RESUMEN COMPUTACIÓN

1. *Estructura de datos y algoritmos*

→ ayuda a determinar la complejidad del algoritmo

1.1 Orden de complejidad

/ compara ambos algoritmos / tiempo y recursos para ejecutar un algoritmo.

Corresponde a la cantidad de tiempo y/o espacio cuantificado que requiere un algoritmo para ejecutar una función en virtud del largo de su input.

La ***notación asintótica*** es una forma de expresar "el orden de complejidad" de un Algoritmo basados en su tasa de crecimiento. Existen tres tipos:

- **Notación Ω:** Conocida como cota inferior asintótica es una función que sirve de tope inferior de otra función cuando el argumento tiende a infinito. → mejor caso posible.
- **Notación Θ:** Conocida como cota ajustada asintótica es una función que sirve de "tope" superior e inferior de otra función cuando el argumento tiende a infinito. → caso promedio complejidad.
- ✱ **Notación O:** Conocida como cota superior asintótica, es una función que sirve de "tope" superior de otra función cuando el argumento tiende a infinito.
  → peor caso de un algoritmo

Propiedades Notación O: (son igual de válidas para las otras notaciones)

1. $O(f) + O(g) = O(f + g)$	Regla de la suma
2. $O(k \cdot f) = k \cdot O(f) = O(f), k\ cte.$	Constantes multiplicativas
3. $O(f) \cdot O(g) = O(f \cdot g)$	Regla de la multiplicación
4. $f \geq g \Rightarrow O(f + g) = O(f)$	Cotas
5. $f \geq g \Rightarrow (h \sim O(g)\ entonces\ h \sim O(f))$	Transitividad

Los órdenes más utilizados en análisis de algoritmos, en orden creciente, son los siguientes **(donde "c" representa una constante y "n" el tamaño de la entrada):**

empeora en tiempo que nos demoraremos para el algoritmo

notación	nombre	
$O(1)$	orden constante	→ bueno/best
$O(\log \log n)$	orden sublogarítmica	
$O(\log n)$	orden logarítmica	→ good
$O(n \cdot \log n)$	orden lineal logarítmica	→ bold.
$O(\sqrt{n})$	orden sublineal	
$O(n)$	orden lineal o de primer orden	→ fair
$O(n^2)$	orden cuadrática o de segundo orden	
$O(n^3)$, ...	orden cúbica o de tercer orden, ...	
$O(n^c)$	orden potencial fija	
$O(c^n)$, n > 1	orden exponencial	→ worst
$O(n!)$	orden factorial	
$O(n^n)$	orden potencial exponencial	

- **Tiempo constante O(c):** Un algoritmo es considerado de Tiempo Constante cuando se da que el tamaño del input no cambia lo que se demora en realizar el algoritmo. Esto es lo **ideal**
- **Tiempo Logarítmico O(log n):** Aquellos donde cada nueva operación tomará la mitad de tiempo con respecto a la anterior. Esto se considera **óptimo**.

COMPLEJIDAD: PEOR CASO [Notación O].

- recorrer matrices → n^3 / diapo 11
- imprimir un FOR → n / diapo 10
- busqueda binaria → O(log n) / diapo 12

- **Tiempo Lineal - O(n):** Un algoritmo se llama lineal cuando el tiempo que se demora en ejecutar es dependiente del largo del input. Esto se considera **normal.**
- **Tiempo Polinomial - O(n elevado a k):** Un algoritmo es limitado superiormente por una expresión polinomial según el tamaño del input. Esto ocurre con **mucha frecuencia.**

1.2 Algoritmos de ordenamiento

Los algoritmos de ordenamiento son un conjunto de instrucciones que toman un arreglo o lista como entrada y organizan los elementos en un orden particular.

Invariante: condición que se sigue cumpliendo después de la ejecución de determinadas instrucciones permaneciendo sin variación.

- **Bubble Sort:** algoritmo de ordenamiento que compara todos los elementos de un array (excepto el último) con su vecino de la derecha e intercambia de posición si están "fuera de orden". Esto es mueve siempre el mayor hacía "a la derecha", el último elemento siempre será el mayor (el segundo mayor queda penúltimo... y así sucesivamente) y compara los elementos y los intercambia si están fuera de orden.

Si no hay swap ← algoritmo ordenado

$O(n^2) \Longleftrightarrow$ ←

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp

alist = [54,26,93,17,77,31,44,55,20]
print(alist)
bubbleSort(alist)
print(alist)
```

```
[54, 26, 93, 17, 77, 31, 44, 55, 20]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

El resultado de las operaciones sería de $n * n/2 + k$, en notación implica $O(n^2/2 + k) = O(n^2)$

- **Selection Sort:** Es un algoritmo de ordenamiento que compara todos los elementos de un arreglo buscando el mínimo elemento entre una posición i y el final de la lista e intercambia dicho mínimo con el elemento de dicha posición i-ésima.

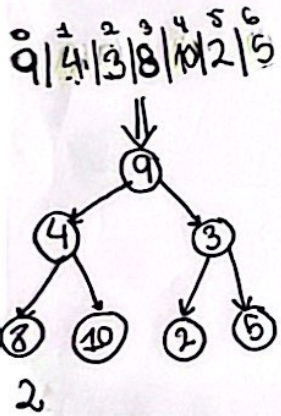
```
def selectionSort(alist):
    for outter in range(len(alist)-1,0,-1):
        positionOfMax=0
        for inner in range(1, outter +1):
            if alist[inner ]>alist[positionOfMax]:
                positionOfMax = inner

        temp = alist[outter]
        alist[outter ] = alist[positionOfMax]
        alist[positionOfMax] = temp

alist = [54,26,93,17,77,31,44,55,20]
print(alist)
selectionSort(alist)
print(alist)
```

```
[54, 26, 93, 17, 77, 31, 44, 55, 20]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Lo anterior implica que se debe considerar como $O(n^2)$



- heapSort: ordeno y dejo el elemento mas grande como el nodo mas grande; comparo nodo padre con hijos; nodop > nodohijo en contenido.
→ transformo un arreglo a un arbol de nodos.
¡Si nodop no es mayor a nodoh se hace una operación, haciendo un swap entre el nodo padre al nodo hijo.
→ complejidad: O(nlogn)

2

Ejemplos de estructuras de datos.

→ **Arreglos:**
el uso es para almacenar datos en a secuencia indexada.
Ej: lista de temperaturas diarias.

→ **Listas enlazadas:**
Cuando necesitas insertar y eliminar elementos con frecuencia, especialmente en posiciones intermedias.
Ej: implementar una cola de tareas.

→ **Pilas:**
Almacenar elementos con acceso tipo LIFO (Last in, First out).
Ej: sistema de deshacer/rehacer en editores.

→ **Colas:**
Almacenar datos con acceso tipo FIFO (First IN, First out)
Ej: sistema de atención en línea.

→ **Tabla Hash:**
Mapeo clave-valor para acceso rapido
Ej: Almacenar inventario con nombres de productos claves.

→ **Arboles binarios de busqueda (BST):**
Eficiente para busquedas ordenadas.
Ej: Almacenar un indice de búsqueda.

→ **Grafos**
Modelar relaciones entre objetos.
Ej: representar un mapa de ciudades y rutas.

FP 23|1|10|6|2|
SP 1|23|10|6|2
TH 1|23|23|6|2.
1|5|10|23|2
1|2|5|10|23

- **Insertion Sort:** Algoritmo que cuando hay k elementos ordenados de menor a mayor, toma el elemento k+1 y se compara con todos los elementos ya ordenados, se detiene cuando encuentra un elemento menor o cuando ya no se encuentran elementos, insertando en ese punto el elemento k+1 debiendo desplazarse los demás elemento. → $O(n^2)$

ej. inserciones de datos hasta ordenarlo.

Parto de arreglo desordenado, sin tocar el primero y comparo con los otros, moviendo hacia la derecha el otro y así ordeno sucesivamente.

```
def insertionSort(alist):
    for outer in range(1,len(alist)):
        minVal = alist[outer]
        i = outer - 1
        while (i>=0):
            if (minVal < alist[i]):
                alist[i+1] = alist[i]
                alist[i] = minVal
                i = i - 1
            else:
                break

alist = [54,26,93,17,77,31,44,55,20]
print(alist)
insertionSort(alist)
print(alist)
```

