoste>::vertice origen,
  ↪  vector<typename GrafoP<tCoste>::vertice& P);
```

Como salida tenemos:

- **Vector de costes mínimos** de tamaño `G.numVert()`.
- **Vector de vértices** (P) de tamaño `G.numVert()`, tal que `P[i]` es el vértice anterior a i en el camino de costes mínimo (origen, i).

NOTA: Un grafo no dirigido, es un grafo dirigido pero con flechas en ambas direcciones, por tanto, si tenemos un algoritmo que funciona para grafos dirigidos también serán válidos para grafos no dirigidos.

NOTA: **Dijkstra Inverso** es igual que Dijkstra pero con la diferencia que en vez de tener un origen vamos a tener un destino.

9.1.1. Ejemplo algoritmo Dijkstra

Encontramos un vector de booleanos, que indica si hemos pasado o no por dicho vértice. Estos vértices en el vector 'S' se inicializan a `False` y partimos desde el origen a todos los vértices del grafo.

	0	1	2	3	4
S	T	F	F	F	F

El algoritmo se acaba cuando hemos pasado por todos los vétices del grafo, es decir, el vector de booleano está lleno con valores `True`.

Como hemos comentado previamente, el algoritmo devuelve dos cosas: el vector de vértices 'camino' y el coste mínimo 'distancia'.

Distancia

	0	1	2	3	4
D	0	10	∞	30	100

Es el **coste mínimo** de ir desde el origen hasta cualquier otro vértice del grafo.
En este algoritmo siempre partimos del vértice origen.
Hacemos uso de un vector y este se inicializa con los costes directos entre el origen y los demás vértices del grafo.

Camino

	0	1	2	3	4
P	0	0	0	0	0

Es un **vector de vértices**, cuando el camino es directo se inicializa a '0'.
Nos permite recuperar el camino, que son los vértices desde el origen hasta el destino por los que ha pasado.

Gracias a este algoritmo vamos a poder reducir el coste de ir de un vértice a otro, calculando los costes de ir de un vértice a otro y quedándonos con el mínimo.

Tenemos el grafo y la matriz de costes siguientes:

Matriz de Costes

	0	1	2	3	4
0	∞	10	∞	30	100
1	∞	∞	50	∞	∞
2	∞	∞	∞	∞	10
3	∞	∞	20	∞	60
4	∞	∞	∞	∞	∞

numVert = 5

Vemos que tenemos el grafo anterior y queremos ir desde el vértice 0 'origen' hasta el vértice 4 'destino'.

Para ello, en el vector de costes mínimos 'D' tenemos los valores obtenidos previamente arriba, pero vamos a realizar una optimización del mismo y además encontramos el vector de vértices 'P', que está inicializado a 0 (hay camino directo) ó *infinito* (no hay camino directo).

Los vectores comentados anteriormente ya tienen almacenado tanto los costes como los vértices por donde pasan, pero no siempre es eficiente hacer uso de los caminos directos, por eso vamos a realiar un par de modificaciones.

Vemos que no hay camino directo entre el origen y el vértice '2', por tanto, para acceder a él lo haremos desde el vértice '1' y su coste será el de ir desde origen a '1' ($0 \rightarrow 1$) más el de '1' a '2'($1 \rightarrow 2$), además en el vector de vértices en la 'casilla' del número '2' pondremos '1' (vértice que accede a él) y en el vector de booleanos podemos que ya se ha pasado por el vértice '0'(origen) y '1'.

```
1ª) i = 1
  Min {D[1],D[2],D[3],D[4]}= Min {10,∞,30,100}= 10
  w = 1
```

	0	1	2	3	4
S	T	T	F	F	F

Recalculamos los costes a través del nuevo vértice

```
D[2]=min{D[2], D[1]+G->Costes[1,2]}=
      min{∞, 10+50}=60
```

	0	1	2	3	4
D	0	10	60	30	100

	0	1	2	3	4
P	0	0	1	0	0

El coste de ir al vértice '3'($0 \rightarrow 3$) seguirá siendo el mismo, ya que es el único camino que hay y este es de coste 30, se dejan igual los vectores tanto de coste como de vértices y se indica en el vector de booleanos que ya hemos pasado por el vértice '3'.

```
D[3]=min{D[3], D[1]+G->Costes[1,3]}=min{30,10+∞}=30
      D y P siguen igual.

