# Modelo Relacional

**RDBMS** –Relational Data Base Managemet Systems / o llamado motor de base de datos (MySQL, SQL Server, Postgred) es un programa que permite administrar los contenidos de una o más base de datos almacenadas en disco

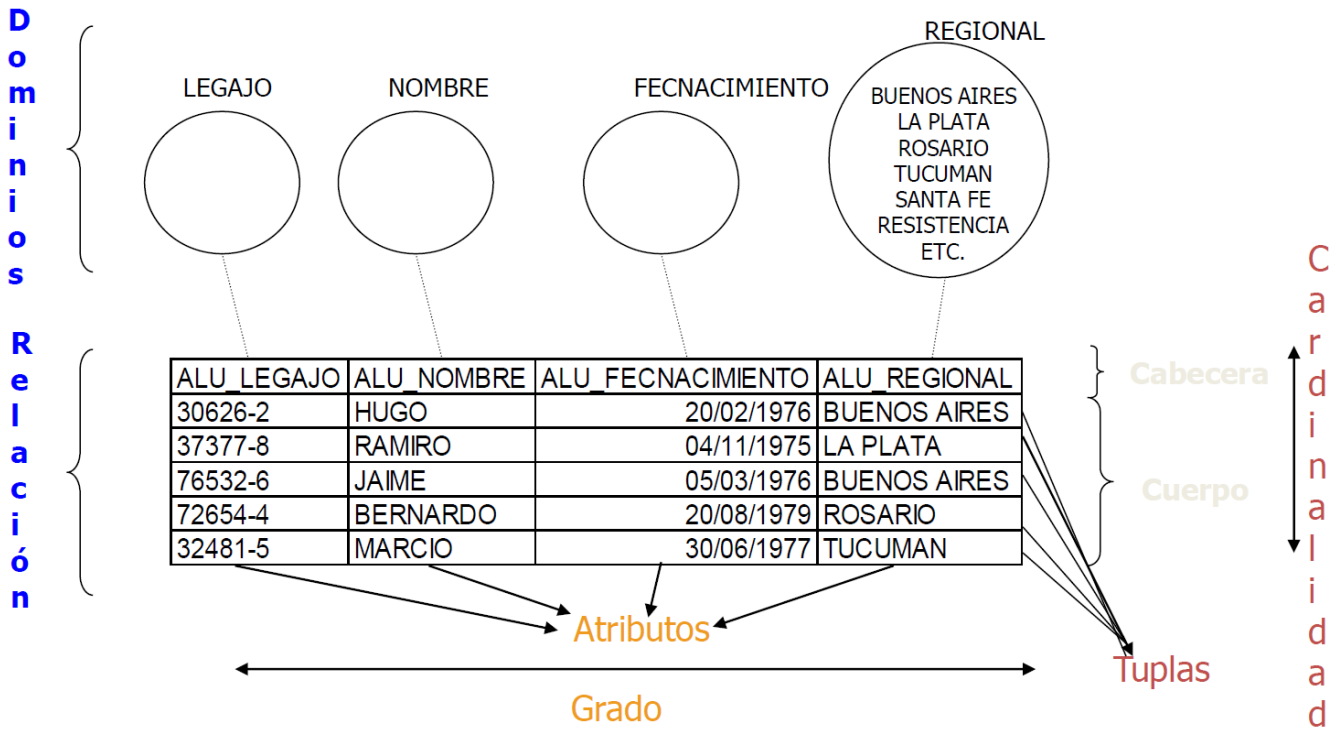
## Modelo Relacional

El concepto fundamental en el que se basa el modelo relacional es que los datos se representan de una sola manera, como una estructura tabular, conformada por filas y columnas.
A esta estructura se la definió formalmente como relación, aunque de manera informal y en sus implementaciones se la conoce con el nombre de tabla.

