Apuntes Examen Final EDA

Estructuras de datos

Tipos abstractos de datos (TAD)

Conjuntos de elementos con las operaciones que los caracterizan
Independientes de su implementación

Diccionarios:

Tablas de hash y árboles de búsqueda binarios
Se trata de un conjunto de elementos
Éstos son generalmente entradas que asocian una clave con un valor

Operaciones:
- Búsqueda de un elemento
- Inserción
- Borrado

Colas de prioridad:

Heaps binarias y binomiales
Es un conjunto ordenado de elementos
- Max-heap: ordenados de mayor a menor
- Min-heap: de menor a mayor

Operaciones:
- Consulta del elemeto mayor (o menor)
- Inserción de un elemento
- Borrado del elemento mayor (o menor)

Tablas de Hash

Las **tablas de hash** son estructuras de datos que permiten asociar una clave con un valor, para más tarde encontrar de forma eficiente ese valor a partir de su clave.

Esas entradas "clave-valor" se almacenan en una tabla, y su posicion viene determinada por una **función de hash**.

El acceso a un valor debe ser muy rápido, pero es imposible evitar colisiones -que se producen cuando diferentes claves se mapean a la misma posición en la tabla.

La estrategia de **resolución de colisiones** que veremos a continuación se llama '**Encadenamiento separado**' (o '**Open chaining**').

Encadenamiento separado

Esta estrategia consiste en tener una tabla de listas, en lugar de entradas, en las que se almacenan las entradas una detrás de otra.

El siguiente código C++ es una simple implementación de este tipo de tablas de hash, que mapea claves int a valores int.

```
#include <iostream>
#include <vector>
#include <list>
using namespace std;

typedef pair<int, int> IP;
typedef list<IP> IPL;
typedef IPL::iterator IPLI;
typedef vector<IPL> IPM;

// Separate Chaining Hash Table
// Implementacion de una tabla de hash de encadenamiento separado
// Viene a ser lo mismo que el unordered_map<int, int> de la STL
struct HashTable {

  int cnt;  // numero de elementos de la tabla
  IPM tbl;  // tabla de entradas

  // Capacidad inicial de 8
  // Factor de carga: 2
  HashTable() {
    cnt = 0;
    tbl = IPM(8);
  }

  // Simple funcion de hash: el propio valor de 'key' modulo 'mod'
  inline int hash(int key, int mod) {
    return key % mod;
  }
```

```
  // Redimensiona la tabla de entradas creando una nueva tabla el doble de grande,
  // e insertando ahi todas las entradas existentes
  // Complejidad: O(n)
  void resize() {
    IPM tbl_tmp = IPM(tbl.size() * 2);
    for (int i = 0; i < (int)tbl.size(); ++i) {
      for (IPLI it = tbl[i].begin(); it != tbl[i].end(); ++it) {
       const IP &key_val = *it;
       int pos = hash(key_val.first, tbl_tmp.size());
       tbl_tmp[pos].insert(tbl_tmp[pos].end(), key_val);
      }
    }
    tbl = tbl_tmp;
  }

  // Inserta una entrada <clave, valor> en la tabla
  // Coste amortizado: O(1)
  // En el peor de los casos, todas las entradas irian a la misma lista: O(n)
  void insert(int key, int val) {
    // Encontrar la posicion
    int pos = hash(key, tbl.size());
    // Sustituir el valor si existe una entrada con esa 'key'
    for (IPLI it = tbl[pos].begin(); it != tbl[pos].end(); ++it) {
      if (it->first == key) {
       it->second = val;
       return;
      }
    }
    // Si no la 'key' es nueva, insertarla en la siguiente posicion libre
    tbl[pos].insert(tbl[pos].end(), IP(key, val));
    ++cnt;
    // Si se ha superado el factor de carga, redimensionar
    if (cnt >= 2 * (int)tbl.size()) resize();
  }

  // Eliminar la entrada cuya clave es 'key'
  // Coste amortizado: O(1)
  // Aunque si todas las entradas estan en la misma posicion, O(n)
  void erase(int key) {
    // Encontrar la posicion en la que se encontraria la entrada
    int pos = hash(key, tbl.size());
    for (IPLI it = tbl[pos].begin(); it != tbl[pos].end(); ++it) {
      if (it->first == key) {
       // clave encontrada -eliminarla
       tbl[pos].erase(it);
       return;
      }
    }
  }

  // Retorna el valor asociado a la clave 'key'
  // Si ningun valor ha sido asociado a esa clave, devuelve -(2^31)
  // Misma complejidad que 'insert' y 'erase'
  int get_value(int key) {
    int pos = hash(key, tbl.size());
    for (IPLI it = tbl[pos].begin(); it != tbl[pos].end(); ++it) {
      if (it->first == key) {
       return it->second;
      }
    }
    return 1 << 31;
  }

};
```

Direccionamiento abierto

El **direccionamiento abierto** es otra estrategia de **resolución de colisiones** para tablas de hash. Aquí propongo una sencilla implementación en C++.

Esta estrategia de resolución de colisiones consiste en almacenar cada entrada en la siguiente posición libre de la tabla, a partir de su posición natural -la que proporciona la función de hash.

```
#include <iostream>
#include <vector>
using namespace std;

typedef pair<int, int> IP;
typedef vector<IP*> IPV;

// Open Addressing Hash Table
// Implementacion de una tabla de hash de direccionamiento abierto
// Viene a ser lo mismo que el unordered_map<int, int> de la STL
struct HashTable {

  int cnt;  // numero de elementos
  IPV tbl;  // tabla de entradas

  // Crea la hash table, con capacidad inicial de 8
  // Factor de carga: 0.8
  HashTable() {
    cnt = 0;
    tbl = IPV(8, NULL);
  }

  // Simple funcion de hash: el propio valor de 'key' modulo 'mod'
  inline int hash(int key, int mod) {
    return key % mod;
  }

  // Encuentra la posicion de la tabla en la que se encuentra 'key'
  // O si la 'key' no esta en la tabla, encuentra la posicion (vacia)
  // en la que es posible insertar 'key'
  // En el caso peor habra que patearse la tabla entera O(n)
  // Igual el coste amortizado es O(1)
  inline int slot(int key, const IPV &table) {
    int pos = hash(key, table.size());
    while (table[pos] != NULL) {
      if (table[pos]->first == key) return pos;
      pos = (pos + 1) % (int)table.size();
    }
    return pos;
  }

  // Redimensiona la tabla
  // Crea una nueva tabla el doble de grande,
  // y anade los elementos de la tabla original uno a uno
  // Complejidad: O(n)
  void resize() {
    IPV tbl_tmp(2 * tbl.size(), NULL);
    for (int i = 0; i < (int)tbl.size(); ++i) {
      if (tbl[i] != NULL) {
       int pos = slot(tbl[i]->first, tbl_tmp);
       tbl_tmp[pos] = tbl[i];
      }
    }
    tbl = tbl_tmp;
  }
```

```
  // Inserta un par <clave, valor> en la tabla
  // En el peor de los casos, todas las entradas irian a la misma posicion
  // de la tabla -complejidad O(n)
  // En el mejor caso, cada entrada se mapearia en una posicion distinta
  // en cuyo caso la complejidad seria O(1)
  // Coste amortizado: O(1)
  void insert(int key, int val) {
    // Encontrar la posicion para la nueva entrada
    int pos = slot(key, tbl);
    if (tbl[pos] == NULL) {
      // La posicion esta ocupada -> la 'key' ya tiene un valor asociado
      // hay que sobreescribirlo
      ++cnt;
      tbl[pos] = new IP(key, val);
      // Factor de carga rebasado: hay que redimensionar la tabla
      if (cnt >= (int)(0.8 * tbl.size())) resize();
    } else {
      // Hay hueco -> insertar ahi la nueva entrada
      tbl[pos]->second = val;
    }
  }

  // Elimina la entrada cuya clave es 'key'
  // Al igual que la insercion, el coste puede ser
  // entre O(1) -caso mejor-, y O(n) -caso peor.
  // Coste amortizado: O(1)
  void erase(int key) {
    // Encontrar entrada y eliminarla de la tabla
    int pos = slot(key, tbl);
    if (tbl[pos] != NULL) delete tbl[pos];
    // no puede haber huecos entre la posicion natural de una entrada
    // y su posicion establecida -si no, la busqueda de la funcion 'slot' fallaria
    // asi que hay que reubicar las entradas que caerian en la posicion eliminada
    int nex_pos = pos;
    while (tbl[pos] != NULL) {
      // eliminar la entrada
      tbl[pos] = NULL;
      bool loop = true;
      while (loop) {
       // si la siguiente posicion esta vacia, hemos terminado
       nex_pos = (nex_pos + 1) % tbl.size();
       if (tbl[nex_pos] == NULL) return;
       // 'p' es la posicion natural por hash de la siguiente entrada en la tabla
       // si 'p' no esta dentro del intervalo ciclico [pos, next_pos],
       // hay que rellenar la posicion vacia con la entrada en nex_pos
       int p = hash(tbl[nex_pos]->first, tbl.size());
       loop = pos <= nex_pos ? pos < p && p <= nex_pos : pos < p || p <= nex_pos;
      }
      // hacer la reubicacion de la entrada conflictiva
      tbl[pos] = tbl[nex_pos];
      pos = nex_pos;
    }
  }

  // Retorna el valor asociado a la clave 'key'
  // Si ningun valor ha sido asociado a esa clave, devuelve -(2^31)
  // Misma complejidad que 'insert' y 'erase'
  // Coste amortizado: O(1)
  int get_value(int key) {
    int pos = slot(key, tbl);
    if (tbl[pos] != NULL) return tbl[pos]->second;
    return 1 << 31;
  }

};
```