```
[54, 26, 93, 17, 77, 31, 44, 55, 20]
[17, 20, 26, 31, 44, 54, 55, 77, 93]
```

Por lo tanto, el tiempo requerido para una insertion sort de un arreglo de n elementos es proporcional a $n^2/4$. Lo anterior implica que, obviando constantes, se debe considerar como $O(n^2)$

1.3 Estructuras de datos básicas

→ forma de organizar y almacenar info.

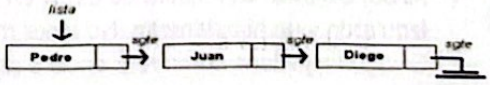
Estructura de Datos: forma de organizar un conjunto de datos elementales con el objetivo de facilitar su manipulación.

- **Lista enlazada:** Es una secuencia de nodos, en los que se guardan campos de datos arbitrarios y una o dos referencias, enlaces o punteros (punteros) al nodo anterior o posterior.
- **Nodos:** Contenedor (de datos) que posee dos partes fundamentales, espacio para la información y un puntero.
- **Listas enlazadas simples:** Aquella que tiene un enlace por nodo y apunta al siguiente nodo en la lista, o al valor NULL o a la lista vacía, si es el último nodo.
- **Lista enlazada simple circular:** Aquella que tiene un enlace por nodo y apunta al siguiente nodo de la lista o al primero si es el último nodo. → algoritmo de scheduling.

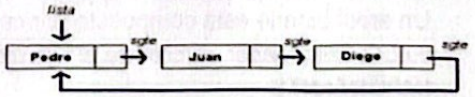
Nulo final de la lista último nodo y se accede a través del nodo cabeza

toman un O(1) peor caso O(n) para recorrerla

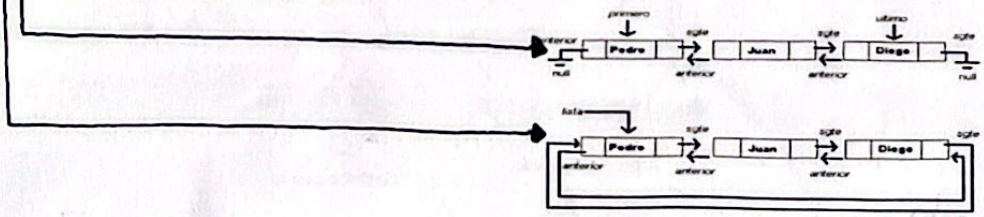
Lista enlazada simple →



Lista enlazada simple circular →



- **Listas doblemente enlazadas:** Aquella que tiene dos enlaces por nodo y apuntan al siguiente y al anterior nodo en la lista. El anterior del primero es nulo y el siguiente del último es nulo.
- **Listas doblemente enlazadas circulares:** Aquella que tiene dos enlaces por nodo y apuntan al siguiente y al anterior nodo en la lista. El anterior del primero es el último y el siguiente del último es el primero.



- Arreglos: no se puede agregar info de manera estática.
- tabla de hash: diccionarios asociados a una llave y un dato vinculado.
  → colisiones se abordan con encadenamiento o dirección abierta (lista enlazada y lineal respectivamente), se usa en una búsqueda (datos de bancos, cuentas, etc).

recursos compartidos entre consumidores (impresoras)...

front: retorna primer elemento de la cola.
agrego elemento al final [enqueue]
elimino el primero de la cola [dequeue]

- **Colas:** Es una lista ordenada en la que el modo de acceso a sus elementos es de tipo FIFO. Un nodo para las colas posee sus elementos(atributos), un puntero hacia atrás o adelante solo uno. Nodo centinela primero y último. → priority queue: cola ordenada.
- **Pilas:** Es una lista ordenada en la que el modo de acceso a sus elementos es de tipo LIFO. Un nodo para las colas posee sus elementos(atributos), un puntero hacia arriba o abajo solo uno. Y su nodo centinela es el tope.

se usa para algoritmo de busquedas

→ agrego primero no sale hasta que todos salen.

→ push: agregar, pop = sacar, top = primer elementos } O(1)

1.4 Estructuras de datos avanzadas

→ redes de buyes, problemas de mochila, cubrimiento, etc.

- ❖ ***Teoría de Grafos:*** Un grafo G=(V,E) es una pareja ordenada en la que V es un conjunto no vacío de vértices y E es un conjunto de aristas.

Estos poseen:

- **Vértice:** El vértice es el elemento (nodo) que compone un grafo (un nodo). → pueden o no contener info.
- **Grado:** Es la cantidad de conexiones que posee un vértice.
- **Camino:** Es el conjunto de vértices interconectados a través de una "ruta" de aristas. Es decir, el recorrido que se hace para llegar de un nodo X a un nodo Y en un Grafo.
- **Arista:** Es la línea que une los vértices de un grafo. Esta puede ser dirigida (de origen a destino) o no dirigida). Entre las aristas, tenemos: → tienen un costo.
  - **Aristas adyacentes:** Aristas que convergen en el mismo vértice.
  - **Aristas paralelas:** Aristas cuyo si el vértice inicial y final son el mismo.
  - **Aristas cíclicas:** Comienza y termina en el mismo vértice.

Algoritmos de grafos: Prim/Kruskal ; Dijsktra.

Tipos de Grafo:

- **Grafo Simple:** Es aquel donde al menos una arista une dos vértices cualesquiera.
- **Grafo Conexo:** Es aquel donde cada par de vértices está conectado por un camino.
- **Grafo Completo:** Es aquel donde todos los pares de vértices tienen una arista que los une.

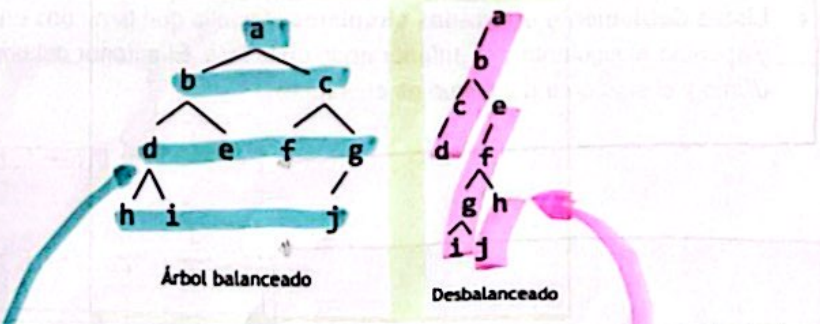
→ aplicaciones (mapa / rutas para transporte / redes, etc) / redes sociales, relaciones de parentesco).

- ❖ ***Árbol Binario:*** Estructura de datos en la cual cada nodo puede tener un hijo izquierdo y un hijo derecho. No tener más de dos hijos (de ahí el nombre "binario"). Si algún hijo tiene como referencia a null, es decir que no almacena ningún dato, entonces este es un nodo externo, caso contrario el hijo es llamado un nodo interno. Un árbol binario está compuesto por cero o más nodos donde cada nodo contiene: valor o contenedor, referencia al hijo izquierdo(left) y referencia al hijo derecho(right).

IMPORTANTE

→ Un árbol binario puede estar vacío, pero si no está vacío, tiene un nodo raíz (root)

→ Cada nodo del árbol puede ser alcanzado desde la raíz siguiendo una única ruta.



tipos de busqueda:

- busqueda en anchura: busco todo el nivel y luego paso al otro. [colas]
- busqueda en profundidad (DFS): ir a lo mas profundo antes de retroceder [pilas]

4 [se alternan para que sea mas optimo].

1.5 Algoritmos de Recorrido en Profundidad

- **PreOrden:** nodo raíz, nodo izquierda, nodo derecha.
- **PostOrden:** nodo izquierdo, nodo derecha, nodo raíz.
- **InOrden:** nodo izquierda, nodo raíz, nodo derecha.

1.6 Arboles AVL

Los árboles AVL son árboles binarios "autobalanceados". Esto se hace a través del factor de balance (alto subárbol derecho menos el alto del subárbol izquierdo). Todos los AVL tiene un factor de balance igual a: -1, 0, o 1

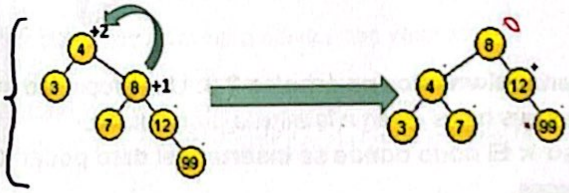
- **Factor de Balance = h(subárbol derecho) – h(subárbol izquierdo)**

Rotaciones en Árboles AVL: Se tiene un arbol de R raiz, D hijo derecho e I hijo izquierdo.

Rotación simple a la izquierda: Se hace cuando se desbalancea el árbol por la inserción de un nodo en la hoja externa del lado.
ÁRBOL NUEVO SERÁ:

- raíz nueva: raíz hijo derecho.
- hijo derecho nuevo: hijo derecho de D
- hijo izquierdo nuevo: nuevo árbol que tendrá como raíz, la raíz R del árbol original, izquierdo de D será el hijo derecho y el hijo izquierdo del árbol original será el hijo izquierdo del subárbol.

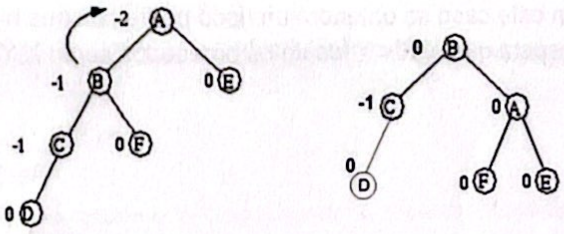
rotar una vez a la izquierda y pasar hijo de arbol derecho al nuevo arbol izquierdo.



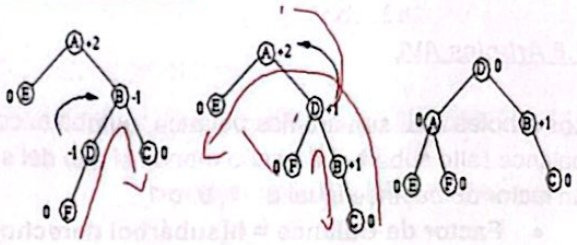
Rotación simple a la derecha: Se hace cuando se desbalancea el árbol por la inserción de un nodo en la hoja externa del lado izquierdo.
ÁRBOL NUEVO SERÁ:

- raíz nueva: raíz hijo izquierdo.
- hijo izquierdo nuevo: hijo izquierdo de I.
- hijo izquierdo nuevo: nuevo árbol que tendrá como raíz la raíz del árbol original, el hijo derecho de I será el hijo izquierdo y el hijo derecho del árbol original será el hijo derecho del nuevo subárbol.

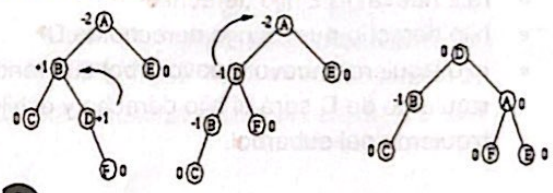
rotar una vez a la derecha y pasar hijo del arbol izquierdo al nuevo arbol derecho



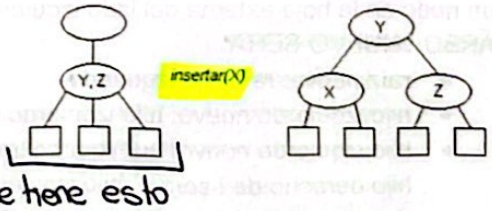
Rotación Doble de Izquierda a Derecha: Se hace cuando se desbalancea el árbol por la inserción de un nodo (izquierdo o derecho) como hijo del izquierdo del hijo derecho de la raíz del árbol.
Se hace una Rotación Simple a la Derecha sobre el subárbol derecho y, luego una Rotación Simple a la Izquierda sobre el árbol completo.



Rotación Doble de Derecha a Izquierda: Se hace cuando se desbalancea el árbol por la inserción de un nodo (izquierdo o derecho) como hijo del derecho del hijo izquierdo de la raíz del árbol.
Se hace una Rotación Simple a la Izquierda sobre el subárbol izquierdo y, luego se hace una Rotación Simple a la Derecha sobre el árbol completo.

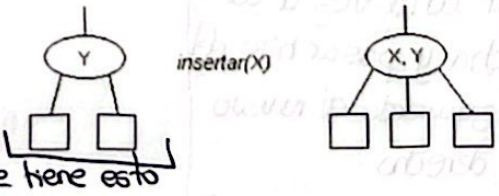


Insertar elementos en árboles 2-3: Una propiedad que poseen los árboles 2-3 es que todas sus hojas están a la misma profundidad.
Caso 1: El nodo donde se insertará el dato posee dos datos (datos Y y Z) y tres enlaces.
En este caso se obtendrá un nodo padre con dos hijos donde se respeta que X < Y < Z donde X, Y y Z son los "de peso" de los nodos.



caso 1: se hace esto

Caso 2: El nodo donde se insertará el dato posee un dato (dato Y) y dos enlaces.
En este caso se obtendrá un nodo padre con tres hijos Nulos y con dos elementos donde se respeta que si X < Y los datos ordenados serán X,Y respectivamente.

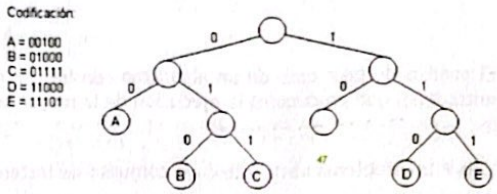


CASO 2: se hace esto

1.7 Arbol Digital → Se usan en busqueda.

Son árboles donde los elementos se almacenan en los nodos internos de igual forma que en los árboles binarios, pero sus llaves están dadas por bits y, en base a ello, se dan sus ramificaciones. Entonces los elementos se representan como una secuencia de bits de la forma: **X = b1b2b3...bn**

Luego, la posición de inserción de un elemento ya no depende de su valor, sino de su representación binaria en un alfabeto definido. Por ello, no todas las hojas contienen elementos.



Para buscar en un árbol digital un elemento X se procede de la siguiente manera:

✓ Se examinan los bits bi del elemento X, partiendo desde b0 en adelante.
✓ Si bi = 0 se avanza por la rama izquierda y se examina el siguiente bit, bi+1.
✓ Si bi = 1 se avanza por la rama derecha y se examina el siguiente bit.

El proceso termina cuando se llega a una hoja, único lugar posible en donde puede estar insertado X.

1.8 Tablas de Hash

Una tabla hash es una estructura de datos que asocia llaves o claves con valores.
Se tienen 3 pasos para las funciones de las tablas de hash:
Paso 1: Representar la llave de manera numérica en un conjunto determinado.
Paso 2: Separar nuestro "número" para tener componentes más pequeños y facilitar la operación con ellos.
Paso 3: Dividir por un número primo (representado en el alfabeto) y usar el resultado como dirección.

1.9 Análisis mejor, peor y caso promedio

Tiempo de Ejecución de Mejor Caso

Sea P un problema abstracto cuyo conjunto de instancias es I y sea A un algoritmo que resuelve el problema P. Luego:

- Para una instancia $x \in I$, *sea* $T(x)$ el tiempo de ejecución del algoritmo A para la instancia x.
- Entonces, el tiempo de ejecución de mejor caso para A sobre una instancia de tamaño n se define como:

$$t_A(n) \equiv \min_{\substack{X \in I \\ |X|=n}} \{T_A(X)\}$$

En la práctica, el análisis de mejor caso de un algoritmo consiste en calcular el tiempo de ejecución considerando la(s) instancia(s) que produce(n) la ejecución de la menor cantidad posible de operaciones.

• Quick-sort: puntero [pivote], comparo elemento pivote con todos los e... Si hay uno mayor al pivote, aparece un puntero 2 y se deja en dicho elemento. Si el elemento es mas pequeño que el pivote, intercambio el pivote 2 por el numero menor al pivote; una vez recorrido todo cambio el pivote por la posición del puntero 1 y esa parte esta ordenada.

Tiempo de Ejecución de Peor Caso

Sea P un problema abstracto cuyo conjunto de instancias es I y sea A un algoritmo que resuelve el problema P. Luego:

- Para una instancia $x \in I$, *sea* $T(x)$ el tiempo de ejecución del algoritmo A para la instancia x.
- Entonces, el tiempo de ejecución del peor caso para A sobre una instancia de tamaño n se define como:

$$T_A(n) \equiv \max_{\substack{x \in I \\ |x|=n}} \{T_A(x)\}$$

El análisis de peor caso de un algoritmo consiste en calcular el tiempo de ejecución considerando la(s) instancia(s) (o entrada(s)) que produce(n) la ejecución de la mayor cantidad posible de operaciones elementales.

Sea P un problema abstracto cuyo conjunto de instancias es I y sea A un algoritmo que resuelve el problema P. Luego:

- Para una instancia $x \in I$, *sea* $T(x)$ el tiempo de ejecución del algoritmo A para la instancia x.
- Entonces, el tiempo de ejecución del caso promedio para A sobre una instancia de tamaño n se define como:

$$\overline{T}_A(n) \equiv \frac{1}{|I_{P,n}|} \sum_{x \in I_{P,n}} T_A(x)$$

1.10 Regla del Límite

Dadas dos funciones $f(n)$ y $g(n)$ asociadas a algoritmos diferentes que abordan un mismo problema, es normal que queramos saber si una de dichas funciones crece más rápido que la otra. Una manera de averiguar esto es mediante la Regla del Límite.

Si consideramos la siguiente expresión:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = k$$

Podemos decir que:

- Si $k \in \mathbb{R}^+$ es una constante (distinta de cero), significa que ambas funciones son equivalentes en complejidad.
- Si $k = \infty$ significa que $f(n)$ tiene un orden de crecimiento mayor que $g(n)$
- Si $k = 0$ significa que $g(n)$ tiene un orden de crecimiento mayor que $f(n)$

1.11 Ecuaciones de recurrencia

Una ecuación de recurrencia es una relación que define una secuencia recursiva; cada término de la secuencia es definido como una función de términos anteriores. Por ejemplo:

$$Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}$$

Ecuaciones de Primer Orden

Son aquellas en que el valor de T(n) depende, principalmente, del valor de T(n-1) - sin que intervengan otros valores de T(n-x). Luego, el caso general para ecuaciones de primer orden está dado por:

$$T(n) = aT(n-1) + b_n$$

Siendo $a > 0$ y b_n una función lineal de n.

Ecuaciones lineales con coeficientes constantes

Son aquellas en que el valor de T(n) depende, de varios valores de T(n-x). Además, estas ecuaciones son de la forma:

$$f_n = Af_{n-1} + Bf_{n-2} + Cf_{n-3} + \cdots$$

Donde A, B y C son constantes. Además, se sabe que estas ecuaciones **siempre** tienen soluciones de la forma:

$$f_n = \lambda^n$$

1.12 Teorema Maestro

Es un "grupo de fórmulas" que permiten encontrar la complejidad de algoritmos recursivos del tipo "Divide and Conquer" que cumplen con un conjunto de reglas específicas en tres casos fundamentales para su "función de trabajo". La forma general de las ecuaciones que se pueden resolver es:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Donde $a \geq 1$, $n > b > 1$ y $f(n)$ es una función polinómica continua positiva

Casos de Resolución

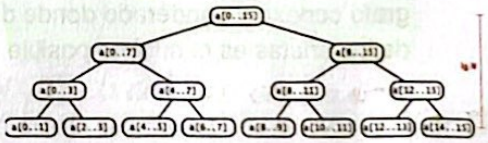
- Considerando que $f(n) \in O(n^c)$ $\forall$ $c < \log_b a$ se tendrá que $T(n) = O(n^{\log_b a})$
- Considerando que $f(n) \in \Omega(n^c)$ $\forall$ $c > \log_b a \wedge af\left(\frac{n}{b}\right) \leq kf(n); k < 1 \wedge n \to \infty$ se tendrá que $T(n) = \Theta(n^c)$
- Considerando que $f(n) \in \Theta(n^c log^k n)$ $\forall$ $c = \log_b a$ se tendrá que $T(n) = \Theta(n^c log^{k+1} n)$

1.13 Divide And Conquer

Consiste en solucionar un problema dividiendo el mismo, de manera recursiva, en dos o más subproblemas similares.

¿Cómo funciona?

- Dividir en subproblemas (caso recursivo)
- Resolver los subproblemas al caso más pequeño (caso base recursión)
- Al combinar las soluciones (return)

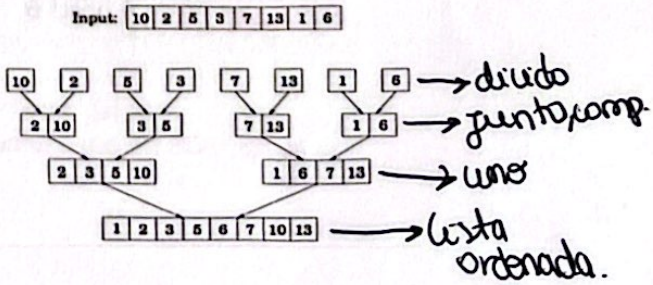


Ejemplo: Merge Sort → O(n log n)

- Divide → El problema (arreglo) se separa en dos partes. Tiempo constante D(n) = Θ(1).
- Conquer → Resolver recursivamente en dos subproblemas de tamaño n/2 cada uno. → 2T(n/2).
- Merge → Combinar las dos mitades ordenadas, en un tercer arreglo ordenado. Función de trabajo con n combinaciones en promedio → C(n) = Θ(n)

→ se ordena de a pares.

→ divide en pares y va ordenando.



1.14 Decrease And Conquer

Explotar la relación entre la solución al problema más pequeño y el problema original: La solución del problema pequeño es usada para construir la solución al problema original.

Cuenta con dos enfoques:

- **Top down:** Se comienza con el problema original, el que se va haciendo más pequeño a cada paso.
- **Bottom up:** Se resuelve el problema comenzando por soluciones muy pequeñas a partes del problema original. Incremental, la solución se construye paso a paso.

Cuenta con dos formas de decrecer:

- **Decrecer por una constante:** En cada paso, el algoritmo hace decrecer el tamaño del problema en un valor constante.