Propuesto por el Dr. E.F. Codd en 1970, consta de:

1. **Estructura**: Recopilación de objetos o relaciones
2. **Manipulación de datos**: Operaciones sobre las relaciones para producir otras
3. **Integridad**: Para obtener precisión y consistencia

# Estructura




### 1.1 Dominio

Colección de valores, de los cuales los atributos obtienen sus valores reales
No contienen nulos. Conjunto de valores escalares de igual tipo
**Implementación**
Restricciones de Chequeo
Claves Foráneas (si tengo un elemento que es FK -> tiene un dominio definido)

### 1.2 Relación (TABLA)

- **Cabecera**: Conjunto fijo de pares atributo-dominio (nombre - Dominio nombre)
- **Cuerpo**: Conjunto de tuplas que varía con el tiempo.
    - **Tupla(FILA)**: Conjunto de pares atributo-valor (nombre: “Juan”, apellido: “Perez”)
- **Tabla**: Representación de una relación.

**CAMPO**=Intersección de fila y columna (nulo o no nulo)

**Claves candidatas** : Las que podrían identificar unívocamente la tupla
Claves candidatas = Claves Alternas + **PK**
Ej: Si tengo los atributos legajo y dni → defino a legajo como mi PK → dni sería CA que aunque podría identificarme unívocamente, se optó por legajo

## 3. Integridad

REGLA DE INTEGRIDAD DE LAS ENTIDADES
Cada fila de cada tabla pueda ser identificada de forma unívoca

Se consigue con el constraint PK

**PK:** Ningún componente de la clave primaria puede aceptar nulos

REGLA DE INTEGRIDAD REFERENCIAL

La base de datos no debe contener valores no nulos de clave foránea para los cuales no **exista** un valor concordante de clave primaria en la relación referenciada.
Se consigue con el constraint FK o también con Triggers , Views

**FK**: Atributo (o conjunto de atributos) de una relación (R2) cuyos valores (no nulos) deben coincidir con los de la clave primaria de una relación (R1).

Determinar acciones a llevar a cabo ante operaciones que puedan violar la integridad referencial
**Eliminación**: RESTRICT / CASCADE / SET NULL
**Modificación**: RESTRICT / CASCADE / SET NULL(setearle un null)
**Inserción**: RESTRICT(No dejarte hacerlo)

## Independencia de Datos

**Independencia lógica:** Se podrían hacer cambios en tablas sin que afecten a las aplicaciones que no los requieren. query= SELECT nombre, apellido. Borrar una columna que no sea esas dos y no afecta a la query
**Independencia física:** Es posible modificar la estructura de almacenamiento, la distribución física o la técnica de acceso sin afectar las aplicaciones. Voy a seguir accediendo, quizá más rápido

En el pasado tu BD (BDMS) es relacional? Edgar Codd definió:

- Independencia entre el motor de base de datos y los programas que acceden a los datos
- Los datos y los metadatos(structs de las tablas) deben estar en tablas dentro de la BD
- Motor debe asegurar integridad(reglas también guardadas en la BD)
- Soportar información faltante mediante valores nulos (NULL)
- Proveer lenguajes para:
    - Definición de datos
    - Manipulación de datos: debe haber operaciones para insertar, eliminar, actualizar o buscar
    - Definición de restricciones de seguridad, restricciones de integridad, autorización y delimitar una transacción.

**Álgebra Relacional**: Para poder entender como un Motor de Base de Datos resuelve una determinada consulta.
`Las Claves compuestas:` La composición es una abstracción, siempre busca 1 solo valor(lo que hace la bd es concatenar a modo de Strings para reducir a 1 solo valor la clave)

# ESTRUCTURAS DE DATOS

Persistir la Información, como? → (Base de Datos - Estructuras Datos)
**GRAFOS:** Es la idea en sí, una cosa abstracta, no existe en la realidad computacional, para existir deben reflejarse a través de una representación computacional. Estas representaciones pueden ser de dos tipos dinámicas o estáticas

## REPRESENTACIÓN COMPUTACIONAL DE GRAFOS

**DINÁMICAS:** El espacio consumido para representar computacionalmente al grafo, concuerda exactamente con la cantidad de nodos y vértices a representar,esto es que no se consideran todas las posibilidades de relación posibles, sino que solo se representa lo que ocurre en este momento

- Se representan a través de estructuras linkeadas:
  - **LISTA DE ADYACENCIAS**: Define una lista enlazada para cada nodo, que contendrá los nodos a los cuales es posible acceder. En este caso el espacio ocupado es O(V + A), muy distinto del necesario en la matriz de adyacencia, que era de O( $V^2$ ).
  - **LISTA DE GRAAL**: Es igual que las listas de adyacencia nada más que en vez de guardar el nodo en la lista, se guarda un puntero al nodo.
  - **REPRESENTACIÓN DE PFALTZ:** Relación dinámica de nodos y arcos enlazados mediante punteros, ocupando espacio en memoria a medida que realmente se necesita y liberándose cuando se efectúa una baja. Es más eficiente que matrices.