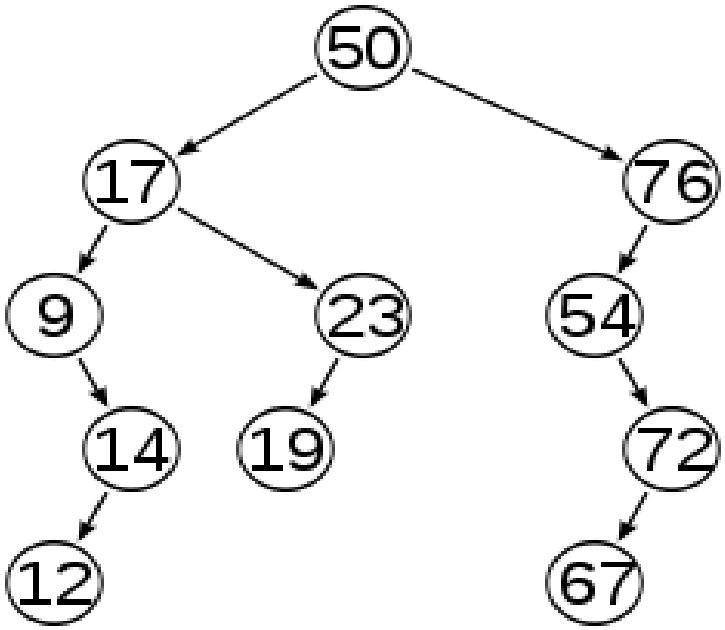
Árboles binarios de búsqueda (BST)

Un BST es muy útil cuando es necesario recorrer los elementos del diccionario en orden según su clave de forma recurrente

Un árbol binario es un BST cuando cumple las siguientes condiciones:

- Un árbol binario vacío es un BST

- La clave del nodo que tratamos es mayor que cualquiera de las claves que contienen los hijos de su subárbol izquierdo y menor que todas las claves de los hijos de su subárbol derecho



Árboles AVL

Ideado por Adelson-Velskii y Landis, el **árbol AVL** es un árbol binario de búsqueda auto-balanceado

Tiene la propiedad de que *la diferencia entre las alturas de los dos subárboles que cuelgan de cualquier nodo es como mucho 1*

Esto agiliza sus funciones básicas de **inserción**, **borrado**, y **consulta**, que tienen un coste de **`O(log n)`**.

Sea **T** un árbol binario con hijos **T1** y **T2**. Como **BST** (*árbol binario de búsqueda*), los valores en el subárbol izquierdo **T1** son todos menores que el valor en **T**, y los valores en el subárbol izquierdo **T2** son iguales o mayores.

Altura **h(T)** de un AVL

- Si T1 y T2 son árboles vacíos, es **1**
- De lo contrario, **h(T) = max(h(T1), h(T2)) + 1**

Definición de un AVL

- Un árbol **T** vacío es un AVL
- Si **T1** y **T2** son AVL, y **|h(T1) - h(T2)| <= 1**, **T** es un AVL

Operaciones básicas

- Inserción
Añade un nuevo valor al árbol.
Complejidad: O(log n)

- Borrado
Elimina un valor del árbol, si éste lo contiene.
Complejidad: O(log n)

- Búsqueda
Encuentra un valor, si está en el árbol.
Complejidad: O(log n)

Rotaciones

Tras una operación de inserción, o un borrado, es posible que el árbol quede desequilibrado, y se viole la condición **|h(T1) - h(T2)| <= 1** en algún nodo

Si es así, es necesario realizar una operación de **rotación** para re-equibrar el árbol (en el caso de la *inserción*; en caso de *borrado*, posiblemente varias)

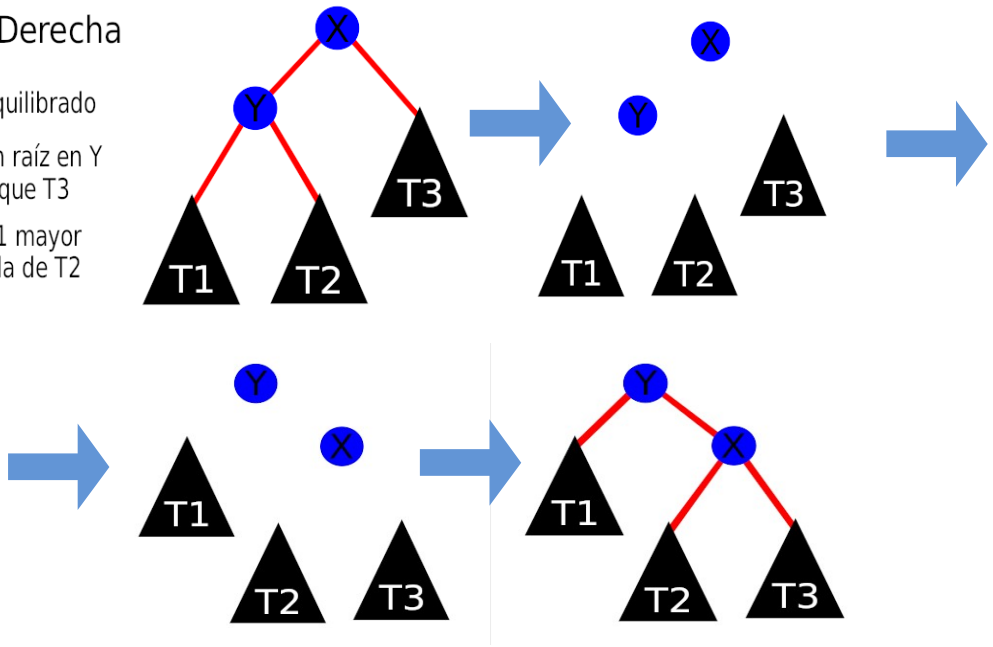
El **coste** de las rotaciones es constante **O(1)**

Rotación Derecha

X está desequilibrado

Subárbol con raíz en Y más alto que T3

Altura de T1 mayor o igual que la de T2

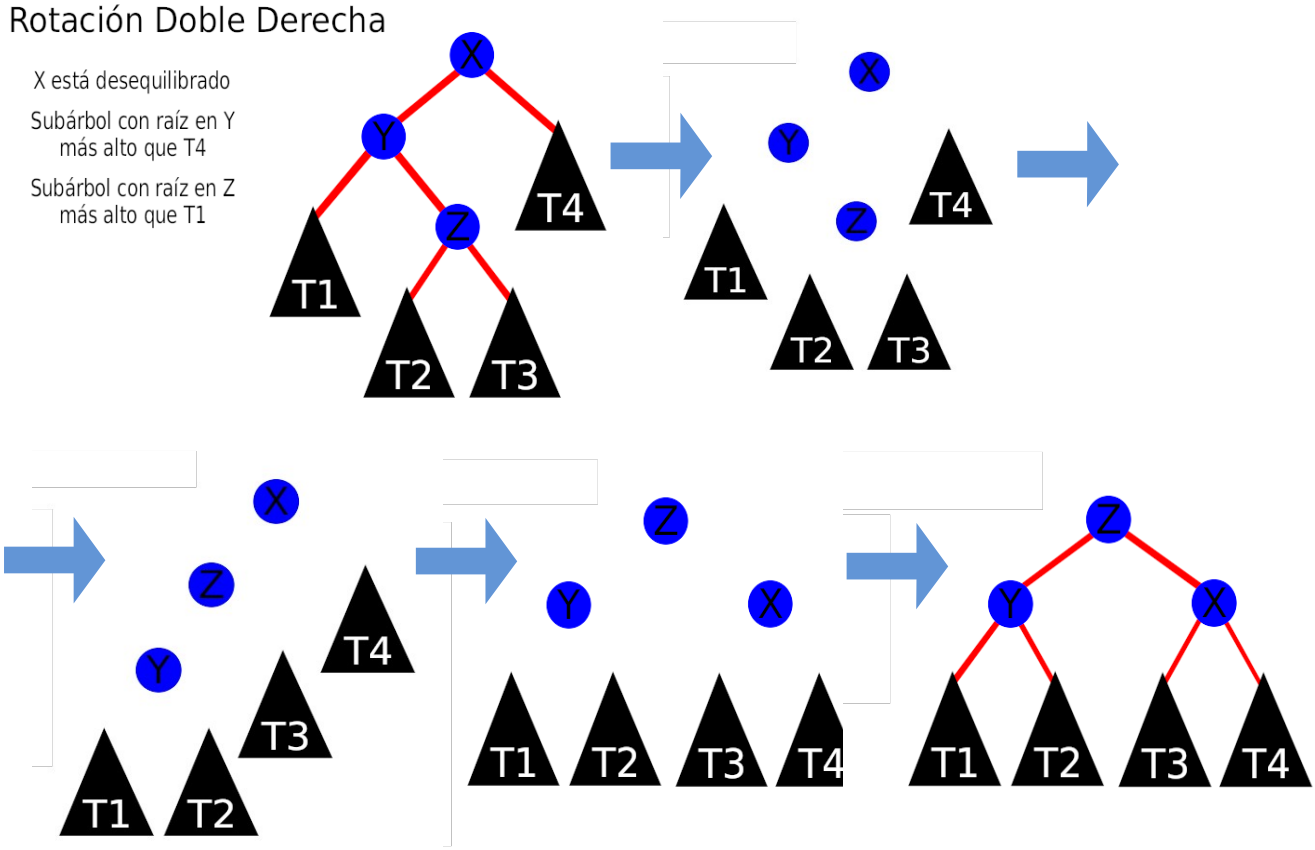


Rotación Doble Derecha

X está desequilibrado

Subárbol con raíz en Y más alto que T4

Subárbol con raíz en Z más alto que T1



Implementación de un AVL en C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> IV;