- **Decrecer por factor constante:** En cada paso, el algoritmo hace decrecer el tamaño del problema en un factor constante, por ejemplo, a la mitad tamaño del problema original, o a un tercio, etc.
- **Decrecer por factor variable:** En cada paso, el algoritmo hace decrecer el tamaño del problema en un factor variable, es decir, no necesariamente decrece por el mismo factor en cada paso.

Ejemplos:

Insertion Sort: Decrecer y conquistar de forma bottom up ya que se ordena incrementalmente en cada paso.

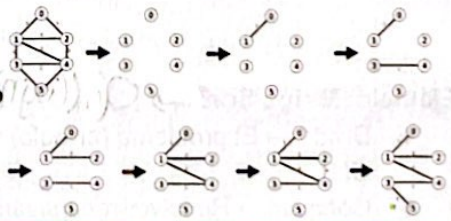
Búsqueda Binaria: Decrecer y conquistar de forma top down ya que se genera, cada vez, un arreglo "más pequeño" (no se busca en el sub arreglo sino hasta el "caso base").

1.15 Algoritmos codiciosos (Greedy)

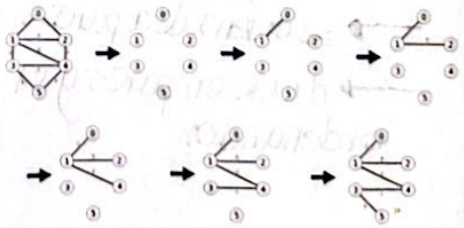
Es aquel algoritmo que, para resolver un determinado problema, sigue una heurística consistente en elegir la opción óptima en cada paso local con la esperanza de llegar a una solución general óptima.

- **Algoritmo de Kruskal:** Algoritmo para encontrar el árbol de cobertura mínimo en un grafo conexo y ponderado donde dicho árbol incluye todos los vértices y el peso total de las aristas es el mínimo posible.

→ ordeno aristas de menor a mayor y las agrego, elijo hasta llegar al ciclo las cuales descarto y luego tengo el arbol de cobertura mínima
→ ordeno y elijo.



- **Algoritmo de Prim:** MISMO QUE ARRIBA. Esto lo hace al optimizar las decisiones basado en el vértice "pivote" y el peso de la arista asociada al mismo y/o si es que existe un "camino mínimo" de un vértice A a un vértice B pasando con un vértice C que el directo de A hacia B.

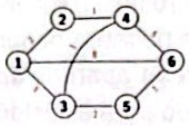


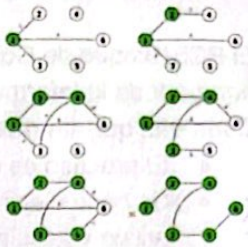
• orientados a encontrar un arbol de cobertura mínima; un arbol de cobertura esta formado por los vertices de un grafo y un subconjunto de sus aristas.
→ toman la mejor decision en base a lo que se tengo.

- **Algoritmo de Dijkstra:** Es un algoritmo para encontrar la ruta más corta – de menor peso – en un grafo desde un vértice A hasta un vértice B pasando por "n" vértices intermedios, donde n > 0, y cada arista tiene un "peso" asociado p > 0. Las aristas pueden ser "dirigidas" (digrafo).

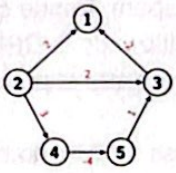
→ grafo dirigido y aristas positivas

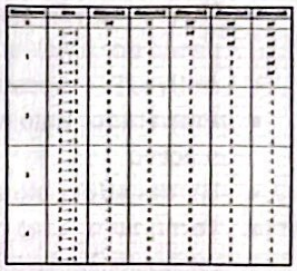
→ lo uso cuando quiero saber la ruta de menor costo, hacia todos los nodos





- **Algoritmo de Bellman-Ford:** Es un algoritmo para encontrar la ruta más corta – de menor peso – en un grafo dirigido desde un vértice A hasta un vértice B pasando por "n" vértices intermedios, donde n > 0, y cada arista tiene un "peso" asociado p.





Programación dinámica: Es una técnica que consiste en resolver los subproblemas una sola vez, guardando sus soluciones (por ejemplo, vector) para una futura utilización.
Backtracking Recursivo: El backtracking es un enfoque recursivo que explora todas las opciones posibles para encontrar una solución a un problema.

2. Sistemas operativos

2.1 Definiciones iniciales

Un Sistema Operativo: Es un programa o conjunto de programas de un sistema informático que gestiona los recursos de hardware y provee servicios a los programas de aplicación de software.
Proceso: Es un programa en ejecución que abarca un espacio en la memoria y que posee la capacidad para leer, escribir datos.
Direcciones de memoria: Es el espacio físico (y virtual) que puede ser compartido por los distintos programas.
Archivos: Son espacios de almacenamiento en memoria secundaria que son manejados por el SO a través de syscalls y que están organizados en forma de árbol.
Input/Output - I/O: Se refiere a dispositivos que le ingresan datos al SO y que producen salidas.
Kernel: Es el Software que dirige el funcionamiento fundamental del SO.
Shell: Es la interfaz de comandos de UNIX y aunque no forma parte del SO*, utiliza syscalls.

2.2 *Procesos* / programa en ejecucion / RAM y como se gestionan.

Un proceso es una actividad que se caracteriza por la ejecución de una secuencia de instrucciones, un estado y recursos asociados.

El PCB(Bloque de Control de Procesos) es un registro especial donde el sistema operativo agrupa toda la información que necesita conocer respecto a un proceso particular. /estructura de datos.

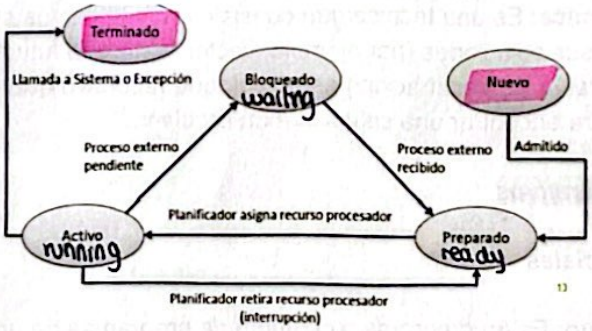
Cada vez que un nuevo proceso aparece se crea un PCB. Luego:

- El proceso es reconocido para el sistema operativo.
- Se hace elegible para competir por los recursos del sistema dado que el PCB está activo y asociado a dicho proceso.
- Cuando el proceso termina, el PCB es eliminado para dejar espacio libre en el registro y poder almacenar otro PCB.

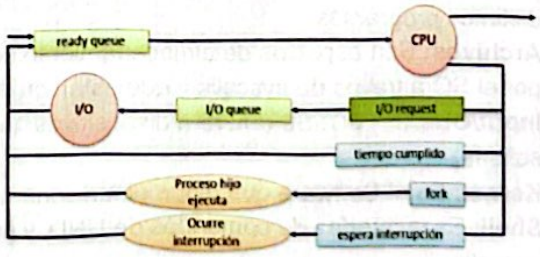
guarda el estado del proceso y el numero del proceso, y el program counter

Estados de un proceso:

- **Nuevo o en Espera:** Antes de tener su primer estado "preparado" los procesos pasan por un estado de espera. Desde que son creados hasta que son ejecutados. /new
- **Activo:** El proceso está utilizando la CPU (se encuentra en ejecución). /running.
- **Preparado:** Esto se refiere a que el proceso no está en ejecución, pero podría pasar a activo /ready
- **Bloqueado:** Este estado se da cuando un proceso está a la espera de un evento /waiting
- **Terminado:** Esto ocurre cuando un proceso realiza una llamada a sistema (o excepción) solicitando su propio término y posterior eliminación. /terminated



Los procesos se vinculan en forma de un árbol donde cada proceso tiene un padre y puede tener 0 o más procesos hijos.
La dinámica de ejecución de los procesos en CPU sigue típicamente la lógica descrita por el diagrama:



El sistema operativo posee dos modos fundamentales para su operación:

- Modo Kernel (privilegiado): En este modo se pueden realizar todas las operaciones.
- Modo Usuario (no privilegiado): Genera una excepción cuando se intenta realizar una instrucción privilegiada

Cambio de modo: El sistema comienza en Modo Kernel y cambia a Modo usuario cuando cede el control (al usuario) y recupera el solo cuando ocurre una llamada a sistema, interrupción o excepción.

El sistema operativo posee distintos elementos para controlar los procesos: Interrupciones de HW y de SW (syscalls y excepciones), Scheduling y Sincronización (hablaremos de Threads en esta parte)

→ Interrupciones HW: generadas por dispositivos I/O.
→ " SW: sistemas, etc.

2.3 Interrupciones

vector de interrupciones controla estas

Interrupción: Es una señal recibida por el procesador de una computadora, para indicar que debe "interrumpir" el curso de ejecución actual y pasar a ejecutar código específico para tratar esta situación. → cambio de contexto / dado por lo que se ejecuta / multitasking.

Excepciones: Son una interrupción sincrónica dada por un error en la ejecución de un programa.

Syscalls: Ocurre cuando un programa en ejecución llega a una instrucción que requiere del sistema operativo para alguna tarea. (método utilizado por programas para comunicarse con el Kernel).

- **Open:** Se utiliza cuando un programa inicializa el acceso a un archivo.
- **Read:** Se utiliza cuando un programa requiere acceder a los datos de un archivo almacenado en el sistema de archivos.
- **Write:** Se utiliza cuando se desea escribir información desde un buffer declarado por el usuario a un componente.
- **Close:** Se utiliza cuando un programa finaliza el acceso a un archivo.
- **Fork:** Es una llamada que permite crear una copia (hijo) de un proceso determinado (padre). Ambos tendrán distinto identificador de proceso (PID) y el padre tendrá "conocimiento" del PID del hijo. / copia padre.
- **Exec:** Permite poner en ejecución un proceso.
- **Wait:** Es una llamada que permite que un proceso espere a que se ejecute otro proceso.
- **Exit:** Termina la ejecución de un proceso en el contexto ideal.
- **Kill:** Permite terminar con la ejecución de un proceso independiente de su punto de ejecución.
- sleep: dormir el proceso hasta que se necesite.

2.4 Scheduling → mucho scheduling + cambio contexto = thrasing

Es el conjunto de políticas y mecanismos construidos dentro del sistema operativo que gobiernan la forma de ejecutar los procesos.

TIPOS:

- Scheduling expropiativo: no necesariamente termina el proceso para pasar al otro.
- Scheduling no-expropiativo: hasta que termina el proceso pasa al siguiente.
- Batch scheduling
- Interactive scheduling
- Real-time scheduling

→ el primero que llega se atiende hasta que se termine

FIRST COME FIRST SERVED /NO EXPROPIATIVO

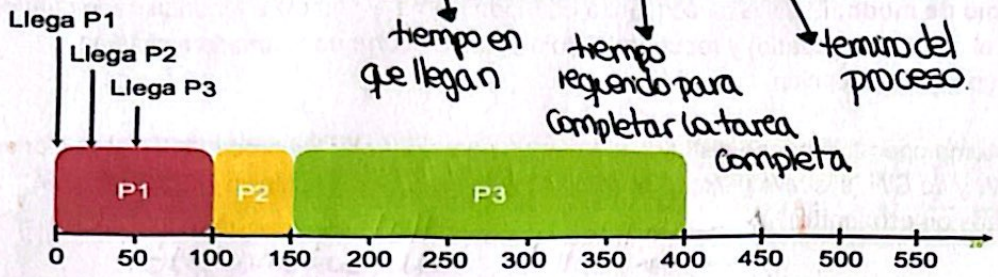
[end time - arrival time]

Process	Arrival time	CPU burst	End time	Turnaround
P1	0	100	100	100
P2	20	50	150	130
P3	50	250	400	350

tiempo en que llegan

tiempo requerido para completar la tarea completa

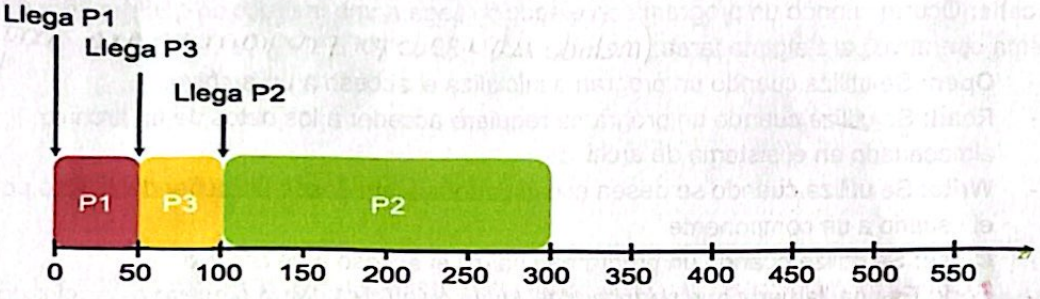
termino del proceso



SHORTEST JOB FIRST /NO EXPROPIATIVO

→ se ejecuta el proceso con menor CPU burst

Process	Arrival time	CPU burst	End time	Turnaround
P1	0	50	50	50
P2	100	200	300	200
P3	50	50	100	50



ROUND ROBIN: Cada proceso recibe un quantum de tiempo en la CPU, debiendo abandonarla al término del mismo. **EXPROPIATIVO.**

→ si es expropiativo se comporta como FCFS

→ se ejecutan a medida que llegan.

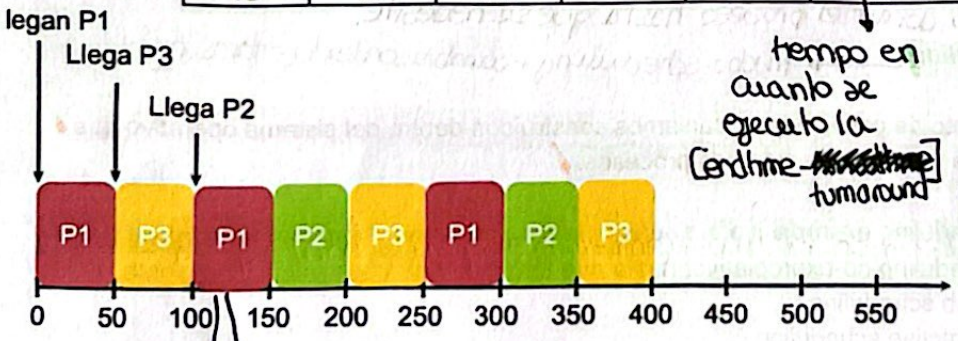
Ejemplo: q = 50

→ tiempo maximo ejecutando el proceso, si es que hay procesos.

Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time
P1	0	150	300	300	0	0
P2	100	100	350	250	100	0
P3	50	150	400	350	50	0

tiempo en cuanto se ejecuto la [end time - turnaround]

tiempo que se demoro en ejecutarse. [execution - Arrival]



se pone el P1, ya que P2 llego recien en 100, luego va P2.

P1 → P2 → P3

llega a la cola de P1 y P3, por ende, P2 va luego de P1 nuevo.

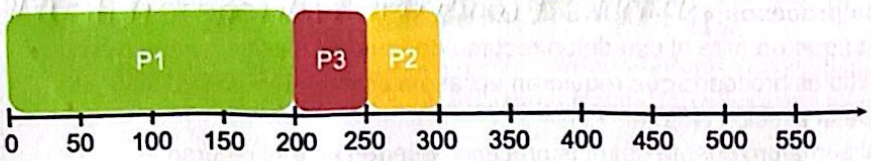
PRIORITY SCHEDULING: Proceso con la prioridad más alta se va atendiendo primero, a medida que vayan llegando. EXPROPIATIVO

mayor prioridad empieza.

Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time	Priority
P1	0	200	200	200	0	0	63
P2	0.1	50	300	299.9	250	249.9	24
P3	0.2	50	250	249.8	200	199.8	50

se atiende el proceso con mas prioridad

¿Es correcto este ejemplo?

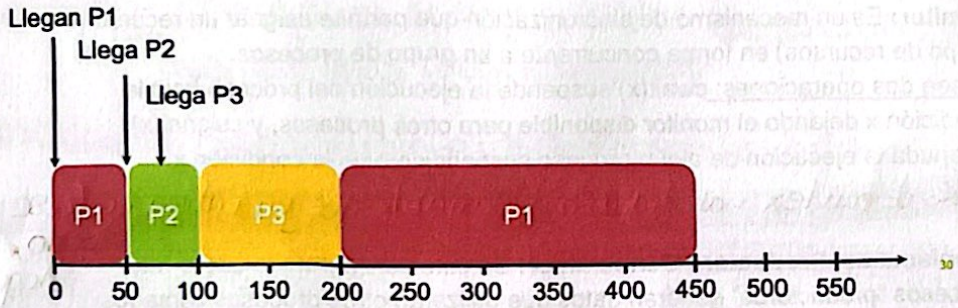


EARLIEST DEADLINE FIRST: Prioriza los procesos con deadline menor para ejecutarse primero viendo a medida que van llegando. EXPROPIATIVO.

prioriza con un deadline menor.

Process	Arrival time	CPU burst	End time	Turnaround	Execution start	Response time	Deadline
P1	0	300	450	450	0	0	500
P2	50	50	100	50	50	0	350
P3	75	100	200	125	100	25	450

menor deadline sigue procesando.



OBTENCIÓN DE DATOS TABLA.
End Time: tiempo donde termina cada proceso según la recta realizada.
Turnaround: End Time - Arrival Time
Execution Start: End Time - Turnaround / tiempo donde empiezan según la recta realizada.
Response Time: Execution Start - Arrival Time

2.5 Threads

Un Thread es la unidad mínima que se puede planificar en un SO y se refiere a un contexto individual único de ejecución (una tarea que puede ser ejecutada al mismo tiempo con otra tarea).

create(): crea un thread
exit(): salir / termina un thread
join(): espera que un thread salga } funciones en el main.
yield: cambio de contexto, se da el paso a otro thread

TIPOS:

Threads a nivel de usuario (ULT): El Sistema operativo maneja procesos por lo que no tiene "conocimiento" de la existencia de Threads. El manejo de ellos se realiza mediante una Biblioteca a nivel de usuario.

Threads a nivel de kernel (KLT): El Sistema operativo maneja los procesos como un todo por lo que mantiene información de cada proceso y sus Threads.

Esquema combinado: Busca aprovechar las ventajas de ULT y KLT lo que implica que la aplicación es escrita al nivel del paralelismo lógico más conveniente y ejecutada a nivel de paralelismo físico más conveniente.

Sincronización de procesos • variables de condición: condiciones arbitrarias.

- **Semáforos:** que permite el uso de un recurso de forma exclusiva, en caso de que existan múltiples procesos que requieran acceso a éste (ofrece una solución al problema de la sección crítica).
  El valor del semáforo cuenta cuántos procesos pueden usar el recurso.
  Posee una operación de inicialización init(): Inicia el semáforo con el número máximo de procesos que tiene derecho a acceder al recurso. Si este semáforo se inicia con 1, se convierte en un semáforo binario.
  Posee dos operaciones para su funcionamiento: **wait()** valor del semáforo es > 0, indica que pueden entrar más procesos al recurso, si es = 0 el proceso queda esperando hasta que es despertado por otro proceso. **signal()** cuando el proceso deja de utilizar el semáforo se incrementa en 1 mediante el comando signal, ósea puede entrar otro proceso.
- **Monitor:** Es un mecanismo de sincronización que permite asignar un recurso (o grupo de recursos) en forma concurrente a un grupo de procesos.
  Posee dos operaciones: cwait(x) suspende la ejecución del proceso bajo la condición x dejando el monitor disponible para otros procesos, y csignal(x) Reanuda la ejecución de algún proceso suspendido bajo la condición x.

• Locks de mutex: abstracción de variable que garantiza si la sc es utilizada o no.
Acquire: entrar a sc

Fabulas

- **Problema del Productor - Consumidor:** Se caracteriza porque uno o más procesos "productores" generan datos que utilizarán otros procesos llamados "consumidores".

→ ausencia de sincronización predomina.

Se usa semaforo para P y C, un mutex para el buffer.

Para resolver este problema debemos tener un mecanismo de comunicación que permita a los productores y consumidores intercambiar información y que ambos procesos deben sincronizar su acceso al mecanismo de comunicación para que la interacción entre ellos no sea problemática, es decir, cuando mecanismo de comunicación se llene, productor se debe bloquear y cuando este vacío se bloquea al consumidor. → consumidor saca si N>0.
→ se produce y luego se consume; productor no produce mas de N elementos.

- **El problema de Lectura - Escritura:** En este problema existe un determinado objeto que va a ser utilizado y compartido por una serie de procesos concurrentes.

Conceptos SC:
- exclusión mutua: a lo mas un thread/proceso esta en la SC
- progreso: al menos un thread/proceso puede entrar a la SC.
- ausencia de inanición: si un thread/proceso quiere entrar a su SC debe hacerlo después de un tiempo acotado.

AC

Sincronización:

→ el objetivo es sacarle el mayor provecho posible a la concurrencia, pero sin afectar la lógica de los programas.

→ **race conditions:** representan una dependencia de las salidas de una operación en base al orden temporal de ejecución entre los distintos procesos. de las instrucciones internas sin control por parte del programador.

→ **sección crítica:**

- región de memoria de acceso compartido sobre lo que trabaja código.
- requiere únicamente un thread este trabajando sobre dicha región al mismo tiempo.
- conceptos:

1. exclusión mutua: uno o mas threads trabaja solo un thread trabajando con variable y luego el otro.

2. Progreso: todo thread pasa por un tiempo acotado.

3. Ausencia de inanición: todos deben pasar por la sección crítica.

→ **riesgo de colgarse:** la CPU queda asignada a un proceso actual sin poder cambiarse.

→ **busy waiting:** la CPU evalua si las variables de sincronización, pero sin avanzar en el procesamiento.

→ **atomicidad:** no se asegura que los cambios en los valores que sincronizan trabajen atomicamente.

- **HERRAMIENTAS PARA ASEGURAR SC:**

1. **Spinlock:** pueden entrar 2 a la sección crítica.
2. **Peterson:** algoritmos con 2 procesos.
3. **Locks de mutex:** garantiza exclusión mutua, se abstrae una variable y se garantiza la atomicidad.
4. **Semaforos:**
   → permite que accedan N>0 pero limitados threads a la SC.
   → P(), wait(): decrementa el valor del contador, entro thread a la SC.
   → V(), signal(): incrementa " , salio thread de la SC.
5. **Variables condición:**
   → limita quien entra primero a la SC.
   → mecanismos con condición, no con limitación de threads.

- spinlock()