**ESTÁTICAS**: el espacio consumido para representar al grafo es fijo respecto a la cantidad de nodos y vértices a representar. n : cant nodos, m : cant aristas

- **MATRIZ DE ADYACENCIAS**: se asocia cada fila y cada columna un nodo del grafo, (n x n)
- **MATRIZ DE INCIDENCIAS**: se asocia una fila con un nodo y cada columna con una arista del grafo (m x n)

## CARACTERÍSTICAS DE LOS GRAFOS

- **GRADO**: Cantidad de aristas que tocan un vértice(v)
  - **Positivo**: Salen del v | **Negativo**: Llegan a v
- **CAMINO**: Conexión entre dos vértices mediante uno o más arcos.No me importa si es dirigido (`Longitud = #Nodos - 1`)
- **PASO**: Camino dirigido entre dos Nodos.(N → → → N)
- **CICLO**: Es un camino donde no se recorre dos veces la misma arista, y donde se regresa al punto inicial. Un ciclo hamiltoniano tiene además que recorrer todos los vértices exactamente una vez

Estructuras de Datos (Solo son 4): `PILAS, COLAS, LISTAS y ÁRBOLES`
Todo lo demás como por ej: vector (son representaciones computacionales, puedo meter alguno de esos 4)

(*) Cuando tenemos relaciones con `SIMETRÍA` pura → Es el único caso que tiene sentido usar `Grafo no dirigido`( - equiv a <->), todo lo demás es con `Grafos dirigidos(→)`

## TIPOS DE GRAFOS

**G. COMPLETO**: Todas las relaciones posibles existen
**G. LIBRE**: Lo opuesto al completo
**G. REGULAR**: Todos los vértices tienen igual grado = g (Se toma el positivo)
**G. SIMPLE:** Una arista cualquiera es la única que une dos vértices específicos.
**G. COMPLEJO**: Lo opuesto (N = N, 2 aristas entre 2 v's)
**G. CONEXO:** Cada par de vértices está conectado por un camino. Es fuertemente conexo si cada par de vértices está conectado por al menos dos caminos aislados (disjuntos)
**G. COMPLEMENTARIO**: `G' = G.Completo - G`

Algunas definiciones:
**G. Reflexivo**: Todos los Nodos están relacionados consigo mismo
**G. A-Reflexivo**: Algunos | **G. ANTIReflexivo**: Ninguno

## CLASIFICACIÓN

**Por su restricción**

- **G. RESTRICTO**: `ANTIReflexivo + ANTISimétrico + ANTITransitivo`
  *Expresado desde el concepto y no desde el ejemplo*: No puede haber bucles, Si voy hacia un lado no puedo volver y Si puedo llegar a un lugar a través de alguien, no puedo llegar directamente
  ¿Por qué restringir el modelo? Sera mas facil de manejar y programar
- **G. IRRESTRICTO:** Lo opuesto a restricto(basta con que uno no cumpla)
  Google maps es un grafo irrestricto ponderado(las relaciones tienen un valor) | AUTO 5 min | A PIE 15 min. También Facebook (*Final 2022 Febrero*)

**Por su direccionalidad**

- G. Dirigido y G. No Dirigo

Una **estructura de datos** es un grafo dirigido y restricto, con la característica de `unicidad(grado negativo = 1 ↔ solo llega 1 arista a cada nodo)`
Lo dividimos en dos:
**ED Biunívocas** : grado positivo = 1 (→ N →) | `PILAS, COLAS, LISTAS`
`PILA : Anula la jerarquía, lo apila viéndolo verticalmente(al inicio)`
`COLA : Respeta la jerarquía, que se encola lo nuevo(al final)`
`LISTA : Administra la jerarquía`
`(*) Tambien los puedo identificar como su forma de sacar un elemento`