// nodo del arbol
struct Node {
  int val;    // valor
  int hei;    // altura
  Node* lef;  // subarbol izquierdo
  Node* rig;  // subarbol derecho
};

// implementacion del AVL
// permite elementos repetidos (como el 'multiset' de la stl)
struct AVL {

  // nodo raiz del arbol
  Node* root;

  // retorna la altura de 'nod'
  inline int height(Node* nod) {
    return nod == NULL ? 0 : nod->hei;
  }

  // balance del arbol con raiz en 'nod':
  // sera negativo si el subarbol derecho es mas alto que el izquierdo
  // positivo si el subarbol izquierdo es el mas alto
  // o cero si son iguales
  inline int balance(Node* nod) {
    return nod == NULL ? 0 : height(nod->lef) - height(nod->rig);
  }

  // actualiza la altura de 'nod'
  inline void update_height(Node* nod) {
    int hl = height(nod->lef);
    int hr = height(nod->rig);
    nod->hei = (hl > hr ? hl : hr) + 1;
  }

  // rotacion derecha sobre 'x'
  inline Node* rotate_right(Node* x) {
    Node* y = x->lef;
    Node* t2 = y->rig;
    y->rig = x;
    x->lef = t2;
    update_height(x);
    update_height(y);
    return y;
  }

  // rotacion izquierda sobre 'x'
  inline Node* rotate_left(Node* x) {
    Node* y = x->rig;
    Node* t2 = y->lef;
    y->lef = x;
    x->rig = t2;
    update_height(x);
    update_height(y);
    return y;
  }
```

```
// rotacion doble a la derecha
inline Node* double_rotate_right(Node* nod) {
  nod->lef = rotate_left(nod->lef);
  return rotate_right(nod);
}

// rotacion doble a la izquierda
inline Node* double_rotate_left(Node* nod) {
  nod->rig = rotate_right(nod->rig);
  return rotate_left(nod);
}

// rotaciones: equilibran el arbol despues de una insercion o borrado
// al acabar, es necesario recalcular las alturas de los nodos tratados
// complejidad: O(1)
inline Node* update_balance(Node* nod) {
  // si el arbol ha quedado desequilibrado hay que hacer algun tipo de rotacion
  int bal = balance(nod);
  if (bal > 1) {
    // el subarbol izquierdo es mas alto que el derecho
    // hay que mirar su balance
    int bal_lef = balance(nod->lef);
    if (bal_lef >= 0) {
     // el sub-subarbol izquierdo es tan o mas alto: rotacion simple derecha
     return rotate_right(nod);
    } else {
     // el hijo derecho del subarbol izquierdo de 'nod' es mas alto
     // rotacion doble derecha
     return double_rotate_right(nod);
    }
  } else if (bal < -1) {
    // el subarbol derecho es mas alto que el izquierdo
    // hay que mirar su balance
    int bal_rig = balance(nod->rig);
    if (bal_rig <= 0) {
     // el sub-subarbol derecho es tan o mas alto: rotacion simple izquierda
     return rotate_left(nod);
    } else {
     // el hijo izquierdo del subarbol derecho de 'nod' es mas alto
     // rotacion doble izquierda
     return double_rotate_left(nod);
    }
  }
  // no hubo rotacion, devolver el nodo tal cual
  return nod;
}

// insercion: anadir 'val' en el (sub)arbol 'nod'
// el re-balanceado se produce como mucho 1 vez
// complejidad: O(log n)
Node* insert(Node* nod, int val) {

  // nodo vacio, podemos insertar el valor aqui
  if (nod == NULL) return new Node({ val, 1, NULL, NULL });

  // si el valor a insertar es menor que el de 'nod'
  // insertar en el subarbol izquierdo
  // si es mayor o igual, ir al subarbol derecho
  // al acabar, recalcular la altura de 'nod'
  if (val < nod->val) nod->lef = insert(nod->lef, val);
  else nod->rig = insert(nod->rig, val);
  update_height(nod);

  // equilibrar el arbol si es necesario
  return update_balance(nod);
}
void insert(int val) {
  root = insert(root, val);
}
```

```
// devuelve el nodo de mayor valor del arbol con raiz en 'nod'
inline Node* max_node(Node* nod) {
  while (nod->rig != NULL) {
    nod = nod->rig;
  }
  return nod;
}

// borrado: eliminar 'val' del (sub)arbol 'nod'
// el re-balanceado puede producirse en todos los padres del nodo borrado
// complejidad: O(log n)
Node* erase(Node* nod, int val) {

  // si el nodo esta vacio, no hay nada que borrar
  if (nod == NULL) return NULL;

  // si el valor a borrar es menor que el de 'nod' buscar en el subarbol izquierdo
  // si es mayor, ir al subarbol derecho
  if (val < nod->val) nod->lef = erase(nod->lef, val);
  else if (val > nod->val) nod->rig = erase(nod->rig, val);
  else {
    // el valor del nodo es el que queremos borrar
    // hay 3 posibilidades:
    if (nod->lef == NULL && nod->rig == NULL) {
     // 'nod' es una hoja -ponerlo a NULL, liberar memoria y retornar
     Node* leaf = nod;
     nod = NULL;
     delete leaf;
     return NULL;
    } else if (nod->lef == NULL || nod->rig == NULL) {
     // 'nod' es el padre de una hoja
     // sustituir 'nod' por su hijo, eliminar hoja y retornar
     Node* leaf = nod->lef == NULL ? nod->rig : nod->lef;
     *nod = *leaf;
     delete leaf;
     return nod;
    } else {
     // sustituir 'nod' por el nodo de mayor valor de su subarbol izquierdo
     // y eliminar ese nodo de mayor valor
     Node* leaf = max_node(nod->lef);
     nod->val = leaf->val;
     nod->lef = erase(nod->lef, leaf->val);
    }
  }

  // actualizar altura, y a lo mejor equilibrar el arbol
  update_height(nod);
  return update_balance(nod);
}
void erase(int val) {
  root = erase(root, val);
}

// consulta: esta 'val' presente en el (sub)arbol 'nod'?
// complejidad: O(log n)
Node* find(Node* nod, int val) {
  // nodo vacio, el valor buscado no esta en el arbol
  if (nod == NULL) return NULL;

  // si el valor buscado es menor que el de 'nod'
  // buscar en el subarbol izquierdo
  // si es mayor, ir al subarbol derecho
  // si es el mismo, lo hemos encontrado :)
  if (val < nod->val) return find(nod->lef, val);
  else if (val > nod->val) return find(nod->rig, val);
  return nod;
}
bool find(int val) {
  return find(root, val) == NULL ? 0 : 1;
}
```

```
// imprimir los valores del arbol, en pre-orden
// complejidad: O(n)
void print_preorder(Node* nod) {
  if (nod != NULL) {
    cout << nod->val << ' ';
    print_preorder(nod->lef);
    print_preorder(nod->rig);
  } else {
    cout << "* ";
  }
}
void print_preorder() {
  print_preorder(root);
  cout << endl;
}

// imprimir los valores del arbol, en in-orden
// complejidad: O(n)
void print_inorder(Node* nod) {
  if (nod != NULL) {
    print_inorder(nod->lef);
    cout << nod->val << ' ';
    print_inorder(nod->rig);
  } else {
    cout << "* ";
  }
}
void print_inorder() {
  print_inorder(root);
  cout << endl;
}

// imprimir los valores del arbol, en post-orden
// complejidad: O(n)
void print_postorder(Node* nod) {
  if (nod != NULL) {
    print_postorder(nod->lef);
    print_postorder(nod->rig);
    cout << nod->val << ' ';
  } else {
    cout << "* ";
  }
}
void print_postorder() {
  print_postorder(root);
  cout << endl;
}