D[4]=min{D[4], D[1]+G->Costes[1,4]}=min{100,10+∞}=100
      D y P siguen igual
```

```
2ª) i = 2
Min {D[2],D[3],D[4]}= Min {60,30,100}= 30
  w = 3
```

	0	1	2	3	4
S	T	T	F	T	F

Ahora nos queda calcular el coste de ir desde el origen al vértice '4', para ello tenemos 4 maneras:

- camino directo: ($0 \rightarrow 4$) con un coste de 100.
- camino secundario: ($0 \rightarrow 1 \rightarrow 2 \rightarrow 4$) con un coste de 70.
- camino secundario: ($0 \rightarrow 3 \rightarrow 2 \rightarrow 4$) con un coste de 60.
- camino secundario: ($0 \rightarrow 3 \rightarrow 4$) con un coste de 90.

Como queremos obtener el coste mínimo, nos vamos a quedar con el camino cuyo coste sea el más pequeño, en este caso será el camino secundario ($0 \rightarrow 3 \rightarrow 2 \rightarrow 4$) que tiene un coste de 60.

Finalmente, actualizamos los vectores de costes, de vértices y el de booleanos el cual tendría que tener a `True` todos los vértices del grafo.

Resultado:

Figura 9.1: Resultado de aplicar Dijkstra

9.1.2. Código del algoritmo Dijkstra

Suma de costes

Como hemos comentado anteriormente haremos uso de un tipo de dato llamado `INFINITO` para indicar que no hay camino directo entre dos vértices del grafo.

```
template <typename tCoste> tCoste suma(tCoste x, tCoste y){
  //vamos a crear un alias para el tipo de dato infinito
  const tCoste INFINITO = GrafoP<tCoste>::INFINITO;
  if(x == INFINITO || y == INFINITO) return INFINITO;
  else return x+y;
}
```

Algoritmo Dijkstra

```
template <typename tCoste> vector<tCoste> Dijkstra (const GrafoP<tCoste>&
↪  G,typename GrafoP<tCoste>::vertice origen, vector<typename
↪  GrafoP<tCoste>::vertice>& vertice){
  typedef typename GrafoP<tCoste>::vertice vertice;
  vertice v,w;
  const size_t n = G.numVert(); //guardamos el numero de vertices
  vector<bool> S(n,false); //vector de booleanos inicializado a falso
  vector<tCoste> D; //Costes minimos
  //Inicializamos D y P con los caminos directos desde el origen
  D = G[origen];
  D[origen] = 0;
  P = vector<vertice>(n,origen);
  S[origen] = true;
  // Localizar vértice w no incluido en S con menor coste desde origen.
  for(size_t i = 1; i<=n-2; i++){
    tCoste costeMinimo = GrafoP<tCoste>::INFINITO;
```

```
    for(v=0;v<= n-1; v++){
      if(!S[v] && D[V] <=costeMinimo){
        costeMinimo = D[v];
        w = v;
      }
    S[w] = true;
  //Recalcular coste hasta cada v no incluido en S, a través de w.
  //Aqui optimizamos de verdad, donde se recálcala los valores del vector.
  for(v=0; v<=n-1; v++)
    if(!S[v]){
      tCoste Owv = suma(D[w], G[w][v]);
      if(Owv < D[v]){
        D[v] = Owv; //Coste de ir del origen a v a través de w.
        P[v] = w;
      }
    }
  }
  return D;
}
```

Método Camino

Este método recibe el vértice origen, un vertice cualquiera (destino) y el vector de vértices y devuelve el camino.

```
template <typename tCoste> typename GrafoP<tCoste>::tCamino camino(typename
↪  GrafoP<tCoste>::vertice orig, typename GrafoP<tCoste>::vertice v,const
↪  vector<typename GrafoP<tCoste>::vertice>& P){
  typename GrafoP<tCoste>::tCamino C;
  C.insertar(v,C.primera());
  do{
    C.insertar(P[v], C.primera());
    v = P[v];
  }while(v!=orig);
  return C;
}
```

9.2. Algoritmo de Floyd

El *algoritmo de Floyd* calcula todos los caminos de coste mínimo **entre cada par de vértices** del grafo G.

```
template <typename tCoste>
matriz<tCoste>
  Floyd(const GrafoP<tCoste>& G, matriz<typename GrafoP<tCoste>::vertice>& P);
```

Como salida tenemos:

- **Matriz de costes mínimos** de tamaño $n * n$, con $n =$ `G.numVert()`.

- **Matriz de vértices** (P) de tamaño $n * n$, tal que, `P[i][j]` es un vértice intermedio, por el que pasa el camino de coste mínimo i, j.

$$A_k\,[i,j] = \min\{A_{k-1}\,[i,j],\ A_{k-1}\,[i,k] + A_{k-1}\,[k,j]\}$$

El subíndice A_{k-1} significa la iteración en la que estamos. Siempre nos quedamos con el valor mínimo.

En *Dijkstra* asumimos que el coste mínimo de ir del origen así mismo es 0. En *Floyd* el equivalente a lo que hacemos en Dijkstra será colocar 0 (n veces), esto se traduce en poner todo a 0 la **diagonal principal de la matriz de costes mínimos**.

9.2.1. Código del algoritmo de Floyd

Algoritmo de Floyd