**ED Unívocas** : grado positivo >= 1 | `ÁRBOL`
**ED Complejas**: Combinación de las ED, ej árbol de pilas, lista de listas

¿Para qué me sirven las ED en la vida real? Fácil el **SO**(base de la computación) está formada por: `pila de procesos, cola de impresión, lista de interrupciones, árbol de directorios.`

**ÁRBOL**: Es mejor porque tengo 2 opciones o más, en la lista solo 1 por nodo
Una lista la puedo armar de mayor a menor o viceversa, pero no las dos (frente a esto el árbol gana)
***Final**: Cualquiera de las 4 ED puede representarse computacionalmente como un vector (incluido el arbol)*

# ÁRBOL

**Definiciones**: Biunívoco, `Nivel, Grado, Profundidad = Cantidad de Niveles`
**Crecimiento exponencial** , en la cantidad de elementos (x cada nodo tiene más nodos hijos)
`Cantidad máxima de elementos x nivel = grado^nivel`
`Cantidad máxima de elementos total = grado^#niveles - 1 -> niveles > log elementos`
**Grado de un árbol:** Grado máximo de los nodos de un árbol

*Entonces si tengo 14 elementos en una lista me tomara como máximo 14 busquedas(por que tiene 14 niveles), en cambio en un árbol me tomara como máximo 4 búsquedas(porque tiene 4 niveles)*

Árbol crece exponencialmente → `búsqueda logarítmica`
A diferencia de las otras 3 E.D. que tienen **crecimiento lineal** → `búsqueda` **lineal**

**Representación computacional**
**Vector**: Se lee de izquierda a derecha por nivel
[0,1,2,3,4,5,6] (Los números son los índices de las posiciones)
Si el **grado = 2** y hay **niveles = 3**
`[0 | 1,2 | 3,4,5,6] → ( nivel 0 | nivel 1 | nivel 2 )`

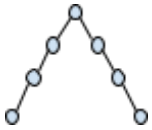
```
Si tengo el padre 0 → [1, 2] Hijos izquierdo y derecho
           padre 1 → [3, 4]
           padre 2 → [5, 6]
```

Generalizamos: `Padre x → [2*x+1, 2*x+2]` / tendria ‘g’ hijos
Si tengo el hijo 5 , cuál es su padre?
5 % 2 = 1 , osea que es un “2k+1” → hijo izquierdo
El padre sería : 5 = 2*x+1 → x=2

Si el árbol es de grado 4, calcular el padre de x
(x-4)/4 ← x , Si x es múltiplo de 4  (x-1)/4 ← x , Si x es múltiplo de 4 + 1

**Características**

**Árbol lleno**: si es un **árbol de altura k** , tiene **g^k - 1 nodos**
**COMPLETO:** Si todos los nodos tienen grado=n o grado=0(son hojas)
**BALANCEADO:** Si lo agarro de la raíz y no balancea. Todos los subárboles(los hijos de la raíz) pesan lo mismo o tienen una diferencia indivisible.
**PERFECTAMENTE BALANCEADO**
Los árboles AVL están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa. Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad O(log n)
**PERFECTO : COMPLETO + P.B.**


**Ejemplo de un g. balanceado pero no perfecto**

**Para un árbol, entre más completo y P.B.** → (+)Performante, voy atener que hacer menos consultas en una búsqueda para encontrar el valor deseado
Ej: *un árbol de 8 niveles con 8 elementos es casi tan malo como una lista de 8 elementos*

# BÚSQUEDA

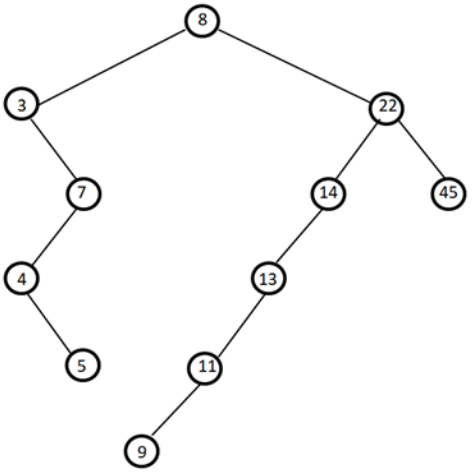
**ÁRBOL BINARIO DE BÚSQUEDA (ABB)**

Un ABB como lo indica su nombre es un árbol de grado 2 diseñado para buscar como método alternativo a una lista representada en un vector, donde los elementos menores se ingresan a la izquierda y los mayores a la derecha