Node* build_preorder(const IV &vals, int start, int end) {
  if (start > end) return NULL;
  int i = start + 1;
  while (i < end && vals[i] < vals[start]) ++i;
  Node* nod = new Node({ vals[start], 1, NULL, NULL });
  nod->lef = build_preorder(vals, start + 1, i - 1);
  nod->rig = build_preorder(vals, i, end);
  update_height(nod);
  return nod;
}

Node* build_preorder2(const IV &vals, Node* &nod, int mx, int &pos, int len) {
  nod = new Node({ vals[pos], 0, NULL, NULL });
  ++pos;
  if (pos < len && vals[pos] < nod->val) {
    build_preorder2(vals, nod->lef, nod->val, pos, len);
  }
  if (pos < len && (mx == -1 || mx > vals[pos])) {
    build_preorder2(vals, nod->rig, mx, pos, len);
  }
  update_height(nod);
  return nod;
}
```

```
  Node* build_sorted(const IV &vals, int start, int end) {
    if (start > end) return NULL;
    int m = (start + end) / 2;
    Node* nod = new Node({ vals[m], 1, NULL, NULL });
    nod->lef = build_sorted(vals, start, m - 1);
    nod->rig = build_sorted(vals, m + 1, end);
    update_height(nod);
    return nod;
  }

  // inicializa el arbol a partir de un vector con los elementos en preorden
  void init(const IV &preorder) {
    int n = preorder.size();
    int i = 0;
    build_preorder2(preorder, root, -1, i, n);
    // version menos eficiente para inicializar:
    // root = build_preorder(preorder, 0, n - 1);
    // inicializa el arbol a partir de un vector de elementos ordenados:
    // root = build_sorted(preorder, 0, n - 1);
  }

};
```

Heaps

Los **heaps** son estructuras de datos basadas en árboles, que permiten acceder a un conjunto de datos de forma ordenada

Max-heap y min-heap

El max-heap permite acceder a los elementos de mayor a menor, mientras que con el min-heap el acceso es de menor a mayor

Propiedad de heap

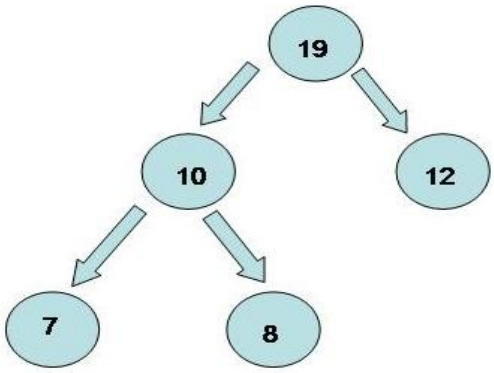
Un árbol cumple la **propiedad de heap** si **el valor de un nodo es mayor o igual que el de sus nodos hijos** -caso del max-heap-, o menor o igual si se trata de un min-heap

Heap binario

Árbol binario

Es el tipo de árbol que respalda los heaps binarios

Para un heap, además, el árbol debe ser **completo** -es decir, todos sus niveles deben estar llenos, salvo tal vez el último, que debe rellenarse de izquierda a derecha



Operaciones básicas

- **Consulta**: Permite conocer el elemento de máximo (o mínimo) valor del heap
Complejidad: O(1)

- **Inserción**: Añade un nuevo valor al heap
Complejidad: O(log n)

- **Borrado**: Elimina el máximo (o mínimo) del heap
Complejidad: O(log n)

Una implementación en C++ de un heap binario tendría esta pinta:

```cpp
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> IV;

// Implementacion de un max-heap binario
struct MaxHeap {

  // Arbol binario representado con un vector
  vector<int> tree = { 0 };

  // posicion del nodo padre de 'pos' = pos / 2
  inline int parent(int pos) {
    return pos / 2;
  }

  // posicion del hijo izquierdo de 'pos' = 2 * pos
  inline int left(int pos) {
    return 2 * pos;
  }

  // posicion del hijo derecho de 'pos' = 2 * pos + 1
  inline int right(int pos) {
    return 2 * pos + 1;
  }

  // Devuelve true si el heap esta vacio
  bool empty() {
    return tree.size() <= 1;
  }

  // Asciende el valor en la posicion 'pos'
  // Mientras el valor del nodo padre de 'pos'
  // sea menor o igual que el valor del nodo 'pos'
  // intercambia sus valores
  inline void rise(int pos) {
    // mientras el nodo tenga padre
    while (pos / 2 > 0) {
      // posicion del padre
      int pos_par = parent(pos);
      // si el valor en pos no es mayor que el de su padre, hemos terminado
      if (tree[pos] <= tree[pos_par]) return;
      // si es mayor, intercambiar sus valores y repetir la operacion
      swap(tree[pos], tree[pos_par]);
      pos = pos_par;
    }
  }
  // Inserta un valor en el heap
  // Anade el valor en la siguiente posicion libre
  // y lo 'asciende' por el arbol hasta que se cumpla la propiedad de heap
  // Complejidad: O(log n)
  void push(int x) {
    tree.push_back(x);
    rise(tree.size() - 1);
  }
```

```
  // Hunde el nodo en la posicion 'pos'
  // Mientras el valor del nodo 'pos' sea menor que el de alguno de sus hijos,
  // intercambia su valor con el del hijo de mayor valor
  inline void sink(int pos) {
    // mientras el nodo tenga al menos 1 hijo
    while (pos * 2 < (int)tree.size()) {
      int tmp = pos;  // almacenara el nodo de mayor valor
      int pos_lef = left(pos);   // hijo izquierdo
      int pos_rig = right(pos);  // hijo derecho
      // comparar el valor de pos con el del hijo derecho
      if (pos_rig < (int)tree.size() && tree[pos_rig] > tree[tmp]) tmp = pos_rig;
      // y ahora con el del hijo izquierdo
      if (tree[pos_lef] > tree[tmp]) tmp = pos_lef;
      // tmp seguira siendo igual a 'pos' si el valor en 'pos' es el mayor
      // en este caso hemos terminado
      if (tmp == pos) return;
      // de lo contrario, intercambiar los valores y repetir la operacion
      swap(tree[pos], tree[tmp]);
      pos = tmp;
    }
  }
  // Elimina el nodo de mayor valor del heap
  // Consiste en intercambiar el valor en la raiz con el del ultimo nodo,
  // luego eliminar ese ultimo nodo,
  // y hundir el valor en la raiz hasta que el arbol cumpla la propiedad de heap
  // Complejidad: O(log n)
  void pop() {
    if (!empty()) {
      swap(tree[1], tree[tree.size() - 1]);
      tree.pop_back();
      sink(1);
    }
  }

  // Consulta el mayor valor del heap
  // Si el heap esta vacio, devuelve -(2^31)
  // Por definicion de heap, la raiz del arbol contiene ese maximo
  // Complejidad: O(1)
  int top() {
    return empty() ? 1 << 31 : tree[1];
  }

};
```

Heapsort

Este algoritmo de ordenación se basa en heaps, y consiste en:

1. Convertir el array a ordenar en un heap.
2. Sacar todos sus elementos uno por uno, generando así el array ordenado.

A continuación está mi implementación del algoritmo

Pero ojo, que, a diferencia de la anterior implementación del heap binario, aquí el vector tiene su primer elemento en la posición 0 (no en la 1). En consecuencia, la función para calcular el índice del padre o los hijos de un nodo, está ligeramente modificada

```
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> IV;

// Hundir el nodo 'pos' hasta que su padre tenga un valor mayor o igual
// Complejidad: O(log n)
void sink(IV &vec, int pos, int end) {
  while (pos * 2 + 1 < end) {
    int tmp = pos;
    int pos_lef = 2 * pos + 1;
    int pos_rig = 2 * pos + 2;
    if (pos_rig < end && vec[pos_rig] > vec[tmp]) tmp = pos_rig;
    if (vec[pos_lef] > vec[tmp]) tmp = pos_lef;
    if (tmp == pos) return;
    swap(vec[pos], vec[tmp]);
    pos = tmp;
  }
}

// Reorganiza los valores de 'vec'
// de forma que represente el arbol binario de un max-heap
// Complejidad: O(n * log n)
void make_heap(IV &vec) {
  // encontrar la posicion del ultimo nodo que tiene algun hijo
  int pos = (vec.size() - 2) / 2;
  while (pos >= 0) {
    sink(vec, pos, vec.size());
    --pos;
  }
}