```
template <typename tCoste>
matriz<tCoste> Floyd(const GrafoP<tCoste>& G, matriz<typename
↪  GrafoP<tCoste>::vertice>& P){
  typedef typedef GrafoP<tCoste>::vertice vertice;
  const size_t n = G.numVert();
  matriz<tCoste> A(n); //matriz de costes mínimos
//Iniciamos A y P con caminos directos entre cada par de vértices
  P = matriz<vertice>(n); //creamos una matriz de vertices
  for(vertice i = 0; i<= n-1 ; i++){
    A[i] = G[i]; //copiamos los costes del grafo
    A[i][i] = 0; //diagonal principal a 0.
    P[i] = vector<vertice>(n,i); //caminos directos
  }
//Calcular costes minimos y caminos correspondientes
//entre cualquier par de vértices i, j
  for(vertice k = 0; k <= n-1; k++)
    for(vertice i = 0; i <= n-1; i++)
      for(vertice j = 0; j<=n-1; j++){
        tCoste ikj = suma(A[i][k], A[k][j]);
        if(ikj < A[i][j]){ //optimización de los costes minimos
          A[i][j] = ikj;
          P[i][j] = k;
        }
```

```
      }
  return A; //devolvemos la matriz de costes
}
```

Cosas a tener en cuenta del código:

- Con `A[i] = G[i]` → copiamos los costes del Grafo en la matriz entrante.
- Con `A[i][i] = 0` → hacemos que el coste mínimo de ir a donde estamos nosotros mismos sea 0, es decir la **diagonal principal** de la matriz (como hemos comentado anteriormente).
- Con `P[i] = vector<vertice>(n,i)` → asignamos a P[i] un vector de vértices por los que pasa.
- La variable `ikj` → es la suma de los costes mínmos de ir desde un vértice i a otro j pasando por k, es decir, `ikj = suma(A[i][k], A[k][j])`.
- Si la suma de estos costes es menor que el coste del grafo '`A[i][j]`', es decir (`ikj < A[i][j]`), podemos optimizar el coste de ir de $i \to j$.

Método Camino

Este método devuelve una lista de vértices entre dos vértices cualesquiera:

```
template <typename tCoste> typename GrafoP<tCoste>::tCamino caminoAux(typename
↪  GrafoP<tCoste>::vertice v, typename GrafoP<tCoste>::vertice w, const
↪  matriz<typename GrafoP<tCoste>::vertice>& P){
// Devuelve el camino de coste mínimo entre v y w, exluidos estos,
// a partir de una matriz P obtenida mediante la función Floyd().
  typename GrafoP<tCoste>::tCamino C1, C2;
  typename GrafoP<tCoste>::vertice u;
  u = P[v][w];
  if(u != w){
    C1 = caminoAux<tCoste>(v,u,P);
    C1.insertar(u,C1.fin());
    C2 = caminoAux<tCamino(u,w,P);
    C1 += C2; //lista<vertice>::operator *=(), concatena C1 y C2.
  }
  return C1; /7/devolvemos la lista
}

template <typename tCoste> typename GrafoP<tCoste>::tCamino camino(typename
↪  GrafoP<tCoste>::vertice v, typename GrafoP<tCoste>::vertice w, const
↪  matriz<typename GrafoP<tCoste>::vertice>& P){
// Devuelve el camino de coste mínimo desde v hasta w a partir
// de una matriz P obtenida mediante la función Floyd().
  typename GrafoP<tCoste>::tCamino C;
  C = caminoAux<tCoste>(v,w,P);
  C.insertar(v,C.primera());
  C.insertar(w,C.fin());
  return C;
}
```

Cosas a tener en cuenta del código del método *caminoAux*:

- El vértice `u` es el último vértice que se ha utilizado para optimizar el camino entre los vértices v y w.
- Mediante `C1 = caminoAux<tCoste>(v,u,P)` → calculamos el camino desde el vértice v hasta el vértice u.
- Mediante `C2 = caminoAux<tCamino(u,w,P)` → calculamos el camino desde el vértice u hasta el vértice w.
- Con `C1 += C2` → concatenamos los caminos en la lista de vértices.

En este método vemos que solamente insertamos el último vértice por el que se ha pasado en el camino $v \to w$, es decir, u.

9.3. Algoritmo de Warshall

El *algoritmo de Warshall* determina **si hay un camino entre cada par de vértices** del grafo **no ponderado** G.