```c
/*
  Compilar:
    gcc spinlock.c -lpthread
*/

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int max;
volatile int counter = 0; // Variable compartida global
int lock = 0; //0: SC está libre, 1: SC esta ocupada

void *mythread(void *arg)
{
    while(lock);
    lock = 1;
    char *letter = arg;
    int i; // stack (privado por thread)
    printf("%s: inicio [direccion de i: %p]\n", letter, &i);
    for (i = 0; i < max; i++)
    {
                counter = counter + 1; // compartido: solo uno
    }
    printf("%s: terminado\n", letter);
    lock = 0;
    return NULL;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
                fprintf(stderr, "uso: ./spinlock <n_ciclos>\n");
                exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: inicio [contador = %d] [%lx]\n", counter, (unsigned long) &counter);
    pthread_create(&p1, NULL, mythread, "A");
    pthread_create(&p2, NULL, mythread, "B");
    // join espera a que los threads terminen
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: terminado\n [contador: %d]\n [esperado: %d]\n", counter, max*2);
    return 0;
}
```

para porque lock vale 0.

→ cae en un ciclo.

→ termina y luego entra el otro proceso.

PROBLEMA A ESTO: - pueden entrar los dos a la SC. sino se pone el candado antes entre un proceso y otro.
- desperdicia CPU. Busy waiting.

• TURN() ⟶ PETERSON ⟶ ES PARA 2 PROCESOS