// Ordenacion por heap
// Consiste en transformar el vecor en un heap binario
// y sacar repetidamente el mayor valor, anadiendolo al final del vector
// El vector al final del proceso queda ordenado de menor a mayor
// Complejidad: O(n * log n)
void heap_sort(IV &vec) {
  // transforma vec en heap
  make_heap(vec);
  int pos = vec.size() - 1;
  while (pos > 0) {
    // en cada iteracion pone el mayor valor
    // en la ultima posicion de la parte del vector que es heap
    swap(vec[0], vec[pos]);
    sink(vec, 0, pos);
    --pos;
  }
}
```

Heap binomial

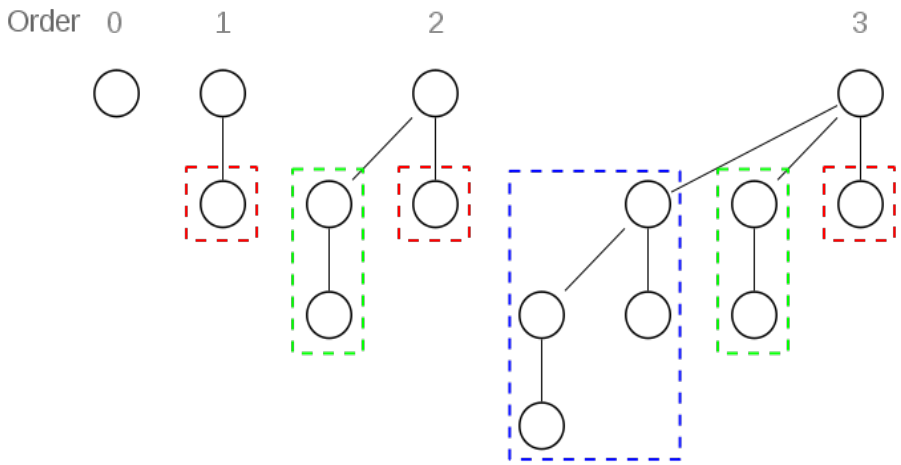
Este tipo de heap se fundamenta en **árboles binomiales**, y se caracteriza por ser capaz de fusionar de forma eficiente dos heaps (en tiempo $O(\log n)$)

Además, la operación de inserción es más eficiente que la del heap binario, consiguiendo un tiempo amortizado constante

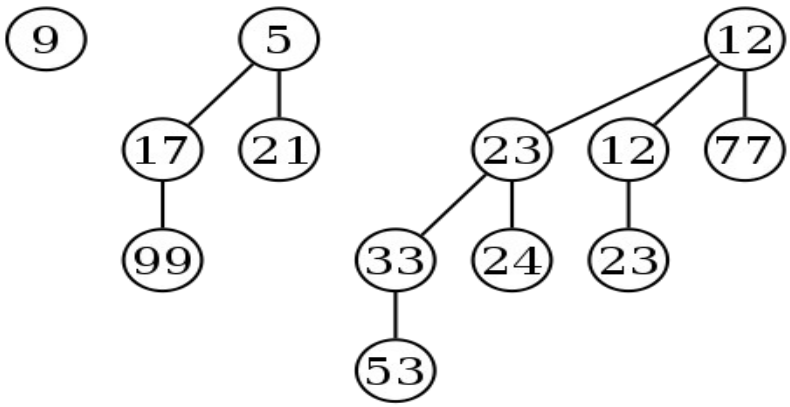
Árboles binomiales

Se definen de forma recursiva:

- Un árbol binomial de orden 0 tiene tan solo un nodo raíz
- Uno de orden k está formado por un nodo raíz, del que cuelgan árboles binomiales de orden *k - 1, k - 2, ..., 0*



Un heap binomial de **n** elementos esta compuesto por un conjunto de árboles binomiales, el orden de los cuales se corresponde con los índeces de los bits que tiene a 1 la representación en base 2 de n



Operaciones básicas

- **Consulta**: Permite conocer el elemento de máximo (o mínimo) valor del heap.
Complejidad: O(1)

- **Inserción**: Añade un nuevo valor al heap.
Complejidad amortizada: O(1)

- **Borrado**: Elimina el máximo (o mínimo) del heap.
Complejidad: O(log n)

Fusión

La fusión de dos heaps binomiales es una operación análoga a la suma de números en base 2

He modelado la siguiente implementación de heap binomial de forma que esta característica sea más o menos obvia, si se examinan las funciones **merge** y **merge_trees**:

```
#include <iostream>
#include <vector>
using namespace std;

struct Node {
  int val;            // valor del nodo
  int ord;            // orden arbol binomial
  vector<Node*> chi;  // vector de subarboles:
                      // debe haber una correspondencia directa entre
                      // el orden del subarbol y su posicion en 'chi', es decir,
                      // en la posicion k solo puede haber un arbol de orden k

  // Limite: permitir arboles binomiales hasta orden 31 (2^31 = 2147483648 nodos)
  Node(int val) : val(val), ord(0), chi(vector<Node*>(32, NULL)) {}
};

typedef vector<Node*> NV;

// Implementacion simple de un max-heap binomial para 'int's
struct MaxHeap {

  int ord;    // orden del arbol binomial mas grande
  Node* max;  // puntero al arbol con mayor valor
  NV trees;   // vector de arboles del heap; igual que en Node,
              // en la posicion k solo puede haber un arbol de orden k (o NULL)

  // Limitar cantidad de valores igual que Node (maximo 2^31 = 2147483648 valores)
  MaxHeap() : ord(-1), max(NULL), trees(NV(32, NULL)) {}

  // Devuelve true si el heap esta vacio
  bool empty() {
    return ord < 0;
  }
```

```
// Actualizar el puntero al arbol con mayor valor
// y el orden del arbol mas grande
inline void update_max(Node* nod) {
  if (nod != NULL) {
    if (max == NULL || max->val < nod->val) max = nod;
    if (ord < nod->ord) ord = nod->ord;
  }
}

// Fusion de dos arboles t1 y t2 no nulos del mismo orden
// El arbol con mayor valor en la raiz sera el padre,
// y el ortro arbol se anadira al vector de subarboles
// El arbol padre incrementa en 1 su orden
// ya que la fusion de dos arboles de orden k resulta en uno de orden (k + 1)
inline Node* merge_trees(Node* &t1, Node* &t2) {
  if (t1->val > t2->val) {
    t1->chi[t2->ord] = t2;
    ++t1->ord;
    return t1;
  } else {
    t2->chi[t1->ord] = t1;
    ++t2->ord;
    return t2;
  }
}

// Fusion de arboles 't1' y 't2' del mismo orden
// 'car' es un arbol del mismo orden que 't1' y 't2'
// Operacion analoga a una suma de numeros en base 2:
//  - si 't1' o 't2' son NULL, su valor respectivo seria 0; si no, 1
//  - 'car' representa el carry: valdria 0 o 1 segun si es NULL o no
inline Node* merge_trees(Node* &t1, Node* &t2, Node* &car) {
  Node* res = NULL;
  if (t1 != NULL && t2 != NULL) {
    // 1 + 1
    // el resultado sera 0 o 1 en funcion de si hay carry o no
    // y el nuevo carry sera siempre 1 ('t1' y 't2' fusionados)
    res = car;
    car = merge_trees(t1, t2);
  } else if (t1 != NULL) {
    // 1 + 0
    // el resultado sera 1 si no hay carry (devolver 't1'), o 0 si lo habia
    // si habia carry, hay que fusionar 'car' y 't1' para obtener el nuevo carry
    res = car == NULL ? t1 : NULL;
    car = car == NULL ? NULL : merge_trees(t1, car);
  } else if (t2 != NULL) {
    // 0 + 1
    // analoga a la suma del anterior if, pero tratando 't2' en vez de 't1'
    res = car == NULL ? t2 : NULL;
    car = car == NULL ? NULL : merge_trees(t2, car);
  } else {
    // 0 + 0
    // el resultado es igual al carry, ya que 't1' y 't2' son NULL
    res = car;
    car = NULL;
  }
  return res;
}
```

```
  // Funcion de fusion del conjunto de arboles de este heap
  // con el conjunto de arboles en 'nodes'
  // 'ord_nod' debe ser el orden del mayor arbol de 'nodes'
  void merge(NV &nodes, int ord_nod) {
    // trataremos los arboles de orden 0 hasta 'max_ord', teniendo en cuenta que
    // la operacion con los arboles de mayor orden puede producir carry
    int max_ord = (ord > ord_nod ? ord : ord_nod) + 1;
    if (max_ord > 31) max_ord = 31;  // respetar limites estructura de datos
    max = NULL;  // resetear 'max' y 'ord' -'update_max' lo actualizara
    ord = -1;

    Node* car = NULL;  // carry inicial = 0
    for (int i = 0; i <= max_ord; ++i) {
      // iterar desde 0 hasta el orden maximo,
      // fusionando los arboles binomiales del mismo orden
      trees[i] = merge_trees(trees[i], nodes[i], car);
      // actualizamos tambien puntero al arbol con valor maximo, y orden del heap
      update_max(trees[i]);
    }
  }

  // Fusionar este heap con 'heap'
  // Complejidad: O(log n)
  void merge(MaxHeap &heap) {
    merge(heap.trees, heap.ord);
  }

  // Consultar el valor maximo en este heap
  // Si el heap esta vacio, devuelve -(2^31)
  // Complejidad: O(1)
  int top() {
    return max != NULL ? max->val : 1 << 31;
  }

  // Anadir el valor 'x' a este heap
  // Basicamente fusiona este heap con un arbol binomial de orden 0
  // y cuya raiz tiene valor 'x'
  // Aunque su complejidad sea en principio O(log n),
  // su coste amortizado (despues de muchas inserciones) es O(1)
  void push(int x) {
    NV nodes(32, NULL);
    nodes[0] = new Node(x);
    merge(nodes, 0);
  }