```
matriz<bool> Warshall(const Grafo& G)
```

Como salida tenemos:

- Una **matriz booleana** cuadrada de tamaño `G.numVert()`, donde la posición `[i][j]` es `true` si existe al menos un camino entre los vértice i y j, y es `false` si no existe un camino entre ese par de vértices.

9.3.1. Código del algoritmo de Warshall

Algoritmo de Warshall

```
matriz<bool> Warshall(const Grafo& G){
  typedef Grafo::vertice vertice;
  const size_t n = G.numVert();
  matriz<bool>A(n); //creamos la matriz de booleanos.
//Inicializamos A con la matriz de adyacencia.
  for(vertice i = 0; i<=n-1; i++){
    A[i] = G[i];
    A[i][i] = true;
  }
// Comprobar camino entre cada par de vértices i, j
// a través de cada vértice k.
  for(vertice k = 0; k <= n-1; k++)
    for(vertice i = 0; i <= n-1; i++)
      for(vertice j = 0; j <= n-1; j++){
        if(!A[i][j]){
          A[i][j] = A[i][k] && A[k][j];
        }
      }
```

```
  return A;
}
```

Cosas a tener en cuenta del código:

- Con `A[i][i] = true` → indicamos que hay camino, es decir, siempre hay camino entre un vétice cualquiera y sí mismo.
- Con `!A[i][j]` → comprobamos si hay o no camino entre dicho par de vértice (si es `true` o `false`).
- `A[i][j] = A[i][k] && A[k][j]` → se ejecuta en el caso de que lo anterior fuera `false` (no hay camino), donde comprobamos si hay camino entre i, j, pero pasando por otro vértice cualquiera k.

10. Árboles generadores de coste mínimo

En este capítulo vamos a conocer algunos algoritmos (TAD partición, Kruskal o Prim) para obtener **grafos conexos acíclico**, es decir un árbol.

Es decir, dado un grafo no dirigido y conexo $G = (V, A)$, se define un **árbol generador de G** (o de expansión) como un árbol que conecta todos los vértices de G sin que haya ciclos.

Su coste es la suma de los costes de las aristas del grafo.

Figura 10.1: Ejemplos de árboles generadores de coste mínimo

10.1. TAD Partición

Una **partición** es de un conjunto de enteros $C = \{0, 1, ..., n-1\}$ es un conjunto de subconjuntos disjuntos, cuya unión es el conjunto total C.

Encontramos la **enumeración de elementos** $\rightarrow$ Definimos una aplicación *biyectiva* (1a1) de un conjunto C en un rango de enteros $[0, n-1]$, tal que n es el cardinal de C, mediante dos funciones:

```
int IndiceElto(tElemento x);
```

Precondición: x $\in$ C.
Postcondición: Devuelve el índice del elemento x, en el rango $[0, n-1]$.

```
tElemento NombreElto(int i);
```

Precondición: $0 \leq i \leq n-1$.
Postcondición: Devuelve el elemento de C cuyo índice es i.

10.1.1. Especificación del TAD Partición

Operaciones del TAD partición:

Constructor de la partición

Postcondición: Crea una partición de subconjuntos unitarios en el intervalo de enteros $0, n-1$.

```
Partición (int n);
```

Unir subconjuntos

Precondición: $0 \leq a$, $b \leq n-1$, a y b son los representantes de sus clases y $a \neq b$.
*Postcondición:*Une el subconjunto del elemento a y b en uno de los dos subconjuntos. La partición queda con un miembro menos.

```
void unir(int a, int b);
```

Buscar el representante de una partición

Precondición: $0 \leq x \leq n-1$.
*Postcondición:*Devuelve el representante del subconjunto al que pertenece x.