```c
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>

int max;
volatile int counter = 0; // Variable compartida global
int turn = 1; // A que thread le toca entrar en la SC
int flag[2] = {0,0}; // Arreglo que indica si algun thread quiere entrar a la SC ⟶ arreglo, de procesos o thread entra a SC

void *mythread1(void *arg)
{
        int me = 0;
        int other = 1 - me; } identificadores.
    char *letter = arg;
    int i; // stack (privado por thread)
    flag[me] = 1; // Quiero entrar a la SC
    turn = other;
    while(flag[other] && turn == other); // Espero si no me toca
    printf("%s: inicio [direccion de i: %p]\n", letter, &i);
    for (i = 0; i < max; i++)
    {
                counter = counter + 1; // compartido: solo uno
    }
    printf("%s: terminado\n", letter);
    flag[me] = 0; // Ya no necesito ni quiero entrar a la SC
    return NULL;
}

void *mythread2(void *arg)
{
        int me = 1;
        int other = 1 - me; } identificadores
    char *letter = arg;
    int i; // stack (privado por thread)
    flag[me] = 1; // Quiero entrar a la SC
    turn = other;
    while(flag[other] && turn == other); // Espero si no me toca
    printf("%s: inicio [direccion de i: %p]\n", letter, &i);
    for (i = 0; i < max; i++)
    {
                counter = counter + 1; // compartido: solo uno
    }
    printf("%s: terminado\n", letter);
    flag[me] = 0; // Ya no necesito ni quiero entrar a la SC
    return NULL;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
                fprintf(stderr, "uso: ./turn <n_ciclos>\n");
                exit(1);
    }
    max = atoi(argv[1]);

    pthread_t p1, p2;
    printf("main: inicio [contador = %d] [%lx]\n", counter, (unsigned long) &counter);
    pthread_create(&p1, NULL, mythread1, "A");
    pthread_create(&p2, NULL, mythread2, "B");
    // join esperan a que los threads terminen
    pthread_join(p1, NULL);
    pthread_join(p2, NULL);
    printf("main: terminado\n [contador: %d]\n [esperado: %d]\n", counter, max*2);
    return 0;
}
```

thread1 thread2

(garantiza que uno pase, no dejando a ambos en espera) (siempre pasa 1 ya que turn es global)

TAREA: Si se cumple el algoritmo de exclusividad mutua en PETERSON, con una CPU y un pr

(se garantiza exclusion mutua)

LOCKS DE MUTEX: (no genera busy waiting)

se abstrae a una variable que garantiza la atomicidad de su manejo y exclusion mutua de acceso a la SC.

```
pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* funcionThread(void *arg)
{
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Inicia job %d \n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Termina job %d \n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}
```

seccion critica

MUTEX MALO:

```
/*
  Compilar:
    gcc -o mutex_mal mutex_mal.c -lpthread
*/

#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;

void* funcionThread(void *arg)
{
    unsigned long i = 0;
    counter += 1;
    printf("\n Inicia job %d \n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Termina job %d \n", counter);
    return NULL;
}

int main(void)
{
    int i = 0;
    int err;
    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &funcionThread, NULL);
        if (err != 0)
            printf("\nNo se puede crear el thread :[%s]", strerror(err));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);

    return 0;
}
```

MUTEX BUENO:

```
/*
  Compilar:
    gcc -o mutex_bien mutex_bien.c -lpthread
*/

#include<stdio.h>
#include<string.h>
#include<pthread.h>
#include<stdlib.h>
#include<unistd.h>

pthread_t tid[2];
int counter;
pthread_mutex_t lock;

void* funcionThread(void *arg)
{
    pthread_mutex_lock(&lock);
    unsigned long i = 0;
    counter += 1;
    printf("\n Inicia job %d \n", counter);
    for(i=0; i<(0xFFFFFFFF);i++);
    printf("\n Termina job %d \n", counter);
    pthread_mutex_unlock(&lock);
    return NULL;
}

int main(void)
{
    int i = 0;
    int err;
    if (pthread_mutex_init(&lock, NULL) != 0)
    {
        printf("\n Falla inicializacion de mutex \n");
        return 1;
    }
    while(i < 2)
    {
        err = pthread_create(&(tid[i]), NULL, &funcionThread, NULL);
        if (err != 0)
            printf("\nNo se puede crear el thread :[%s]", strerror(err));
        i++;
    }
    pthread_join(tid[0], NULL);
    pthread_join(tid[1], NULL);
    pthread_mutex_destroy(&lock);

    return 0;
}
```

→ (no genera busy waiting)
→ (se ejecuta un thread a la vez)

SEMAFOROS → (define cuantos threads entran a la SC).

```c
sem_t mutex;

void* thread(void* arg)
{
    //wait
    sem_wait(&mutex);
    printf("\nEntro a la SC\n");

    //critical section
    printf("\n Como soy tan bacan... me voy a dormir durante la SC... \n");
    sleep(4);
    printf("\n Buenos dias!!! \n");