  // Eliminar del heap su maximo valor
  // El truco es:
  //  - quitar del heap el arbol que contiene el valor maximo,
  //  - considerar los subarboles de ese arbol como un segundo heap,
  //  - y finalmente fusionar ambos heaps
  // Complejidad: O(log n)
  void pop() {
    if (max != NULL) {
      int nod_ord = max->ord - 1;  // orden del mayor subarbol de 'max'
      Node* del = max;             // puntero a 'max' para luego hacer el delete
      NV &nods = max->chi;         // vector de arboles para la fusion (2° heap)
      trees[max->ord] = NULL;      // quitar del heap el arbol con maximo valor
      merge(nods, nod_ord);        // fusionar este heap con 2° heap
      delete del;                  // liberar memoria
    }
  }

};
```

Grafos

Representan un conjunto de elementos
Algunos pares de estos elementos pueden estar conectados mediante enlaces

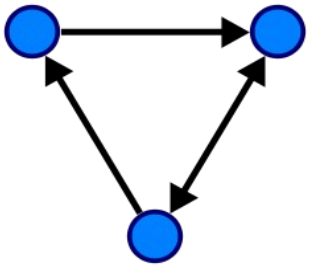
Los elementos se llaman vértices (o nodos); y las conexiones, aristas (o arcos)

Más formalmente, un grafo es un par **G = (V, E)**, donde **V** es el conjunto de vértices, y **E** es el conjunto de aristas

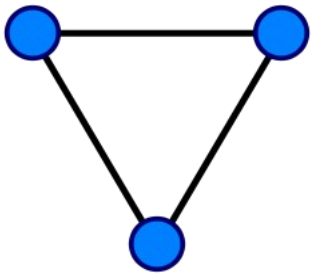
|V| representa el número de vértices
|E| representa el número de aristas

Tres tipos importantes de grafos que veremos:

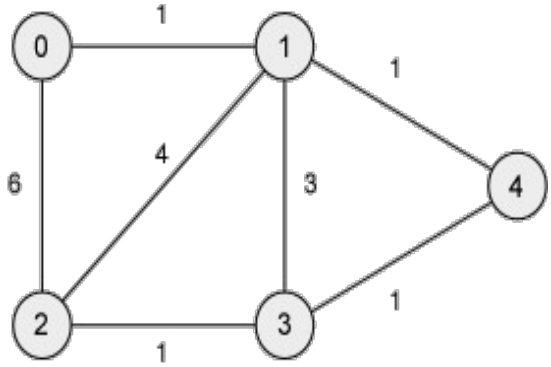
- *Grafo dirigido (dígrafo)*: Las aristas tienen una orientación



- *Grafo no dirigido*: Las aristas se pueden recorrer en ambos sentidos



- *Grafo con pesos*: Las aristas tienen un peso (o coste) asociado, independientemente de si se trata de un grafo dirigido o no



Representaciones de grafo (estructuras de datos)

Matriz de adyacencias

Se trata de una simple matriz de booleanos **A** de tamaño **|V|x|V|**
El elemento **A[i][j]** indica si existe una arista que une el vértice **i** con el **j**

El acceso a cada arista es inmediato – *O(1)* -; sin embargo, la consulta de las aristas conectadas a un vértice concreto es más costosa – *O(|V|)* -, lo cual desaconseja su uso en grafos grandes (muchos vértices) y/o poco densos (pocas aristas)

Lista de adyacencias

Es la forma más habitual de representar un grafo
Consiste en una colección de tamaño **|V|**, que asocia cada vértice con el conjunto de sus vértices vecinos
En C++ suele implementarse con un **vector<vector<int>> G**, donde **G[i]** es el conjunto de vértices conectados al vértice **i**

Algoritmos de búsqueda en grafos

A continuación veremos los algoritmos más utilizados para realizar diferentes tipos de búsqueda en grafos

- **DFS**: Búsqueda en profundidad

- **BFS**: Búsqueda en anchura

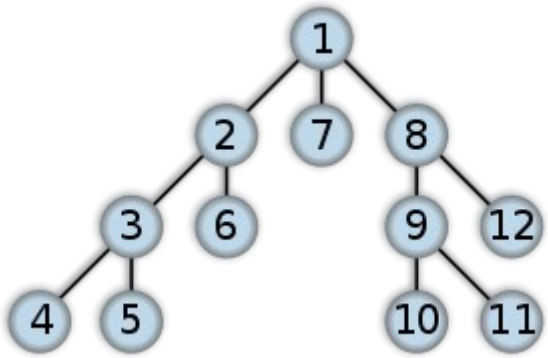
- **Ordenación topológica**: Precedencias en un grafo

- **Dijkstra**: Camino más corto

- **Bellman-Ford**: Camino más corto, competencia transversal

DFS (Depth first search)

Comenzando por el nodo raíz (o un vértice arbitrario), el algoritmo explora nodos tan lejos como sea posible antes de volver sobre sus pasos



Orden en que se exploran los nodos (DFS)

Un ejemplo de DFS es el recorrido en preorden de un árbol

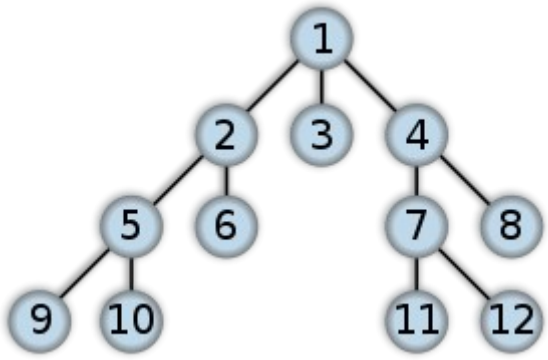
Complejidad: **O(|E|)**

La forma más práctica de implementar el DFS es mediante una función recursiva:

```
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> IV;
typedef vector<IV> G;
typedef vector<bool> BV;


G g;     // grafo: lista de adyacencias
BV vis;  // array de visitados: inicialmente todo a false


void dfs(int x) {
  vis[x] = true;
  // Tratar nodo: en este ejemplo tan solo imprime
  cout << x << endl;
  for (int y : g[x]) {
    if (!vis[y]) {
      dfs(y);
    }
  }
}
```

BFS (Breadth first search)

Comenzando por un nodo raíz (o un vértice arbitrario), el algoritmo explora los nodos en orden de proximidad a esa raíz



Orden en que se exploran los nodos (BFS)

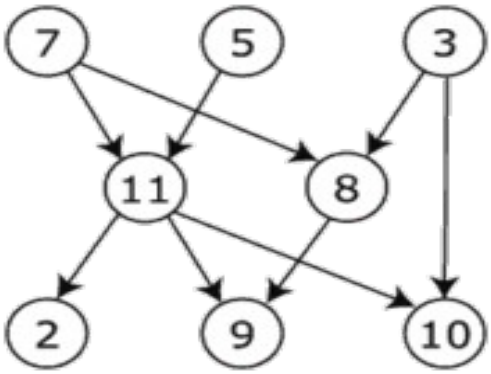
Complejidad: **O(|E|)**

La forma más práctica de implementar el BFS es mediante un método iterativo, utilizando una cola que almacena los vértices que vamos a explorar:

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

typedef vector<int> IV;
typedef vector<IV> G;
typedef vector<bool> BV;
typedef queue<int> IQ;


G g;     // grafo: lista de adyacencias
BV vis;  // array de visitados


void bfs(int x) {
  // añadir el nodo raiz 'x' a la cola
  IQ que;
  que.push(x);
  vis[x] = true;
  // explorar los nodos en el orden en que se han ido descubriendo
  while (!que.empty()) {
    x = que.front();
    que.pop();
    // Tratar nodo (imprimir)
    cout << x << endl;
    for (int y : g[x]) {
      // encolar nodos no visitados
      if (!vis[y]) {
       vis[y] = true;
       que.push(y);
      }
    }
  }
}
```

Ordenación topológica

Es un ordenación lineal de los vértices de un **grafo dirigido y acíclico**, tal que para toda arista dirigida **E = (u, v)**, el vértice **u** aparece antes que **v**



Una posible ordenación topológica:
7, 5, 3, 11, 8, 2, 9, 10

Sea el **grado** de un nodo el número de sus aristas entrantes
El algoritmo consiste en *tratar sucesivamente los nodos de grado 0, eliminando sus aristas salientes* (y así decrementando el grado de sus vecinos)

Complejidad: **O(|V| + |E|)**

```
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

typedef vector<int> IV;
typedef vector<IV> G;
typedef queue<int> IQ;

G g;     // grafo: lista de adyacencias
IV deg;  // array de grados: inicialmente todos a 0

void topo() {
  // contar el grado de cada vertice
  for (int i = 0; i < (int)g.size(); ++i) {
    for (int x : g[i]) ++deg[x];
  }
  // encolar los vertices de grado 0
  IQ que;
  for (int i = 0; i < (int)g.size(); ++i) {
    if (!deg[i]) que.push(i);
  }
  // examinar nodos a los que no les queden aristas incidentes
  while (!que.empty()) {
    int x = que.front();
    que.pop();
    // tratar nodo: en este ejemplo se imprime la ordenación
    cout << x << endl;
    for (int y : g[x]) {
      --deg[y];
      // encolar el nodo vecino si su grado ha quedado en 0
      if (!deg[y]) que.push(y);
    }
  }
}
```