Dado el siguiente conjunto de números desordenados:

8 – 22 – 14 – 13 – 45 – 11 – 3 – 7 – 4 – 5 - 9

Si utilizamos un ABB nos quedaría lo siguiente




## BARRIDOS

Un BARRIDO de un árbol es la forma de leer el mismo, si bien existe una forma de recorrer un árbol que es de arriba hacia abajo y de izquierda a derecha por convención occidental, existen tres formas distintas de leer los elementos que conforman tres barridos diferentes

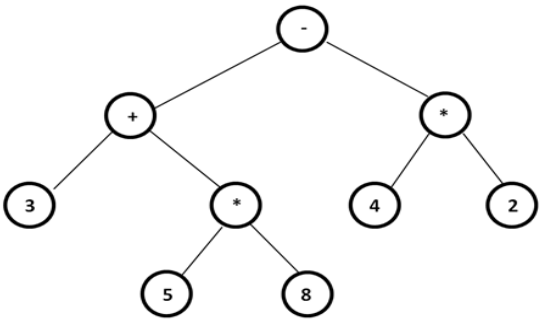
```
barrido(son 3) != recorrido(siempre es igual)
```

El algoritmo de barridos es siempre igual, dado que un árbol se basa en la recursividad, lo único que se modifica es el momento en que se lee el nodo.

```
if root
      Preorden  ->  printf(root->dato);
      recorrer(root->izq);
      Inorden   ->  printf(root->dato);   ó Simétrico (solo en árboles binarios)
      recorrer(root->der);
      Postorden ->  printf(root->dato);
```

Si a dicho árbol se barre inorder se obtiene la expresión matemática en notación INFIJO, si se lo barre postorden se obtiene la expresión matemática en notación POSTFIJO o POLACA INVERSA

**INORDEN:** 3 + 5 * 8 – 4 * 2
**POSTORDEN:** 358*+42*-




(**) A este **árbol** se lo llama **de expresión**, sirve para evaluar expresiones como operadores (+,-,*,/)

(***) Se nota rápidamente que la cantidad de nodos siempre es par en un árbol de expresión

# Metodos de Clasificacion

Un motor de BD es igual a un SO(sin manejo de perifericos)

**CLASIFICACIÓN – IN SITU**
No requieren de una gran cantidad de memoria extra para transformar una estructura de datos. Ej ordenamiento, sin la necesidad de un array auxiliar

**CLASIFICACIÓN – INTERNA Y EXTERNA**
**INTERNA:** Corra en la Memoria Principal  **EXTERNA:** Corra en Memoria Secundaria(Disco)

**CLASIFICACIÓN – COMPLEJIDAD**
Cuán complejo es para la computadora encontrar la respuesta, usando mi algoritmo
Ej: piloto automático, la máquina de turing en la película que demoraba

Algorítmicamente: Todo tiene solución y se divide
**complejidad P :** pueden ser resueltos en una máquina determinista en tiempo a lo sumo polinómico
**complejidad NP :** No se cuando va a terminar (no determinista)

**¿Cómo calculamos la complejidad?**
Contando la cantidad de **comparaciones**
`for(int c=0; c < a.leng/2; c++)` -> Hay n/2 comparaciones -> O(n/2)

El ser humano es lineal, acepta la proporcionalidad (regla de 3) cualquier algoritmo que supere la linealidad(recta), no es aceptable computacionalmente(o sea para la compu) , ademas de no ser aplicable comercialmente

# Métodos de ordenamiento

**Elementales: Todas en el peor caso es O(n^2)**
**Bubble Sort:** El más lento de todos(pero el más rápido si viene ordenado).
**Selection Sort:** Comienza buscando el elemento más pequeño del array y se lo intercambia con el que está en la primera posición.
**Insertion sort:** Se recomienda cuando el array está casi ordenado y tiene pocos elementos.