```
int encontrar(int x) const;
```

10.1.2. Implementación del TAD Partición

El TAD partición podemos implementarlo de diversas maneras (vector, listas, árboles,..), por tanto, vamos a ver como se implementan los algoritmos anteriores.

Implementación mediante vector de pertenencia

En cada posición del vector vamos a encontrar el representante de cada subconjunto.

Figura 10.2: Ejemplo implementación vector de pertenencia

Vemos que en el subconjunto $\{0, 4\}$, en la posición '**0**' del vector, econtramos el representante '**4**', es decir, cada elemento en negrita es el representante del subconjunto.

Si queremos unir dos subconjuntos, primero llamaremos al método `encontrar()` y luego al método `unir()`.

El coste de las operaciones son:

- `encontrar()` $\in O(1)$, porque accedemos a la posición directamente.
- `unir()` $\in O(n)$, porque recorremos todo el vector.
- `Partición()` $\in O(n)$.

Vamos a ver un ejemplo, uniendo los subconjunto de los elemento 4 y 1:

P = {{2,3,**7**} {0,1,**4**,6,8,9} {**5**}}

	0	1	2	3	4	5	6	7	8	9
P	4	4	7	7	4	5	4	7	4	4

Figura 10.3: Resultado implementación vector de pertenencia

Vemos que todos los '1' del vector han pasado a ser '4', es decir, todos los elementos que estaban en el subconjunto donde el representante era el valor '1' se han incluido en el subconjunto cuyo representante es el valor '4'.

Implementación mediante lista de elementos

Es una implementación cuya eficiencia espacial es menor (se gasta más espacio).

P = {{2,3,**7**} {0,**4**} {**5**} {**1**,6,8,9}}

	0	1	2	3	4	5	6	7	8	9
P	4	1	7	7	4	5	1	7	1	1
	-1	6	3	-1	0	-1	8	2	9	-1

Figura 10.4: Ejemplo implementación lista de elementos

Ahora almacenamos tanto el 'vector de pertenencia' (P) que contiene el representante de cada subconjunto y un campo que indica que elemento le precede, en el caso del elemento '0' el valor es -1, es decir, no hay más valores en el subconjunto, pero si nos vamos al elemento '8' vemos que su valor es 9, es decir que el elemento '8' precede al elemento '9'.

Queremos realizar la operación `unir(4,1)`, como vemos en la *Figura 10.4*, tanto el elemento '4' como el '1' son representantes de sus subconjuntos, por tanto si realizamos la unión los valores que suceden a los elementos va a cambiar, quedando:

P = {{2,3,**7**} {0,1,**4**,6,8,9} {**5**}}

	0	1	2	3	4	5	6	7	8	9
P	4	4	7	7	4	5	4	7	4	4
	-1	6	3	-1	1	-1	8	2	9	0

Figura 10.5: Resultado implementación lista de elementos

Es decir, si nos fijamos en el valor que tenía anteriormente el elemento '4' que era 0, ahora vemos que ese valor es 1 (ahora del elemento 4 vamos al elemento 1), además podemos ver que el representante de los elementos '1', '6', '8' y '9' ahora es '4', el representante del nuevo subconjunto.

El coste de las operaciones son:

- `encontrar()` $\in O(1)$, porque accedemos a la posición directamente.
- `unir()` $\in O(n)$, pero el tiempo de ejecución es menor.

Implementación mediante lista de elementos con longitud

Esta implementación es muy parecida a la anteror, pero contiene un campo más para poder evitar el peor caso.

En este nuevo campo almacenamos la longitud para así evitar unir una lista larga con una corta, haciendo que siempre se una una corta a una larga.

P = {{2,3,**7**} {0,**4**} {**5**} {**1**,6,8,9}}

	0	1	2	3	4	5	6	7	8	9
P	4	1	7	7	4	5	1	7	1	1
	-1	6	3	-1	0	-1	8	2	9	-1
	■	4	■	■	2	1	■	3	■	■

Figura 10.6: Ejemplo implementación lista de elementos con longitud

Si nos fijamos en la *Figura 10.6* veremos que el elemento '0' su representante es el elemento '4', además es el último elemento del subconjunto, ya que tiene el valor -1 y al ser el último elemento su longitud es 'nada' (el cuadro negro), a diferencia del elemento '7' cuyo representante es el elemento '7', su sucesor es el elemento '2' y la longitud es 3 (porque es el número de elementos del subconjunto), solamente los representantes tienen este valor asignado.

Vamos a realizar la operación `unir(4,1)`:

P = {{2,3,**7**} {0,**1**,4,6,8,9} {**5**}}

	0	1	2	3	4	5	6	7	8	9
P	1	1	7	7	1	5	1	7	1	1
	6	4	3	-1	0	-1	8	2	9	-1
	■	6	■	■	■	1	■	3	■	■

Figura 10.7: Resultado implementación lista de elementos con longitud

Como vemos, se realizan los mismos cambios que en el ejemplo anterior (*Figura 10.5: Resultado implementación lista de elementos*), pero con el extra de que tenemos que cambiar la longitudes de los representantes. Al unir los subconjuntos de los elementos '4' y '1' y al ser el nuevo representante del nuevo subconjunto el elemento '1' este cambiará su longitud de 4 a 6.

Porqué es el elemento '1' el nuevo representante y no '4', esto se debe a que ahora unimos el **subconjunto corto** al **subconjunto largo**, como el subconjunto cuyo representante era '4' era el corto en comparación al subconjunto con representante '1' obtenemos el resultado del la *Figura 10.7*.

Ahora el coste de las operaciones son:

- `encontrar()` $\in O(1)$, porque accedemos a la posición directamente.
- `unir()` $\in O(n)$, pero el tiempo de ejecución se reduce a la mitad.

Implementación mediante bosque de árboles