Dijkstra

El algoritmo de Dijkstra nos permite encontrar caminos de coste mínimo en un **grafo con pesos** (no negativos) en las aristas, a partir de un vértice origen

El algoritmo necesita un min-heap para almacenar pares (vértice, distancia), ya que consiste en visitar vorazmente los vértices más cercanos al origen

En realidad, su comportamiento es análogo al del BFS: si en el BFS la distancia a un nodo era el número de predecesores hasta el orígen, aquí la distancia viene dada por el peso de las aristas

Complejidad: **O(|E| + |V| log |V|)**

En el siguiente código C++ podeis ver con más detalle la mecánica del algoritmo:

```
#include <limits>
#include <vector>
#include <queue>
using namespace std;

typedef vector<int> IV;
typedef pair<int, int> IP;
typedef vector<IP> IPV;
typedef vector<IPV> G;
typedef priority_queue<IP, IPV, greater<IP>> IQ;

G g;     // grafo: lista de adyacencias
IV dis;  // array de distancias

void dijkstra(int v) {
  // distancia al nodo origen 'v' = 0; infinito para los demas
  dis = IV(g.size(), numeric_limits<int>::max());
  dis[v] = 0;
  // añadir a la cola el primer par (distancia a nodo, nodo)
  // la distancia debe ser el primer elemento del par, ya que
  // priority_queue ordena los pair segun el primer elemento, y luego el 2°
  IQ que;
  que.push(IP(0, v));
  // tratar sucesivamente los nodos a menor distancia
  while (!que.empty()) {
    const IP &p = que.top();
    int x = p.second;  // nodo que vamos a tratar
    int d = p.first;   // distancia a este nodo
    que.pop();
    // trataremos cada nodo solo una vez:
    // otras apariciones de este en la cola se habran añadido antes
    // y estaran a mayor distancia
    if (d <= dis[x]) {
      // examinar todas la aristas que salen del nodo 'x'
      for (const IP &a : g[x]) {
        int y = a.second;  // nodo vecino
        int c = a.first;   // distancia desde el nodo 'x' hasta el nodo vecino

        // si la distancia que hemos sacado de la cola + distancia al nodo vecino
        // iguala o mejora la aproximación que tenemos guardada en el array 'dis',
        // establecerla y encolar el par (nueva distancia mejorada, nodo vecino)
        if (dis[y] > d + c) {
          dis[y] = d + c;
          que.push(IP(dis[y], y));
        }
      }
    }
  }
}
```

Bellman-Ford (competencia transversal)

Este algoritmo nos permite obtener, al igual que Dijkstra, los caminos de coste mínimo en un **grafo con pesos**; sin embargo, *los pesos de las aristas en este caso sí pueden ser negativos.*

Que haya aristes con peso negativo significa que podemos encontrar "ciclos negativos" -es decir, que la suma de los pesos de las aristas del ciclo sea negativa. En este caso, realmente no existe un camino de coste mínimo, ya que podríamos reducir el coste infinitamente dando vueltas por el ciclo

El algoritmo consiste en establecer aproximaciones cada vez mejores a la distancia correcta a los vértices. Esto se consigue haciendo |V| - 1 iteraciones, donde en cada una recorremos todas las aristas y miramos si el nuevo camino a los vértices destino mejora la distancia previamente establecida

Tras |V| - 1 iteraciones, la distancia a cada vértice será necesariamente la correcta, ya que el camino más largo sin ciclos tendrá como máximo tendrá |V| - 1 arisas

Finalmente, se comprueba la existencia de ciclos negativos examinando una vez más todas las aristas: si la distancia a alguno de los vértices se puede reducir más es que hay ciclo negativo

Complejidad: **O(|V| |E|)**

```
#include <iostream>
#include <vector>
#include <limits>
using namespace std;

typedef pair<int, int> IP;
typedef vector<IP> IPV;
typedef vector<IPV> G;
typedef vector<int> IV;

int n, m;  // numero de vertices y numero de aristes
int s, t;  // vertice origen y vertice destino
G g;       // el grafo: lista de adyacencias
IV dis;    // vector de distancias
IV par;    // vector de padres

// funcion para imprimir el camino hasta el vertice 'x'
void print(int x) {
  if (par[x] == x) cout << x;
  else {
    print(par[x]);
    cout << ' ' << x;
  }
}

void print_sol() {
  if (par[t] != -1) {
    // si el vertice destino tiene un padre, es que podemos llegar
    // imprimir la distancia y el camino
    cout << "Distancia: " << dis[t] << endl;
    cout << "Camino: ";
    print(t);
    cout << endl;
  } else
    // no se puede llegar al vertice destino
    cout << "No hay camino desde " << s << " hasta " << t << endl;
}
```

```cpp
// retorna true si no hay ciclos negativos
bool bellman_ford() {
  // en cada iteracion miramos, para todas las aristes del grafo,
  // si hemos encontrado una distancia menor que la que tenemos
  for (int i = 1; i < n; ++i) {
    for (int x = 0; x < n; ++x) {
      // importante! asegurarse que la distància al vertice 'x' se haya establecido
      // si 'x' no tiene padre, la distancia sera 'numeric_limits<int>::max()',
      // y sumarle la distancia al vertice 'y' no tiene sentido (y puede dar overflow)
      if (par[x] != -1) {
       for (const IP &a : g[x]) {
         int y = a.first;
         int d = a.second;
         if (dis[y] > dis[x] + d) {
           dis[y] = dis[x] + d;
           par[y] = x;
         }
       }
      }
    }
  }
  // para acabar miramos si hay algun ciclo negativo
  for (int x = 0; x < n; ++x) {
    if (par[x] != -1) {
      for (const IP &a : g[x]) {
       int y = a.first;
       int d = a.second;
       if (dis[y] > dis[x] + d) {
         return false;
       }
      }
    }
  }
  return true;
}

int main() {
  // leer num vertices y aristas
  while (cin >> n >> m) {
    // inicializa estructuras datos
    g = G(n);
    dis = IV(n, numeric_limits<int>::max());  // distancias inicialmente a infinito
    par = IV(n, -1);                          // padres no establecidos
    // leer aristas
    for (int i = 0; i < m; ++i) {
      int x, y, d;                 // vertice origen, vertice destino, y distancia
      cin >> x >> y >> d;
      g[x].push_back(IP(y, d));
    }
    // leer origen y destino
    cin >> s >> t;
    dis[s] = 0;     // distancia al vertice inicial es 0
    par[s] = s;     // padre del vertice inicial es el mismo
    // ejecutar bellman-ford
    bool sol = bellman_ford();
    if (sol) {
      // si no hay ciclos negativos, 'sol' sera true
      print_sol();
    } else {
      // cuando hay un ciclo negativo, lo podriamos recorrer infinitas veces
      // y la distancia al destino no pararia de decrementar
      cout << "Ciclo negativo encontrado: no existe camino más corto!" << endl;
    }
    cout << endl;
  }
  return 0;
}
```

Búsqueda exhaustiva (Backtracking)

La búsqueda exhaustiva es una técnica general de exploración de espacios de soluciones; es decir, que permite encontrar todas las soluciones de un problema

Backtracking

Es un algoritmo de búsqueda exhausiva
Según cómo lo programemos, podremos encontrar una o todas las soluciones óptimas de un problema

Consiste en generar un árbol de soluciones, del que alguna(s) ramas representarán soluciones óptimas del problema

Su coste es sumamente elevado, ya que consiste en generar con fuerza bruta todas las posibles soluciones, sin tener la certeza de que estas serán válidas

Una técnica para mejorar su rendimiento es la **poda**, que consiste en descartar soluciones parciales tan pronto como sea posible determinar que no son parte de una soución óptima

Un ejemplo de la utilidad del bactracking es la obtención de subconjuntos y permutaciones

A diferencia del resto de algoritmos vistos hasta ahora, el backtracking requiere una implementación diferente adaptada a cada problema. Afortunadamente, siempre tiene una forma muy similar en términos de pseudocódigo:

```
función backtracking(D : datos problema, S : solución parcial)
    si S es solución válida
        tratar solución S
    si no
        mientras pueda añadir un dato Di de D a la solución S
            añadir Di a la solución S
            marcar Di como utilizado
            backtracking(D, S)
            marcar Di como disponible
            eliminar Di de la solución S
```

Veamos por ejemplo un problema cuya solución se puede obtener con backtracking, aplicando además una poda:

- *Dados dos enteros **n** y **m**, obtener todas las combinaciones de **n – m** ceros y **m** unos*