    //signal
    printf("\nSalgo de la SC\n");
    sem_post(&mutex);
}

int main()
{
    sem_init(&mutex, 0, 1);
    pthread_t t1,t2;
    pthread_create(&t1,NULL,thread,NULL);
    sleep(2);
    pthread_create(&t2,NULL,thread,NULL);
    pthread_join(t1,NULL);
    pthread_join(t2,NULL);
    sem_destroy(&mutex);
    return 0;
}
```

→ (si el semaforo llega a 0 no pueden entrar mas procesos)

se comporta como mutex, si la variable es 1

→ siempre sera 0

→ N: cantidad de procesos que pueden entrar.

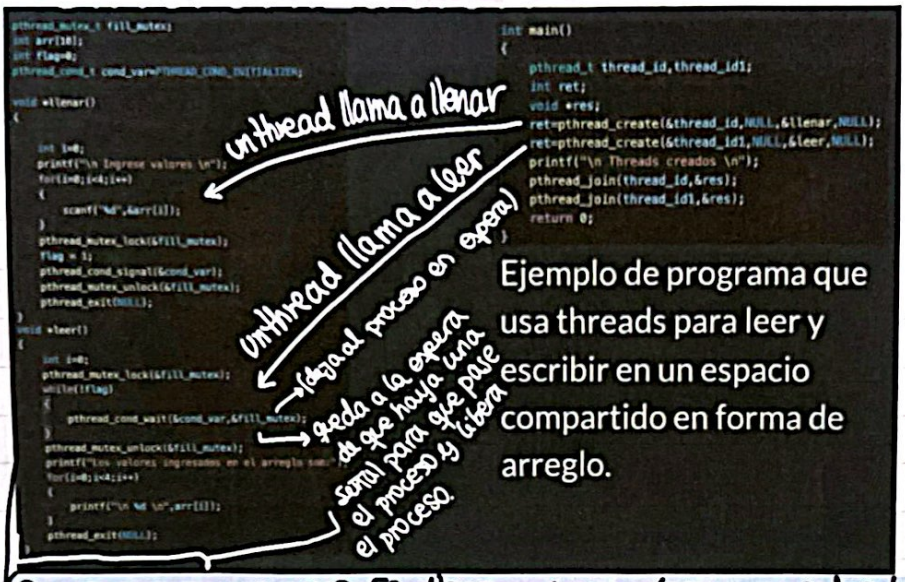
→ inicializar el semaforo

- Son mecanismos de sincronización inventados por Dijkstra.
- En vez de tratarse de un lock que restringe el acceso de un unico thread a una SC, permite que accedan N (>0, N limitado) threads a la SC.
- Para esto se implementan 2 funciones.

* P(), wait() : intenta decrementar el valor del contador para simbolizar que ha entrado un thread a la SC.

* V(), signal() : intenta incrementar el valor del contador representa una ubicacion libre mas para que otro thread pueda entrar a la SC

VARIABLES DE CONDICION:



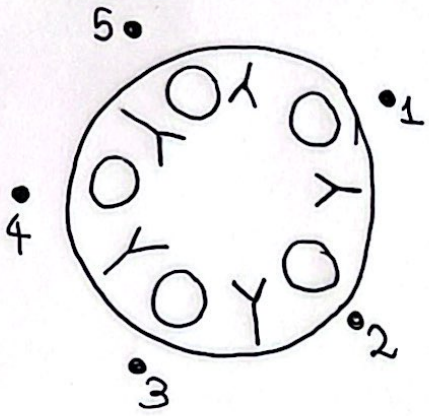
(Quiero que primero se llene y luego lea y muestre las variables)

Si se ejecuta leer primero: se estanca hasta que se le mande una señal para poder pasar.
(pthread_con_wait(&cond_var, &fill_mutex);

- Son mecanismos de sincronizacion en que la condicion no tiene de limitar la cantidad de threads que acceden a la SC, sino que se trata de condiciones arbitarias.
  * Trabaja con dos operaciones:
    - wait() que siempre bloquea al thread
    - signal() que despierta a un thread bloqueado en caso que lo haya.

- Fabula de los alumnos gladiadores.

→ Filosofos comensales.



- todos comen en un periodo de tiempo.
- si todos toman un tenedor a la vez se produce un deadlock, ya que todos quedan a la espera de que se desocupe uno (se come con uno en cada mano).
- tomar en un SC, ambos tenedores a la vez reviso a los vecinos, si estan comiendo ambos tenedores ocupados, sino estan desocupados.
- variable global, arreglo de 0 y 1 para controlar quienes comieron y quienes no.

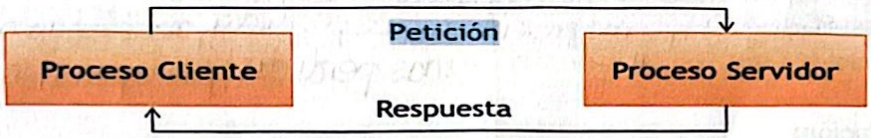
→ solo se accede como lector si el escritor deja de escribir, y si los lectores dejan de leer accede el escritor.

→ variable condición:
- si hay lectores, no hay escritores y viceversa en el recurso.
- semaforo limita la cant de lectores en el recurso, mutex para los escritores en el recurso.

Los procesos lectores entran al recurso solo para leer, procesos escritores entran al proceso para escribir.

Para resolver este tipo de problema existe una serie de restricciones, en donde solo se permite que un escritor tenga acceso al objeto al en un momento del tiempo, mientras el escritor esté accediendo al objeto, ningún otro lector ni escritor podrá acceder a él. Múltiples lectores pueden utilizarlo ya que solo leen.

- **El problema del modelo Cliente - Servidor:** En el modelo cliente-servidor, los procesos llamados servidores ofrecen una serie de servicios a otros procesos que se denominan clientes.



Es necesario que el sistema operativo ofrezca servicios que permitan la comunicación entre los procesos cliente y servidor, Cuando los procesos se ejecutan en la misma máquina, se pueden emplear técnicas basadas en memoria compartida o archivos, mientras que si se ejecutan en equipos diferentes (no comparten memoria), se usan técnicas basadas en paso de mensajes.

2.6 Administración de memoria

La **memoria** es, en concreto, el grupo de circuitos que permiten almacenar y recuperar la información.
Random Access Memory (RAM) es la memoria utilizada por un procesador para recibir instrucciones y guardar los resultados.
Espacio de direcciones: rango de direcciones discretas, donde cada una puede corresponder a un registro de memoria físico o virtual, un dispositivo periférico, un host de red, un sector de disco u otra entidad lógica o física.
La MMU (Memory Management Unit): es el hardware necesario para producir la traducción de memoria lógica a memoria física.Este proceso, llamado paginación, contempla:

- El espacio de direcciones se divide en unidades llamadas páginas.
- Las unidades en la memoria física que les corresponden a las páginas se denominan marcos.
- Se utiliza una tabla de páginas para mantener la relación entre las direcciones virtuales y la memoria física.
- Las unidades de las páginas y de los marcos son siempre del mismo tamaño.

fragmentación: producida debido a la inserciones en memoria y que los procesos van abandonando cuando se termina generando divisiones en la memoria en bloques con tamaños insuficientes para agregar nuevos procesos
→ **interna**: fragmentación dentro de la memoria
→ **externa**: fragmentación entre bloques de la memoria.

Swapping

El swapping es el procedimiento de mover temporalmente un proceso (o parte de él) desde la memoria principal a un dispositivo secundario de almacenamiento (Disco Duro) para luego devolverlo a la memoria principal.

- **Swap-in:** Retornamos desde la memoria secundaria a la principal. HDD a RAM.
- **Swap-out:** Movemos desde la memoria principal a la memoria secundaria RAM a HDD.

Tiene estrategias de swapping

- **Desfragmentación (compactación):** proceso conveniente mediante el cual se acomodan los archivos en un disco para que no se aprecian fragmentos de cada uno de ellos, de tal manera que quede contiguo el archivo y sin espacios dentro del mismo. Existe interna pérdida de espacio en disco debido a que el tamaño de un determinado archivo sea inferior al tamaño del cluster o externa que es entre bloques.
- First-fit: primer espacio donde pueda colocar el proceso
- Best-fit elijo el que me presente menos perdida, mejor espacio posible
- Worst-fit " mas perdida, peor espacio posible.

ayudan a ubicar los procesos en la RAM.

Segmentación

La segmentación es la división de la memoria asignada a un proceso en segmentos más pequeños y persigue que la asignación a espacios en memoria sea más fácil. La MMU (Memory Management Unit) se encarga de llevar registro de esa asignación almacena la Segment Table (en que cada segmento tiene base y limit).
Segmentation fault: intento fallido de acceso a la información a los que no se tiene autorización.

soluciones para mejorar la administración de la memoria

Paginación

La paginación es una estrategia de organización de la memoria física que consiste en dividir la memoria en porciones de igual tamaño. A dichas porciones se las conoce como páginas físicas o marcos los que están identificados por un número.

PUNTOS IMPORTANTES

- Si dejamos segmentos como espacios del mismo tamaño obtenemos páginas de memoria
- Las páginas se ubicará en el espacio "virtual"
- Sus homólogos de espacio físico se llaman frames o marcos.
- La relación entre el Marco y la Página será 1:1 (tienen el mismo tamaño)
- El registro de las Páginas se lleva, a través de la MMU, en una Tabla de Páginas.

EJEMPLO
Se tiene un espacio virtual de 64B, físico de 128B y páginas de 16B. Luego:
Direcciones posibles en una página: 16 → 4 bits (esto da el mayor offset posible en una página)
Espacio virtual de 64B → 6 bits. Dado que 4 bits determinan el offset, los 2 restantes indican el número de páginas (hay 4)

→ transformar de lógica a física (RAM)

Espacio físico de 128B → 7 bits. Al igual que para el caso anterior, el offset está determinado por 4 bits, por lo que hay 3 bits para los marcos (son 8).

Tomemos por ejemplo la dirección virtual 21 (010101):
- ✓ Página 1 (01)
- ✓ Offset: 5 (0101)
- ✓ Según la page table, página 1 (01) corresponde a frame 7 (111)

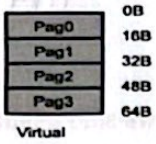
Luego la dirección física es 117 (1110101)

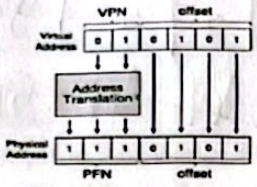
página
offset
pag →

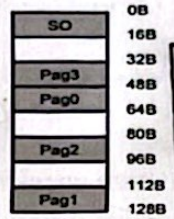
Page	Frame
00 (0)	011 (3)
01 (1)	111 (7)
10 (2)	101 (5)
11 (3)	010 (2)

Page Table

offset: 4bits
3bits marco







físico = 128B = 2^7
virtual = 64B = 2^6
páginas = 16B = 2^4 = 4bits páginas
7 − 4 = 3
6 − 4 = 2
offset: 4bits
2bits páginas

¿Cómo se gestiona un Page Fault? → minimizar pagefaults.

Se produce una interrupción, se busca la página en almacenamiento secundario (disco), se busca un marco libre, se lee la página de almacenamiento secundario y se copia en el marco seleccionado, se actualiza la tabla de páginas y se reiniciamos en la instrucción interrumpida

Algoritmos para evitar las PageFault, ya que estas requieren un acceso a memoria secundaria, por ende, son malas.

→ comparar con el mejor / siempre MIN.

ALGORITMO MIN: al cambiar una página, se debe reemplazar la página que no se va a usar durante más tiempo. [Conoce el futuro] / La máxima cantidad la da el algoritmo MIN.

Ejemplo: Supongamos que poseemos la siguiente secuencia de solicitudes y 7 0 1 2 0 3 0 4 2 3 0 3 2 que contamos con 4 espacios de memoria. Contemos los page faults y apliquemos algoritmo: ← la más antigua saco.

- 4 page faults de inicio (7 0 1 2)
- 0 page faults del 0
- 1 page fault del 3 -> reemplaza al 7 -> (3 0 1 2)
- 0 page faults del 0
- 1 page fault del 4 -> reemplaza al 1 -> (3 0 4 2)
- 0 page faults hasta el final

Total: 6 page faults

7	3	3										
0	0	0										
1	1	4										
2	2	2										
4pf			1pf			1pf						= 6pf

ALGORITMO FIFO: Se reemplaza la página que lleva más tiempo en memoria → [página con mas tiempo en memoria]

Ejemplo: Supongamos que poseemos la siguiente secuencia de solicitudes 0 1 2 0 1 3 0 3 1 2 1 y que contamos con 3 espacios de memoria. Contemos los page faults y apliquemos algoritmo: ← orden de ingreso

- 3 page faults de inicio (0 1 2)
- 0 page faults del 0 y del 1
- 1 page fault del 3-> reemplaza al 0
- 1 page fault del 0 -> reemplaza al 1
- 0 page faults del 3
- 1 page fault del 1 -> reemplaza al 2
- 1 page fault del 2 -> reemplaza al 3
- 0 page faults del 1

Total: 7 page faults

0	3	3	3	2								
1	1	0	0	0								
2	2	2	1	1								
3pf			1pf			1pf			1pf			= 7pf.

First In First Out

→ se reemplaza la ultima pagina que se hallamado.

ALGORITMO LRU (Least Recently Used): Sustituye la página que más tiempo lleva sin ser usada.

Ejemplo: Supongamos que poseemos la siguiente secuencia de solicitudes 7 0 1 2 0 3 0 4 2 3 0 3 2 y que contamos con 4 espacios de memoria. Contemos los page faults y apliquemos algoritmo:

- 4 page faults de inicio (7 0 1 2)
- 0 page faults del 0
- 1 page fault del 3 -> reemplaza al 7
- 0 page faults del 0
- 1 page fault del 4 -> reemplaza al 1
- 0 page faults hasta el final

Total: 6 page faults

mira hacia atras y saco el mas antiguo.

7	3	3
0	0	0
1	1	4
2	2	2

saco el que menos use y que este en la memoria

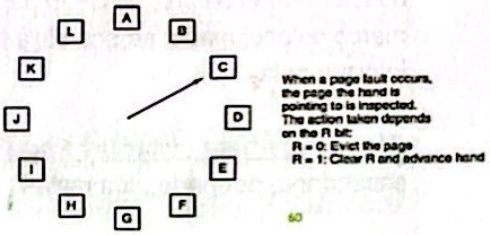
4pf 1pf 1pf = 6pf

ALGORITMO DEL RELOJ: Utiliza un reference bit (que vale 1 si la página ha sido usada)

¿Como funciona?

- El algoritmo elige alguna página arbitrariamente:
- Si el reference bit = 1, se cambia a 0 y se pasa a la siguiente página (según el reloj).
- Si el reference bit = 0, se elige esa página.

NO SE PREGUNTA!!



2.7 Sistema de archivos

Un **archivo** es identificado por un nombre y la descripción de la carpeta o directorio que lo contiene. La organización de los archivos es en directorios jerárquicos con "forma de árbol" y típicamente separados por un /.

RAID: esquemas de almacenamiento que buscan mantener persistencia y seguridad sobre los datos guardados en HDD/SSD. Si los datos se mantienen dentro del RAID, entonces se podrían recuperar los datos después de fallas físicas.

TIPOS DE RAID:

- **RAID 0:** distribuye los datos entre dos o más discos sin información de paridad que proporcione redundancia. Tiene como ventaja su fácil implementación y como desventaja su nula tolerancia a fallos.
- **RAID 1:** consiste en crear una copia exacta (o espejo) de un conjunto de datos en dos o más discos. Tiene como ventaja la tolerancia a fallos y como desventaja desaprovechamiento de la capacidad de los discos.
- **RAID 5:** requiere un mínimo de 3 discos y consiste en almacenar la paridad de ciertos bloques del arreglo y además se distribuye el almacenamiento de la paridad. Tiene como ventaja la lectura veloz y como desventaja la escritura es más lenta.

Asignación de bloques: determinan cómo los bloques son asignados a un sistema de archivos.

- ***Asignación Contigua:*** los bloques de un archivo se almacenan de manera secuencial en ubicaciones contiguas en el disco. Tiene como ventaja la simplicidad y

el acceso rápido ya que el acceso secuencial a los datos es muy rápido ya que los bloques están físicamente juntos. Y como desventaja la fragmentación externa y la dificultad para el crecimiento de archivos.

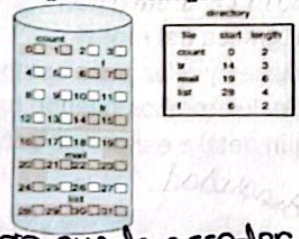
- **Asignación Enlazada**: los bloques de un archivo se almacenan en cualquier lugar del disco, y cada bloque contiene un puntero al siguiente bloque. Tiene como ventaja que no existe la fragmentación externa y facilidad para el Crecimiento de Archivos. y como desventaja acceso lento y poca fiabilidad.
- **Asignación Indexada:** En esta asignación se utiliza un bloque de índice que contiene punteros a todos los bloques de datos del archivo. Tiene como ventaja el acceso directo y su escalabilidad y como desventaja su complejidad.

  TIPOS:
  - **Esquema Enlazado:** cada bloque de índice (index block) contiene una serie de punteros, y el último puntero de cada bloque de índice apunta a otro bloque de índice. Utilizado para manejar sistemas de archivos de gran tamaño donde la cantidad de bloques es desconocida.
  - **Multi-level:** Se organiza en múltiples niveles de indirección, donde cada nivel tiene un conjunto de punteros que apuntan a los siguientes niveles de la jerarquía, hasta llegar a los bloques de datos finales. Es eficiente para sistemas de archivos de gran tamaño y distribuidos
  - **Esquema Combinado:** Combina características de los esquemas enlazados y multinivel. Utiliza una estructura enlazada para los bloques de índice superiores y luego pasa a una estructura multinivel para acceder a los bloques de datos finales. Proporcionar una combinación de eficiencia y flexibilidad en la gestión de sistemas de archivos de gran tamaño.

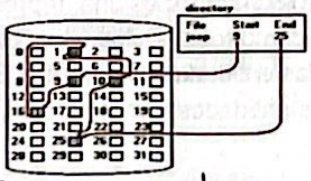
Comparación de los métodos

Método	Ventajas	Desventajas
Contigua	Acceso rápido, simplicidad	Fragmentación externa, dificultad para crecer
Enlazada	Sin fragmentación externa, crecimiento fácil	Acceso lento, fiabilidad
Indexada	Acceso directo, escalabilidad	Overhead del índice, complejidad

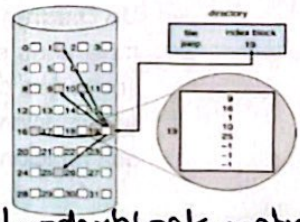
Asignación contigua



Asignación enlazada



Asignación indexada



- acceso aleatorio, puede acceder cualquier bloque, ya sea el inicio y fin.

- almacena punteros, toman espacio de almacenamiento
- acceso secuencial.

- index block, contiene los bloques del archivo
- acceso aleatorio.