**Complejo**
**Shell sort:** Si bien fue una mejora al Insertion Sort, en algunos casos se terminaba teniendo un tiempo de ejecución de O(n^2)
**Merge sort:** Adecuado para trabajos en paralelo.(desv) Requiere memoria extra proporcional a la cantidad de elementos del array
Se busca completar un árbol completo balanceado O(nlog(n))
**Heap sort:** Es el método más acotado en el tiempo,No requiere memoria adicional. Consiste en armar un árbol heap(árbol binario completo) en el cual la clave de cada nodo es mayor o igual a la clave de sus hijos. Siempre es aprox O(nlog(n)) independiente de cómo llegan los datos
**Quick sort:** El más rápido en la práctica. Es in-situ ya que solo usa una pila auxiliar. El peor de los casos se da cuando viene una lista ordenada(vemos que tenemos un árbol con solo nodos hijos izquierdos = casi tan mala como O(n^2))

¿Por qué vimos todo esto? Como el motor BD hace la consulta ORDER BY

# ÍNDICES

El objetivo es crear una **estructura adicional** a la tabla que permita mantener los datos ordenados en función de alguna clave. Por ejemplo la creación de una **PRIMARY KEY** o un índice **UNIQUE**
Los índices son **Estructuras de Almacenamiento Secundario**, que tienen pares (clave, posición)
*Se crea una tabla de hash x cada índice que cree*
Hay desventajas en crear muchos índices? Sí señor, porque por cada nuevo registro o fila se debe agregar ordenadamente a cada tabla hash que tenga. En consecuencia podemos mencionar que una tabla sin PK es mas rapida que una con PK (para Grabar, Borrar,etc) `/Más índices -> Más espacio/`

`Tenemos 2 técnicas para crearlas, dependerá de la arquitectura`
**`Técnicas`**
`Hashing:` `Tablas de hashing, almacenamiento predimensionado(Estático)`
`Árbol-B:` `Nace por la PC (Dinámico)`

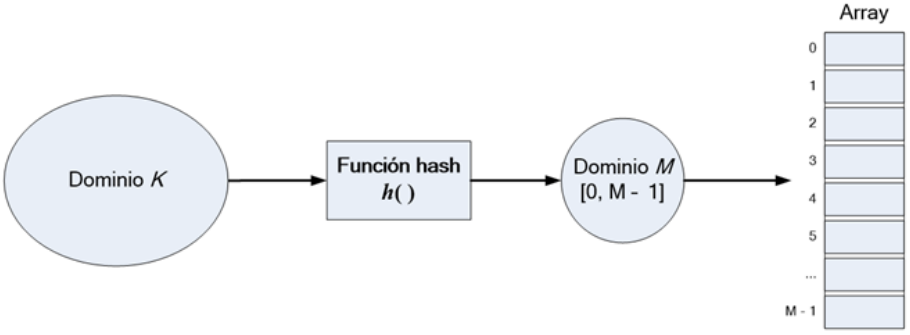
**`Arquitecturas`**
**`Mainframe:`** `Guarda secuencialemente (llega a tener huecos)` **`Mas caro`** `-> Mas rápido y seguro`
**`PC:`** `Guarda dinámicamente (fragmenta para aprovechar los huecos)Un archivo puede estar particionado por todo el disco`

**`Tipos de Acceso`**
`Secuencial: Secuencial a los datos`
`Secuencial Indexado: Secuencial pero al índice`
`Directo o Random: A una clave sin realizar ningún recorrido`

`A-B es más performante para la búsqueda secuencial o por rangos`
`Hashing es más performante para para la búsqueda por acceso directo o secuencial indexada`

**`Funciones del hash`**: `Se considera que una buena función de hash es aquella que tiene las siguientes cualidades:` **`Evitar colisiones, Tiende a distribuir las claves uniformemente, Es fácil de calcular`**
`EJ:`

- `Método de la` **`división o del módulo`**`: h(k)=mod(k)503`
- `Método del` **`cuadrado medio`**`: h(k)=k^2 escogo los digitos del medio`
- `Método de` **`doble`**`: h(k)= bin(k1) OR_exc bin(k2)`