Y una solución válida implementada en C++:

```
#include <iostream>
#include <vector>
using namespace std;

typedef vector<int> IV;

int n, m;  // numero total de dígitos y numero de ceros
IV sol;    // solución

void print_sol() {
  for (int i = 0; i < n; ++i) {
    if (i) cout << ' ';
    cout << sol[i];
  }
  cout << endl;
}

// función de backtracking
// idx => índice del dígito que intentaremos establecer en la solución
// z   => número de ceros disponibles
// u   => número de unos disponibles
void back(int idx, int z, int u) {
  // si hemos establecido todos los dígitos de la soución, la imprimimos
  if (idx == n) print_sol();
  else {
    // intentar añadir un cero a la solución
    // poda: añadir cero sólo si queda alguno
    if (z > 0) {
      // añadir el cero a la solución
      sol[idx] = 0;
      // intentar establecer el siguiente dígito de la solución (idx + 1)
      // restar uno de los ceros disponibles (z - 1)
      back(idx + 1, z - 1, u);
    }
    // intentar añadir un uno a la solución
    // poda: añadir 1 sólo si queda alguno
    if (u > 0) {
      // añadir el 1 a la solución
      sol[idx] = 1;
      // intentar establecer el siguiente dígito de la solución (idx + 1)
      // restar uno de los unos disponibles (u - 1)
      back(idx + 1, z, u - 1);
    }
  }
}

int main() {
  ios_base::sync_with_stdio(0);
  // leer datos y crear la solución
  cin >> n >> m;
  sol = IV(n);
  // llamada inicial a la función de backtracking:
  // 'idx' deber ser 0 ya que intentaremos establecer el primer dígito de la solución
  // el número de ceros 'z' será igual a (número de dígitos 'n' - número de unos 'm')
  // el numero de unos 'm' nos viene dado
  back(0, n - m, m);
  return 0;
}
```

P y NP

Problemas decisionales: Problemas cuya soución es **SÍ** o **NO**
Tiempo polinómico: O(**n^k**), donde **n** es el tamaño de los datos de entrada, y **k** es una constante no negativa
P: Tiempo polinómico
NP: Tiempo polinómico no determinista

Clase P

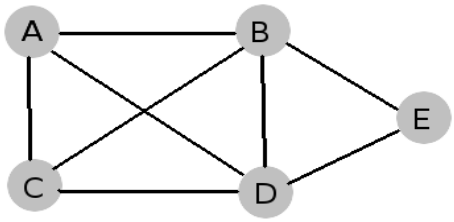
Problemas decisionales que *se pueden resolver en tiempo polinómico*
Se consideran "*fáciles*"

Por ejemplo, una búsqueda: dado un conjunto de **n** enteros **A = { a1, a2, …, an }**, determinar si el número **x** pertenece a **A** => solucionable en O(n)

Clase NP

Problemas decisionales que no se pueden resolver en tiempo polinómico, pero que, *dada una solución, ésta se puede verificar en tiempo polinómico*

Ejemplo: **k-Clique** → Dado un grafo **G = (V, E)**, determinar si existe un subset $V' \subseteq V$ de tamaño **|V'|** = **k** tal que todos los vértices de **V'** están conectados entre ellos



Si el grafo **G** es el de la figura, y **V' = { A, B, C, D }**, podemos verificar que la solución es **SÍ** comprobando que existe una arista desde cada uno de los nodos de **V'** al resto de nodos de **V'**

Pregunta: $P \subseteq NP$?

Sí; si un problema se puede solucionar en tiempo polinómico, su verificación también se puede hacer en tiempo polinómico

Por ejemplo, la ordenación de un conjunto **A** de **n** enteros: se puede realizar en *O(n log n)* con el merge-sort, y su verificación a1 ≤ a2 ≤ … ≤ an en *O(n)*

Clase NP-Hard

Un problema **H** es NP-Hard si todos los problemas en NP se pueden reducir a **H**

Los NP-Hard son al menos tan *difíciles* como los problemas NP más *difíciles*

Ejemplos:

- *Subset-sum* → dado un conjunto de enteros, determinar si existe un subconjunto tal que la suma de todos sus elementos sea igual a 0

- *Traveling salesman* → problema de optimización que pide encontrar el camino cíclico más barato que pase por todos los nodos de un grafo con pesos

Clase NP-Complete

Son problemas que pertenecen a:
- **NP**
- **NP-Hard**

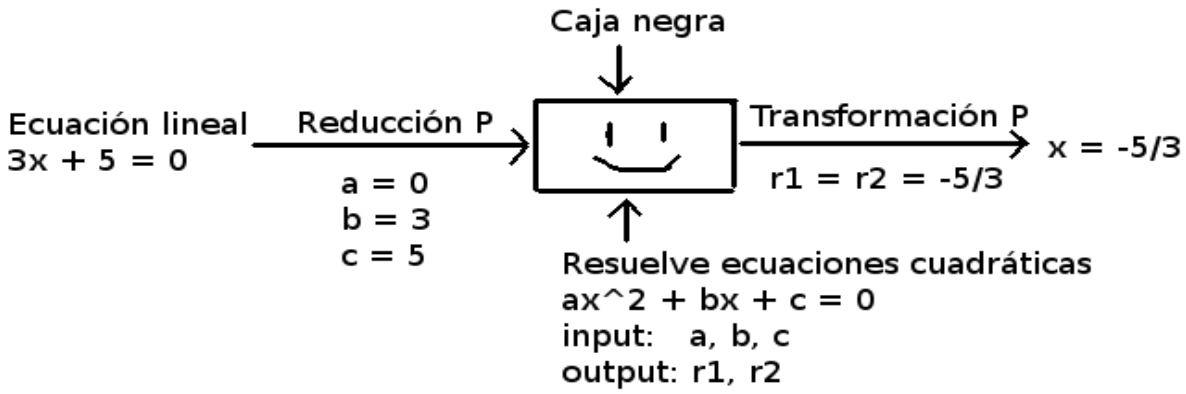
Los NP-Complete son los problemas más *difíciles* de NP

Forman parte de esta clase, por ejemplo, los problemas mencionados anteriormente: k-clique, subset-sum, y traveling-salesman

Reducciones P

Se trata de las transformaciones en tiempo polinómico que convierten un problema en otro

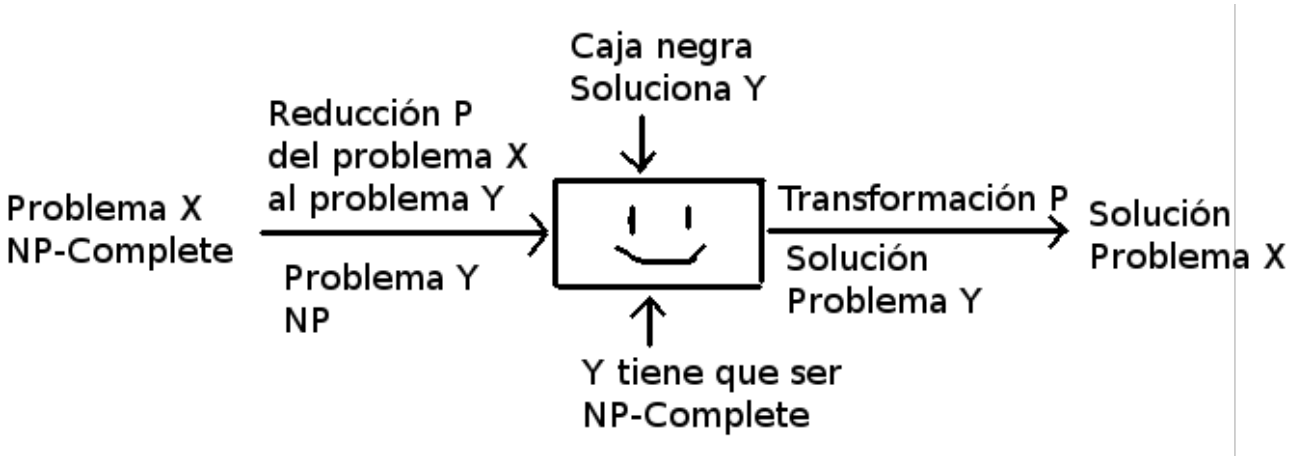
Ejemplo: Tenemos una "caja negra" que resuelve ecuaciones cuadráticas
Queremos resolver una ecuación lineal, y para ello sólo disponemos de la "caja negra" -habrá que reducir el problema de la ecuación lineal al que la "caja negra" puede solucionar (ecuación cuadrática)



Demostrar que un problema Y es NP-Complete

Matemáticamente es muy complicado; afortunadamente existe una forma relativamente sencilla:

- Demostrar que el problema **Y** es NP
- Encontrar una reducción P de un problema **X** NP-Complete a **Y**



La reducción debe ser en tiempo polinómico, ya que si fuera de cualquier otra clase, no sería posible aseguar que el problema Y es NP-Complete

Si obtenemos una solución al problema **Y**, podemos obtener una solución para el problema **X** realizando otra vez una transformación, por lo que *el problema **Y** tiene que ser NP-Complete*

Imaginemos que **Y** se puede resolver en P, y mediante una transformación podemos convertir la solución de **Y** en solución de **X**, entonces podriamos resolver **X** en tiempo polinómico, lo cual es una contradicción ya que sabemos que **X** es NP-C