Ahora cada subconjunto pasan a ser árboles (lo más ancho posible). En cada posición del vector vamos a encontrar el **padre** de dicho elemento en el árbol, donde el **representante** del subconjunto será el nodo **raíz** de dicho árbol.

Al igual que en los ejemplos anteriores el valor -1 indicará la ausencia de elemento en dicho árbol, en este caso, indicará que ese nodo no tendrá padre y por ende será el representante (nodo ráiz).

Figura 10.8: Ejempo implementación bosque de árboles

Ahora si queremos unir los subconjuntos cuyos representantes son 4,1, es decir realizar la operación `unir(4,1)`, tendríamos que unir el árbol con representante '1' en el árbol con representante '4' o viceversa:

Figura 10.9: Resultado implementación bosque de árboles

En este caso, hemos unido el árbol del subconjunto con representante '1' al del representante '4' (*Figura 10.9*), es decir el segundo se une al primero.

En este caso el coste de las operaciones son:

- `encontrar()` $\in O(n)$.
- `unir()` $\in O(1)$.

Implementación mediante bosque de árboles con control de altura

Sin embargo, si tenemos árboles (subconjuntos) muy grandes y muy pequeños puede ser muy ineficiente la operación `unir()`, para poder subsanar este problema encontramos la implementación mediante *bosque de árboles con control de altura*, donde el unimos el árbol más bajo al más alto, ya que la altura no sube (cota).

Una altura inferior → mayor equilibrio del árbol.

Si los árboles tiene la misma altura, da igual quien se una a quien.

Vamos a diferenciar dos tipos de uniónes:

- **<u>Unión por tamaño:</u>** El árbol con menos nodos se convierte en el subárbol del que tiene mayor número de nodos.

  Es una alternativa a hacerlo por azar.

  En este caso, lo relevante no es la altura, si no el tamaño de árbol (como su nombre dice), además un error muy común es pensar que mayor tamaño $\implies$ mayor altura, esto no es así debido a que un árbol general puede tener muchos nodos hermanos y su altura puede seguir siendo 1.

- **<u>Unión por altura:</u>** El árbol menos alto se convierte en subárbol del otro.

  Esta implementación es la más acertada, debido a que vamos a unir el árbol 'más pequeño' al más 'grande', haciendo que el árbol resultante crezca en anchura.

Figura 10.10: Ejemplo bosque de árboles control de altura (unión por altura)

Ahora en el vector de pertenencia P, si el nodo padre, encontramos que el valor indica la altura del árbol - 1 (en negativo), es decir, se pone $-(altura_arbol - 1)$. Cuando la altura es 0 ponemos el valor -1, debido a que encontramos una ambigüedad en el caso de -0.

Ahora al unir dos árboles, la operación `encontrar()` pasa a ser de coste $O(log\ n)$, ya que vamos descartando las ramas donde no se encuentra dicho valor.

Vamos a realizar la operación `unir(4,1)`:

Como vemos en la *Figura 10.10* el árbol con representante (raíz)'4' es mucho más pequeño que el árbol con raíz '1', por tanto, el primero se convertirá en subárbol del árbol con raíz '1', haciendo que el árbol gane anchura y la altura no se vea incrementada (*Figura 10.11*).

Finalmente, vemos que en el vector el contenido del elemento ‘4’ pasa a ser -2(altura del árbol anterior) a 1 (su nodo padre). Si ambos árboles tienen la misma altura, da

Figura 10.11: Resultado bosque de árboles control de altura (unión por altura)

igual en el orden que insertemos los subárboles.

- **<u>Compresión de caminos:</u>** Optimiza la altura de los árboles.

  Ahora con la primera pasada sabemos el representante del nodo ‘x’ a encontrar, en la segunda pasada ya sabemos todos los nodos que pertenecen a dicho representante.

  Queremos realizar la operación `encontrar(20)`:

Figura 10.12: Ejemplo bosque de árboles control de altura (compresión de caminos)

En la primera pasada encontramos que el representante del subconjunto de 20, es el elemento ‘10’, en la segunda pasada colocamos el elemento ‘20’ como hijo izquierdo del representante (elemento ‘10’) y así sucesivamente con todos los elementos del subconjunto.

Si nos fijamos en el vector de pertenencia P, el elemento ‘18’ tiene como representante al elemento ‘10’, pero esto no se corresponde con el árbol visto en la *Figura 10.12*, por tanto, vamos a optimizar su altura, haciendo que el nodo con valor ‘18’ sea ahora un hijo del nodo con valor ‘10’, lo mismo con el elemento ‘14’.

Figura 10.13: Resultado bosque de árboles control de altura (compresión de caminos)

Vemos que los nodos 20,14 y 18 (señalados en rojo) han cambiado de posición en el árbol, disminuyendo la altura e incrementando la anchura del mismo, además si nos fijamos en le vector de pertenencia los valores de elementos 20(14) ,14(18) y 18(25) han cambiado a 10.

Si hiceramos la operación `encontrar(16)`, tanto el nodo 16, como 9 cambiarían de posición siendo hijos de nodo 10 (su representante).

Gracias a esto el coste de las operaciones son:

- `encontrar()` $O(log\ n)$.
- `unir()` $\in O(1)$.

Código de la implementación mediante bosque de árboles control de altura