Dominio K = CHAR -> 256 posibilidades, \ -> hash -> “Tamaño Fijo”
VARCHAR(4) -> 256^4 posibilidades /
Al achicar el dominio de entrada -> Produce Colisiones [h(x)=h(y), si x != y]
Por lo general las funciones hash son: Sen(x), Cos(x). Osea donde el dominio de salida es REAL(un número real), pero al tener que adaptarlo a un entero -> género colisión

`Las Claves compuestas: La composición es una abstracción,` siempre busca 1 solo valor(lo que hace la bd es concatenar a modo de Strings para reducir a 1 solo valor la clave)

Cómo salvar las colisiones

- **Encadenamiento** (Dinámico)
- **Direccionamiento abierto** (Estático)

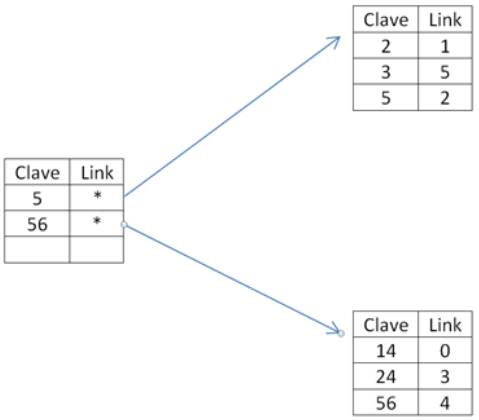
Encadenamiento: Más rápido pero ocupa más espacio

Direccionamiento abierto: Comparó mas elementos -> Mas lento, pero ocupa menos espacio
Cuando un *registro no puede ser ubicado en el índice calculado* por la función de hash, se busca otra posición dentro de la tabla. Hay varios **métodos de elección** de esa posición, los cuales varían en el método para buscar la próxima posición.
Los más usados son:

- **Sondeo Lineal**: busca secuencialmente en la tabla hasta encontrar una posición vacía
- **Sondeo cuadrático**: El nombre cuadrático deriva de la fórmula utilizada $F(i) = i^2$ para resolver las colisiones. Si resulta que la función de hash evalúa a la posición y la posición en es inconclusa, se intentan las posiciones $H+1^2, H+2^2, H+3^2, \ldots, H+i^2$
- **Hashing doble**: aplica la función hash a la clave una segunda vez, usando una función de hash distinta y usa ese resultado como tamaño de salto

## Árbol-B

Es un árbol **N-ario** (rango predefinido, no necesariamente contendrá datos\ **V**: no estar reequilibrando **D**\Desperdicio de espacio).Fragmenta la tabla de hash en forma de hojas de un árbol. Por la naturaleza de la E.D. **O(log m(n))**
Hay 2 tipos de nodos
Nodo Raíz o Rama: Tiene una tabla donde van los valores de las claves ordenados de menor a mayor y un componente puntero que apunta al nodo que contiene claves menores o iguales que ella.




Nodo Hoja: Tiene una tabla donde van los valores de las claves ordenados de menor a mayor y un componente puntero que contiene la posición relativa de los datos secuenciales correspondientes a esa clave

| Clave | Link |
|---|---|
| 14 | 0 |
| 24 | 3 |
| 56 | 4 |

Notamos que si pegamos las tablas de los nodos hoja formariamos una tabla hash
SI yo tengo un árbol normal el primer elemento que se crea es un nodo raíz, este es el primer elemento (*5) este nodo nació raíz y morirá raíz, además de que es hoja circunstancialmente(para este caso que esta solo se considera que también es hoja)
(*5 -> *7) Ahora el nodo (*5) deja ser catalogado como nodo hoja
Vemos que va creciendo hacia las hojas

Árbol-B el primer elemento que se crea es un nodo hoja
Vemos que va creciendo de las hojas hacia la raíz
***Búsqueda:*** Buscar en un árbol-B es muy parecido a buscar en un árbol binario de búsqueda

**Inserción**: Para insertar un elemento x, comenzamos en la raíz y realizamos una búsqueda para él. Asumiendo que el elemento no está previamente en el árbol, la búsqueda sin éxito terminará en un nodo hoja. Este es el punto en el árbol donde el x va a ser insertado.