2.7 Virtualización

La virtualización permite que múltiples sistemas operativos (OS) y aplicaciones se ejecuten en el mismo hardware físico de manera aislada y eficiente. Para lograr esto, las tecnologías de virtualización deben gestionar cuidadosamente los recursos del sistema, como la CPU, la memoria y el almacenamiento.

- **Scheduling (Planificación de Tareas):** En un entorno virtualizado, el hipervisor o el gestor de virtualización debe manejar la planificación no solo para los procesos dentro de cada máquina virtual (VM), sino también entre las propias VMs.
- **Gestión de memoria (RAM):** En un entorno virtualizado es más compleja, ya que el hipervisor debe gestionar la memoria física entre múltiples VMs y sus propios sistemas operativos invitados.
- **File Systems (Sistemas de Archivos):** En un entorno virtualizado, cada VM tiene su propio sistema de archivos, que puede ser gestionado de manera diferente por el hipervisor.

3. Bases de datos

metodología: diseño conceptual (modelo E-R)
" lógico (modelo relacional)
" físico (aca interviene SQL).

3.1 Definiciones Iniciales

- **Atomicidad:** Una transacción se ejecuta en su totalidad o no se ejecuta.
- **Consistencia:** Las transacciones deben llevar la base de datos de un estado consistente a otro estado consistente, preservando las reglas de integridad y las restricciones definidas en la base de datos.
- **Aislamiento:** Cada transacción se debe ejecutar como si fuera la única transacción en el sistema.
- **Durabilidad:** Los resultados de una transacción confirmada son permanentes y no se perderán

3.2 Modelamiento de la base de datos

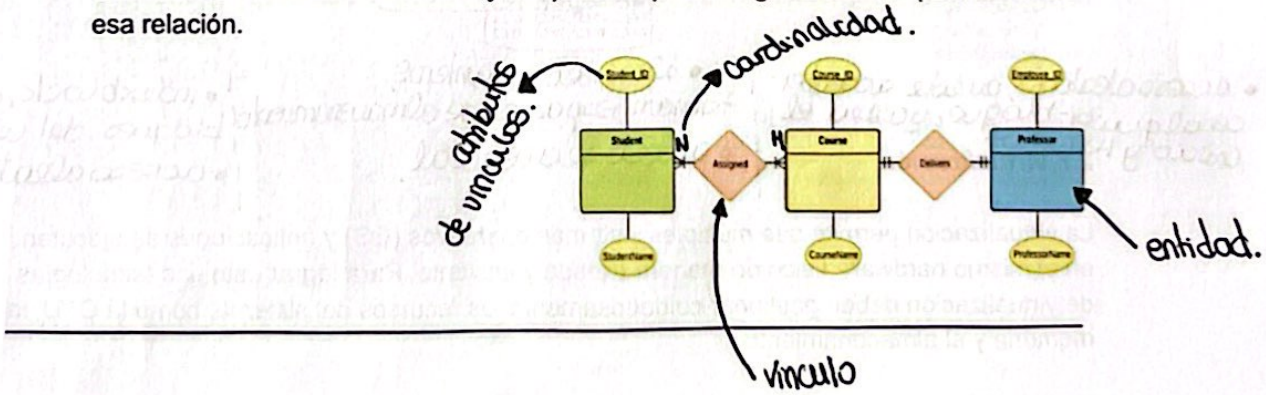
Modelo Entidad-Relación: Se utiliza para representar la estructura y las relaciones entre diferentes entidades de un sistema.

1º - **Entidad:** es un objeto o concepto del mundo real que puede ser identificado y sobre el cual se desea almacenar información en una base de datos.

- **Atributo:** es una propiedad o característica que describe una entidad.

2º - **Relación:** representa la asociación entre dos o más entidades. Estas relaciones se manejan por "cardinalidad" puede haber relaciones uno a uno, uno a muchos o muchos a muchos. /nombre - cardinalidad - atributos asociados a vinculos.

- **Un Diagrama Entidad-Relación:** es una representación gráfica del modelo, mostrando entidades como rectángulos, atributos como elipses y relaciones como líneas que conectan las entidades. Un modelo ER puede incluir rombos cuando hay una relación entre dos entidades y se quiere expresar algún detalle específico sobre esa relación.

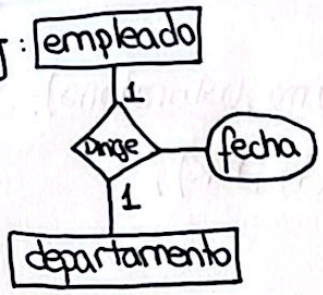


RECETARIO PARA PASAR DE E-R a modelo relacional.

1. Toda entidad transformarla en una tabla, con sus atributos. Uno o varios como clave principal.

2. Para cada vinculo 1-1, agregue los atributos de la clave primaria de una de las entidades asociadas a la otra como clave externa. La entidad que recibe clave con participacion total es la elegida, ya que todas sus ocurrencias participan en la relación.

Ej:



SQL en la prueba, en vez del diagrama

Empleado (Rut(PK), Nombre, sueldo).

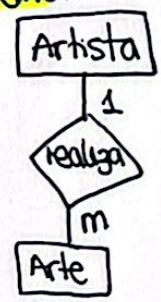
departamento (CodDep(PK), NombreDep, Rut_empleado(FK), Fecha).

* Sino hay partición completa se debe crear.

Dirige (Rut(PK,FK), CodDep(PK,FK)).

3. Para cada vinculo 1-n, agregue los atributos de la clave primaria de la entidad del lado 1 del vinculo a la relación de la entidad del lado n del vinculo como clave externa.

Ej:

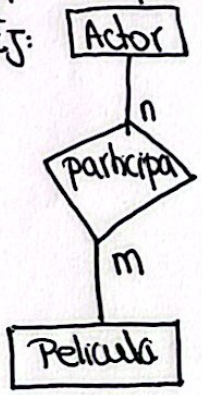


Artista (RUT(PK), Nombre, Fecha_nacimiento)

Arte (CodArte(PK), nombre, ano, RUT_Artista(FK))

4. Para cada vinculo n-m, cree una nueva relación (tabla) con el nombre del vinculo y que tenga como atributo la concatenación de las claves primarias de ambas entidades involucradas. Esa concatenacion sera la llave primaria de la relación (tabla) y cada uno de los conjuntos originales sera la clave externa hacia las relaciones (tablas) de la entidades que conforman el vinculo.

Ej:



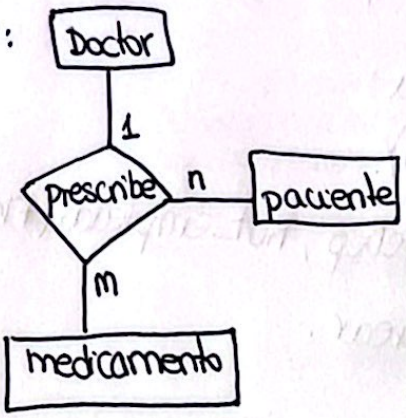
Actor (RUT(PK), Nombre, Fecha_nacimiento)

Pelicula (CodPeli(PK), nombre, ano).

Participa (RUT(PK,FK), CodPeli(PK,FK))

4. Para vinculo n-ario, cree una nueva relacion tabla con el nombre del vinculo y que tenga como atributos la concatenacion de las claves primarias de todas las entidades involucradas. Esta concatenacion sera la clave primaria de la relacion y cada una de los conjuntos originales sera la clave externa hacia las relaciones de las entidades que conforman el vinculo.

Ej:



Doctor (RUT(PK), nombre, apellido, Fecha_nacimiento)

Paciente (RUT(PK), nombre, ano)

medicamento (CodMed(PK), nombre, ano, Laboratorio)

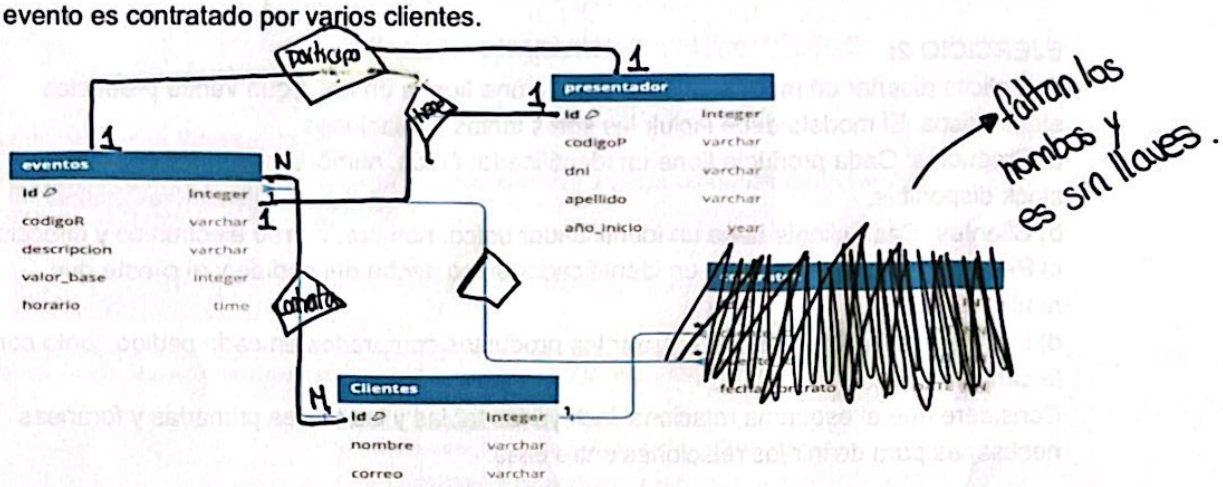
Prescribe (Rut_Doc(PK, FK), CodMed(PK, FK), Rut_paciente(PK, FK))

EJERCICIO

Una empresa dedicada a la realización de eventos requiere una DB para administrar los eventos ofrecidos y los clientes que lo contratan. De los eventos que ofrece, se conoce su código, la descripción, el valor base, el horario del evento y un solo presentador cuyos datos son el código de animador, el dni, el apellido y el año de inicio en la empresa. Además, de los presentadores sabemos que:

a) Un evento tiene solamente un presentador, y el presentador sólo participa en un tipo de eventos.
b) Sus datos son el código de presentador, el dni, el apellido y el año que empezó a ser presentador.

De los clientes se registra su número de cliente y cuando un cliente contrata un evento, lo contrata tal cual está armado. Además, un cliente puede contratar varios tipos de eventos, y un evento es contratado por varios clientes.



Modelo Relacional: Es un enfoque para organizar y estructurar datos en una base de datos donde la información se organiza en tablas bidimensionales llamadas "relaciones".

Cada relación consta de filas y columnas, y se representa como una tabla en la que cada fila corresponde a una "tupla" y cada columna a un "atributo".

Relación: se refiere a la conexión lógica entre dos tablas. Puede haber relación uno a uno, uno a muchos, muchos a uno o muchos a muchos y se establecen mediante el uso de llaves primarias y foraneas.

La clave primaria es un conjunto de uno o más atributos que identifican de manera única cada tupla en una tabla, garantiza que no existan duplicados y permite la referencia cruzada. (representa en tabla unica, osea de manera unica de la tabla; nt; id; etl).
(no puede ser nulo).

La clave foránea es un atributo en una tabla que coincide con la clave primaria en otra tabla. Establece una relación entre las dos tablas facilitando la vinculación entre ellas.

→ entidades se transforman en tablas

Restricciones de integridad

- La clave primaria no puede contener valores nulos y debe asegurar la unicidad de cada registro en la tabla.
- Los valores en una columna deben coincidir con los valores de una clave primaria en otra tabla. / clave foránea.
- Único (Unique): La restricción única garantiza que los valores en una columna o conjunto de columnas sean únicos en la tabla, permite valores nulos.
- Check (Verificación): permite especificar una condición que los valores en una columna deben cumplir.
- No Nulo (Not Null): asegura que los valores en una columna no pueden ser nulos

---

EJERCICIO 2:

Se solicita diseñar un modelo relacional para una tienda en línea que vende productos electrónicos. El modelo debe incluir las sgtes tablas y relaciones:

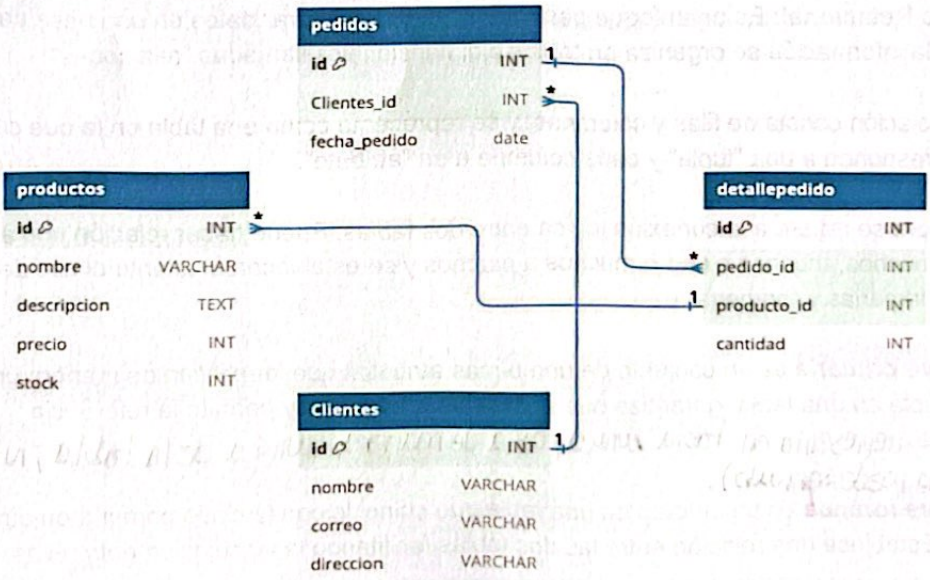
a) Productos: Cada producto tiene un identificador único, nombre, descripción, precio y stock disponible.

b) Clientes: Cada cliente tiene un identificador único, nombre, correo electrónico y dirección.

c) Pedidos: Cada pedido tiene un identificador único, fecha del pedido y el cliente que realizó el pedido.

d) Detalles de Pedidos: Debe registrar los productos comprados en cada pedido, junto con la cantidad de cada producto.

Considere que el esquema relacional incluya las tablas y las claves primarias y foráneas necesarias para definir las relaciones entre ellas.



3.3 Normalización

La normalización tiene como objetivo reducir la redundancia de datos y mejorar la integridad de la información almacenada.

Primera Forma Normal (1FN)
Requiere que cada celda en una tabla contenga un solo valor, es decir, los valores deben ser atómicos. No debe haber duplicados en las filas y las columnas deben tener nombres únicos.

Nombre	Curso	Semestre
Martín Gutiérrez	EDDA; Bases de Datos	2020-2

Nombre	Curso	Semestre
Martín Gutiérrez	EDDA	2020-2
Martín Gutiérrez	Bases de Datos	2020-2

Segunda Forma Normal (2FN)
Para cumplir con la 2FN, la tabla debe estar en 1FN y cada columna no clave debe depender completamente de la primaria.

PRESTAMO(num_socio, nombre_socio, cod_libro, fec_prest, editorial, país)

En esta relación, tenemos dos claves candidatas (num_socio, cod_libro) y (nombre_socio, cod_libro). Ahora, el atributo editorial entrega sólo información del libro, no del socio, por lo que la relación no es 2NF.

PRESTAMO1(num_socio, cod_libro, fecha_prest)
LIBRO(cod_libro, editorial, país)
SOCIO(num_socio, nombre_socio)

Tercera Forma Normal (3FN):
La tabla debe cumplir con la 2FN y ninguna columna no clave debe depender de otra columna no clave. No debe haber dependencias transitivas.

LIBRO(cod_libro, editorial, país)

De esta relación, el atributo país entrega información de la editorial que publica el libro, por lo que la relación no está en 3NF.

LIBRO1(cod_libro, editorial)
EDITORIAL(editorial, país)

Dependencia Funcional:establece reglas sobre cómo los valores de ciertos atributos dependen de otros en una relación específica.

- **Dependencia funcional simple:** un atributo (o conjunto de atributos) determina de manera única otro atributo. Se representa como A -> B, lo que significa que el valor de A determina el valor de B.

Ej:
Si se tiene una tabla con atributos (ID, Nombre, Apellido), se podría expresar la dependencia funcional
Nombre -> Apellido si se asume que no hay dos personas con el mismo nombre y apellidos.

- **Dependencia funcional compleja:** un conjunto de atributos determina otro conjunto de atributos. Se representa como AB —> CD, lo que significa que los valores de A y B juntos determinan los valores de C y D.
  Ej:
  Si se tiene una tabla con atributos (ID, Nombre, Dirección, Ciudad), se podría expresar la dependencia funcional
  (Nombre, Ciudad) -> Dirección, indicando que el nombre y la ciudad juntos determinan la dirección.

EJERCICIO NORMALIZACIÓN

Una empresa tiene una base de datos, posiblemente no normalizada, sobre su inventario y ventas de la siguiente manera:

ID	Producto Nombre	Producto Descripción	Precio	Stock	Cliente ID	Cliente Nombre	Cliente Correo	Cliente Dirección	Pedido ID	Fecha del Pedido	Cantidad
1	Laptop	Laptop Dell XPS 13	1000	50	101	Juan Pérez	juan.perez@example.com	Calle 123	1001	15-01-2023	2
2	Monitor	Monitor Samsung 24"	200	75	102	Ana López	ana.lopez@example.com	Avenida 456	1002	16-01-2023	1
1	Laptop	Laptop Dell XPS 13	1000	48	103	Carlos Ruiz	carlos.ruiz@example.com	Plaza 789	1003	17-01-2023	1
3	Teclado	Teclado Logitech K120	30	150	101	Juan Pérez	juan.perez@example.com	Calle 123	1001	15-01-2023	1

Dado lo anterior, se pide que:

a) Identificar Anomalías: Examine la tabla y mencione qué anomalías pueden ocurrir debido a su diseño posiblemente no normalizado.

b) Primera Forma Normal (1NF): Convierta la tabla a su Primera Forma Normal (1NF).

c) Segunda Forma Normal (2NF): Convierta la tabla resultante de la 1NF a la Segunda Forma Normal (2NF).

d) Tercera Forma Normal (3NF): Convierta la tabla resultante de la 2NF a la Tercera Forma Normal (3NF).

e) Explicar las Transformaciones: Para cada paso de normalización (1NF, 2NF, y 3NF), explique brevemente qué cambios fueron realizados y por qué.

a) Existen datos duplicados en cuatro columnas de la tabla, es decir pertenecen a la misma persona pero son distintos productos y cantidad, por ende, al buscar por alguno de estos datos la información podría salir redundante y no tan específica. Además la información no está representada de manera íntegra. Si se elimina un pedido específico, se pierde información del cliente o de un producto.

b) 1NF, 2NF Y 3NF

id	nombre producto	producto descripción	Precio	stock
1	Laptop	Laptop Dell XPS 13	1000	50
2	Monitor	Monitor Samsung 24'	2000	75
3	Teclado	Teclado Logitech K120	30	150

Cliente_id	nombre cliente	correo cliente	Direccion cliente
101	Juan Prez	Juan.perez@xample.com	Calle 123
102	Ana Lopez	ana.lopez@xample.com	Avenida 456
103	Carlos Ruiz	carlos.ruiz@xample.com	Plaza 789

pedido_id	fecha_pedido	Cliente_id
1001	15-01-2023	101
1002	16-01-2023	102
1003	17-01-2023	103

pedido_id	producto_id	cantidad
1001	1	2
1002	2	1
1003	3	1
1001	1	1

e) Con las tablas realizadas anteriormente se tienen las tres normalizaciones, ya que para 1NF cada celda contiene un solo valor, es decir no existen valores duplicados. Luego para 2NF se tiene el primer requisito que es 1NF y además cada columna no clave depende de una llave primaria como lo es pedido_id, cliente_id y id. Finalmente, para 3NF se tiene 2NF y ninguna columna no clave depende de otro no clave, sino que dependen de las columnas de identificadores c/u.

3.4 Álgebra Relacional y SQL

La álgebra relacional es el conjunto de operaciones matemáticas que se utilizan para manipular y consultar datos almacenados en una BBD.

- **Selección (σ):** se utiliza para recuperar un subconjunto de las filas de una tabla que cumplen con una condición.
- **Proyección:** se utiliza para recuperar un subconjunto de las columnas de una tabla.
- **Unión (∪):** combina dos relaciones que tienen la misma estructura, devolviendo una nueva relación que incluye todas las filas de ambas relaciones.
- **Intersección (∩):** devuelve una nueva relación que incluye solo las filas que son comunes en dos relaciones que tienen la misma estructura.
- **Diferencia (-):** devuelve una nueva relación que incluye las filas que están en una relación, pero no en la otra.
- **Producto Cartesiano (×):** combina todas las filas de una relación con todas las filas de otra relación, devolviendo una nueva relación con las combinaciones posibles.
- **Renombramiento (ρ):** cambiar los nombres de las relaciones y atributos.

date: "2024-03-01" "año-mes-día"

SQL

Sentencias Fundamentales /extrae información de la base de datos.

- **SELECT:** Indica a qué atributos se desea consultar.
- **FROM:** especifica de qué tablas se extraen los atributos en cuestión.
- **WHERE:** Impone la condición sujeta a la cual vamos a extraer información.

CONECTORES LOGICOS.
AND: conjuncion
OR: disyuncion
NOT: negacion.

- * = obtener todo

select * FROM ....

igualar llaves primarias y luego condiciones.