```
class Particion{
  public:
    Particion(int n): padre(n-1){}
    void unir(int a, int b);
    int encontrar(int x)const;
  private:
    mutable std::vector<int>padre;
};

void Particion::unir(int a, int b){
  if(padre[b] < padre[a]){
    padre[a] = b;
  }
  else if(padre[a] == padre[b]){//como aumenta la altura, le restamos 1
    padre[a]--; //árbol resultante tiene un nivel más
    padre[b] = a;
  }
}
```

```
int Particion::encontrar(int x)const{
  int y,raiz = x;
  while(padre[raiz] > -1){ //tiene padre y no es representante
    raiz = padre[raiz];
//Compresión del camino: lo nodos del camino se hacen hijos de la raiz.
    while(padre[x]> -1){
      y = padre[x];
      padre[x] =raiz;
      x = y;
    }
  }
  return raiz;
}
```

10.2. Algoritmo de Kruskal

Con el *algoritmo de Kruskal* obtenemos un árbol generador de coste mínimo mediante la ordenación de todas las aristas del grafo por su peso (de menor a mayor) y luego añadiendo las que son más baratas (la menores) al árbol de expansión, asegurándose de que no se producen cíclos.

10.2.1. Ejemplo algoritmo Kruskal

Partimos de un grafo ponderado, no dirigido y conexo y guardamos las aristas según su peso.

Figura 10.14: Ejemplo algoritmo Kruskal

Vemos que tenemos el grafo comentado anteriormente, los vértices del mismo pero sin aristas y una partición P que contiene los subconjuntos a los que pertenece cada elemento (nodo).

Por tanto, para crear obtener el *árbol generador de coste mínimo* cogemos la arista con menor peso y unimos el par de vértices que la contiene, añadiendo en la partición P los nodos que están unidos en un subconjunto común.

Para ello, vamos a seguir una serie de pasos los cuales como resultado obtendremos el árbol de expansión de coste mínimo:

1. Cogemos la arísta con menor coste → *Figura 10.15: Ejemplo 1 Kruskal.*
2. Unimos los vértices que une dicha arista → realizamos la opración `unir(P.encontrar(4), P.encontrar(7));`, donde buscamos los representantes de cada subconjunto y los unimos (*Figura 10.15.(a)*). Si ambos tienen el mismo representante no se unen.
3. Vemos que los vértices se unen al mismo subconjunto y tienen el mismo representante el cual puede ser el mayor o el menor que unen (*Figura 10.15.(b)*).
4. Ahora hacemos lo mismo con la siguente arista cuyo valor sea el menor de los valores mayores o igual a la arista anterior (*Figura 10.16.(a)*).

Para comenzar vamos cogemos la arista cuyo valor sea el más pequeño, este caso será la que une los vértices '4' y '7', con valor 3:

(a) Incluimos la primera arista

P = {{0} {1} {2} {3} {4,7} {5} {6} {8} {9}}

(b) Incluimos la primera arista que conecta los vértices 4 y 7 y ambos se incluyen en el mismo subconjunto.

Figura 10.15: Ejemplo 1 Kruskal

Ahora elegimos la siguente arista cuyo valor sea mayor a la de la arista anterior elegida y que no produzca cíclos:

(a) Incluimos la siguente arista con valor $\geq$ a la anterior

P = {{0} {1} {2} {3} {4,7} {5} {6} {8,9}}

(b) Se unen los vértices 8 y 9 en el mismo subconjunto

Figura 10.16: Ejemplo 2 Kruskal

Hacemos esto hasta que tengamos todos los nodos unidos sin ciclos, muy importante debido a que en los árboles no hay ciclos, y como resultado tenemos:

Como hemos conseguido una partición que contiene un único subconjunto de vértices, no nos hace falta recorrer más la lista de las aristas ni el grafo debido a que todos los vértices están unidos mediante las aristas con los costes mínimos, en definitiva tenemos termiando el *árbol de expansión de coste mínimo.*