**Split:** Si ocurre que cuando se llega a la hoja no hay espacio para insertar el nodo se produce lo que se denomina split que es un proceso que divide el nodo en dos dejando la mitad de elementos en cada uno respetando el orden de menor a mayor, quedando la mitad de los elementos más chicos en un nodo y la mitad de los elementos más grandes en el otro.

**Eliminación*:*** Para eliminar un elemento x, comenzamos en la raíz y realizamos una búsqueda para él. Asumiendo que el elemento existe, si existe se llegará a la hoja donde está y se borra, si no se dirà que no existe.

**Fusión**: Si ocurre que cuando se elimina el elemento x el nodo queda vacío, debe eliminarse el nodo, lo que puede generar una baja potencial en todos los antecesores de dicho nodo.

(*) Hace los cambios cuando no lo estas usando (en computación se llama Background) para que vos no sientas el impacto del tiempo

**Load Factor**: Porcentaje de carga(solo se aplica en proceso de carga inicial) de los nodos del Árbol B

```
100% -> Pocas actualizaciones
75%  -> Muchas actualizaciones (pq tengo mas huecos -> menos splits)
```

# Compresión

Objetivo: Reducir el espacio que ocupa un archivo. Puede ser con o sin pérdida de información

**Algoritmo de Huffman**

Huffman es un algoritmo de compresión de datos sin pérdida, que es muy eficiente con archivos de texto
El problema radica en el alfabeto ASCII que contiene caracteres innecesarios.
Las computadoras codifican los caracteres mediante ASCII o Unicode, utilizando un tamaño fijo para cada uno de ellos, 1 byte en el primer caso y 4 bytes en el segundo

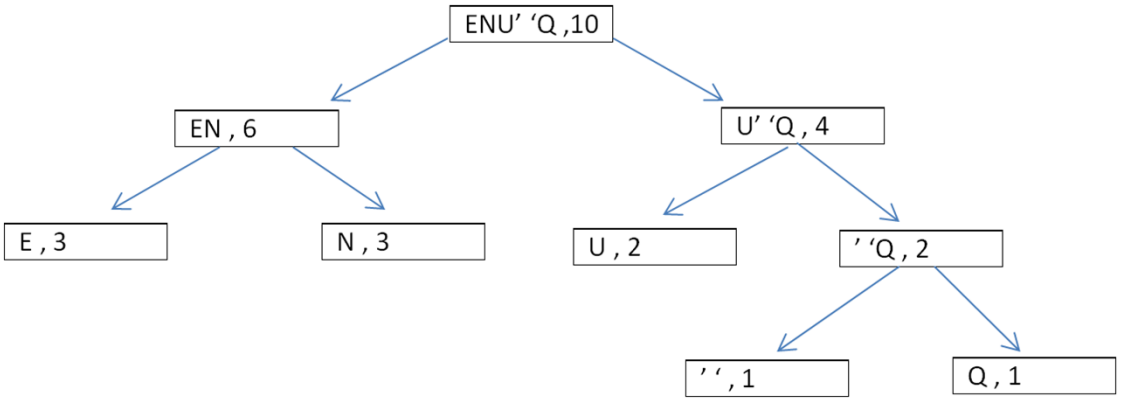
**Proceso de Compresión:** Se comienza leyendo el archivo e identificando todos los caracteres distintos que lo componen. Esos caracteres identificados se almacenan conjuntamente con la cantidad de repeticiones del mismo en un vector

`'EN NEUQUEN'`

Lo podemos visualizar en este vector, donde lo ordenamos de mayor a menor cantidad de repeticiones

| E | N | ‘ ‘ | U | Q |
|---|---|---|---|---|
| 3 | 3 | 1 | 2 | 1 |

Por último crea un árbol binario dividiendo el vector de a dos en función de la cantidad de repeticiones de cada carácter, comenzando con todo el vector como raíz del árbol y llegando hasta que las hojas esten compuestas por un solo carácter




| Usamos la convención 0 izquierda - 1 derecha |
|---|

Es necesario guardar el vector en archivo comprimido, por ese motivo los archivos muy pequeños no se comprimen. Porque el vector puede pesar más que el archivo a comprimir y no estaría reduciendo el espacio. No necesito guardar el árbol(recordemos que es un abstracción), ya que este se genera del vector