Figura 10.17: Resultado ejemplo algoritmo Kruskal

10.3. Algoritmo de Prim

En el *algoritmo de Prim* también vamos a obtener un árbol de expansión de coste mínimo pero usando otra metodología.

Ahora, en vez de tener una partición donde se encuentran todos los vértices del grafo y se va cogiendo la arista con menor valor, vamos a tener un vector de booleanos U que indicará porqué vértices hemos pasado y se empezará desde el vértice con menor coste.

Figura 10.18: Ejemplo del algoritmo Prim

Vemos que partimos desde el vértice con valor 0 y que tenemos 3 aristas disponibles con costes (20, 21 y 24), lo que vamos a hacer es pasar por la arista que tenga menor coste, es decir, por la de coste 20 uniendo los vértices (0 y 3) y las otras dos restantes quedaría 'eliminadas'.

(a) Unimos ambos vértices con la arista de menor coste.

	0	1	2	3	4	5	6	7	8	9
U	T	F	F	T	F	F	F	F	F	F

(b) Ponemos en T (true) los vértices por los que hemos pasado.

Figura 10.19: Ejemplo 1 Prim

Ahora nos encontramos en el vértice '3' y si nos fijamos en el grafo de la *Figura 10.17* vemos que se puede unir con varios vértices (3 ↔ 2 con coste 13), (3 ↔ 6 con coste 13) y (3 ↔ 5 con coste 10), como la arista de menor coste es 10, uniremos (3 ↔ 5).

(a) Unimos 3 y 5.

	0	1	2	3	4	5	6	7	8	9
U	T	F	F	T	F	T	F	F	F	F

(b) Ponemos en T (true) en el vértice 5.

Figura 10.20: Ejemplo 2 Prim

Ahora encontramos un problema, tras haber unido los vértices (2 y 5) nos encontramos en el vértices '2', pero no podemos unirlo con nigún vertice ya que si lo unimos con los vértices (0,3) tendríamos un ciclo.

	0	1	2	3	4	5	6	7	8	9
U	T	F	T	T	F	T	F	F	F	F

Figura 10.21: Ejemplo 3 Prim

Si nos fijamos en el grafo de ejemplo (*Figura 10.17*) vamos que el vértice 2 se puede unir con el 6 ($2 \leftrightarrow 6$ con coste 17), pero es que el vértice 3 también se puede unir con 6 ($3 \leftrightarrow 6$ con coste 13), como el coste de este último es menor, saltamos del vértice 2 al 3 y unimos con el vértice 6.

(a) Unimos 3 y 6

	0	1	2	3	4	5	6	7	8	9
U	T	F	T	T	F	T	T	F	F	F

(b) Ponemos en T (true) en el vértice 6.

Figura 10.22: Ejemplo 4 Prim

Ahora vemos que podemos unir los vertices 6 y 7 ($6 \leftrightarrow 7$ con coste 25), pero también podemos saltar al vértice 0 y unirlo con 1 ($0 \leftrightarrow 1$ con coste 24), como la arista de éste último tiene menor coste y por tanto, se unen primero dichos vértices y luego unimos 6 con 7.

(a) Unimos primero 0 y 1 y luego 6 y 7.

	0	1	2	3	4	5	6	7	8	9
U	T	T	T	T	F	T	T	T	F	F

(b) Ponemos en T (true) en el vértice 1 y 7.

Figura 10.23: Ejemplo 5 Prim

Seguimos uniendo vértices teniendo en cuenta el peso de las aristas y tenemos como resultado final:

(a) Todos los nodos unidos sin ciclos.

	0	1	2	3	4	5	6	7	8	9
U	T	T	T	T	T	T	T	T	T	T

(b) Vector de booleano con todos a T (true).

Figura 10.24: Resultado ejemplo de algoritmo Prim

Como vemos en las figuras (*Figura 10.17: Resultado ejemplo algoritmo Kruskal*) y (*Figura 11.24: Resultado ejemplo algoritmo Prim*) ambos grafos son idénticos, sin embargo, pueden existir casos en que sean diferentes.

Esto es debido a que aunque Kruskal y Prim trabajen con metodologías diferentes pueden devolver el mismo *árbol de expansión de coste mínimo*, ya que al estar uniendo vértices cuya aristas son las de menor coste es muy probable que el resultado sea el mismo.

Es decir, *Kruskal* y *Prim* hacen lo mismo pero de menera diferente.