```
SELECT columna1, columna2
FROM nombre_de_la_tabla
WHERE condicion;
```

- **Comando IN:** se utiliza para filtrar resultados basándose en una lista de valores específicos.

Supongamos que existe una tabla llamada productos con las columnas id_producto, nombre, y categoria. Si deseamos obtener la lista de todos los productos de las categorías Electrónica y Vestuario, nuestra consulta sería la siguiente:

```
SELECT id_producto, nombre, categoria
FROM productos
WHERE categoria IN ('Electronica', 'Vestuario');
```

→ simplificacion del OR.

- **ORDER BY:** ordena los registros resultantes de una consulta por uno o más campos especificados en orden ascendente (ASC) y descendentes (DESC).

Supongamos que tenemos una tabla llamada productos con las columnas nombre, precio y stock. Si deseamos seleccionar todos los productos, pero ordenarlos primero por el precio de forma ascendente y luego por el stock de forma descendente, nuestra consulta sería la siguiente:

```
SELECT * FROM productos
ORDER BY precio ASC, stock DESC;
```

→ SELECT DISTINCT Nombre. Juego ~~FROM~~ tiene.

- **DISTINCT:** hace que la consulta omita los resultados de la consulta duplicados.
- **BETWEEN:** se utiliza para seleccionar valores dentro de un rango específico.

llamada ventas con una columna llamada monto. Si queremos seleccionar todas las ventas cuyos montos están entre 1000 y 5000, nuestra consulta sería:

```
SELECT * FROM ventas
WHERE monto BETWEEN 1000 AND 5000;
```

→ % expresiones intermedias; _ caracter intermedio.

- **LIKE:** se utiliza para buscar un patrón en una columna.

LIKE se utiliza para buscar un patrón en una columna. *Ejemplo:* Supongamos que tenemos una tabla llamada empleados con una columna llamada nombre. Si queremos seleccionar todos los empleados cuyos nombres comienzan con "A", nuestra consulta sería:

```
SELECT * FROM empleados
WHERE nombre LIKE 'A%';
```

SQL: Ejemplo.

VideoJuego (NombreJuego (PK), consola(PK), precio, genero, año).
Tiene (Jugador_ID (PK, FK), NombreJuego (PK, FK), consola (PK, FK), fecha).

Se quiere conocer los IDs de los jugadores que tengan juegos del tipo Soulslike en PC.

→ ¿Que atributos deseamos obtener? ID_jugador.

→ ¿Desde que tablas deben ser seleccionadas las tuplas? Videojuegos, tiene.

→ ¿Cual es la condicion de seleccion?
NombreJuego y consola deben ser iguales, consola = "PC" y genero = "soulslike".

→ CONSULTA.

```
SELECT ID_jugador FROM Videojuegos, tiene
Where Tiene.NombreJuego = Videojuego. NombreJuego And Tiene.consola = videojuego.consola
AND videoJuego.consola = "PC" AND videojuego.genero = "Soulslike".
```

CONSULTA:

```
SELECT <atributos, lo que se busca>.
FROM <tablas>.
WHERE <condiciones, igualar id de tablas>.
```

→ regla para adoptar.
HAVING solo usar con consultas aritmeticas.

- **Funciones de agregación:** realizan operaciones sobre un conjunto de resultados y devuelven un único valor agregado. **Esto permite obtener medias, máximos, etc. sobre un conjunto de valores y se usan en una sentencia SELECT sobre un atributo de columna.**
  - **COUNT:** cuenta el número de instancias del atributo.
  - **MAX:** entrega el máximo valor de la columna.
  - **MIN:** entrega el mínimo valor de la columna.
  - **AVG:** entrega el valor promedio de la columna.
  - **SUM:** suma los valores de la columna.

Ejemplo

Supongamos que tenemos una tabla llamada pedidos con las columnas producto, cantidad, y precio_unitario. Si queremos obtener información sobre los productos pedidos, incluyendo:

- el número total de pedidos, la cantidad máxima y mínima pedida,
- el precio unitario máximo y mínimo, el precio total de todos los pedidos y el precio promedio por pedido

Nuestra consulta sería la siguiente:

```sql
SELECT
    COUNT(*) AS total_pedidos,
    MAX(cantidad) AS cantidad_maxima,
    MIN(cantidad) AS cantidad_minima,
    MAX(precio_unitario) AS precio_unitario_maximo,
    MIN(precio_unitario) AS precio_unitario_minimo,
    SUM(cantidad * precio_unitario) AS precio_total,
    AVG(cantidad * precio_unitario) AS precio_promedio_por_pedido
FROM pedidos;
```

- **GROUP BY**: se utiliza para agrupar filas que tienen los mismos valores en una o más columnas y luego aplicar funciones de agregación a cada grupo.

  Supongamos que tenemos una tabla llamada ventas con las columnas producto, categoria y cantidad_vendida. Si queremos conocer la cantidad total vendida por categoría, nuestra consulta sería la siguiente:

```sql
SELECT categoria, SUM(cantidad_vendida) AS total_vendido
FROM ventas
GROUP BY categoria;
```

- **CREATE TABLE:** esta sentencia nos permite crear una Tabla para nuestra Base de Datos.

```sql
CREATE TABLE clientes (
    id_cliente INT PRIMARY KEY,
    nombre VARCHAR(50),
    correo_electronico VARCHAR(100),
    fecha_registro DATE
);
```

  - **id_cliente** es una columna de tipo entero que se utiliza como clave primaria (PRIMARY KEY) por lo que tendrá valores únicos para identificar cada fila en la tabla.
  - **nombre** es una columna de tipo varchar que puede contener hasta 50 caracteres.
  - **correo_electronico** es una columna de tipo varchar que puede contener hasta 100 caracteres.
  - **fecha_registro** es una columna de tipo DATE que almacena fechas.

Para crear dos (o más) tablas vinculadas mediante una clave primaria (PK) y una clave foránea (FK), debemos considerar la columna que servirá como "nexo" entre ambas tablas.

```sql
-- Crear la tabla de clientes
CREATE TABLE clientes (
    id_cliente INT PRIMARY KEY,
    nombre VARCHAR(50),
    correo_electronico VARCHAR(100),
    fecha_registro DATE
);

-- Crear la tabla de pedidos con una clave foránea
CREATE TABLE pedidos (
    id_pedido INT PRIMARY KEY,
    id_cliente INT,
    fecha_pedido DATE,
    total_pedido DECIMAL(10, 2),
    FOREIGN KEY (id_cliente) REFERENCES clientes(id_cliente)
);
```

La tabla pedidos tiene una columna llamada id_pedido como clave primaria, y otra columna llamada id_cliente como clave foránea (FOREIGN KEY) que referencia la columna id_cliente de la tabla clientes.

Esto establece una relación entre las dos tablas.

- **DROP:** elimina una tabla.
- **INSERT INTO:** se utiliza para agregar uno o más registros a una tabla.

  INSERT INTO se utiliza para agregar uno o más registros a una tabla. ***Ejemplo:*** Supongamos que tenemos la tabla clientes con las columnas id_cliente, nombre, correo_electronico, y fecha_registro. Si queremos agregar dos nuevos clientes a la tabla, nuestra consulta es la siguiente:

```sql
INSERT INTO clientes (id_cliente, nombre, correo_electronico, fecha_registro)
VALUES (55, 'Juan Perez ', 'juanperez@miejemplo.com', '2023-02-01'),
       (56, 'Jose Diaz', 'josediaz@miejemplo.com', '2023-02-10');
```

- **DELETE:** permite eliminar uno o más registros de una tabla.
- **ALTER TABLE:** se utiliza para realizar cambios en la estructura de una tabla existente, se puede agregar, eliminar, modificar algun tipo de dato en las columnas etc.
- **STORE PROCEDURE:**

  STORE PROCEDURE

  Un procedimiento almacenado (stored procedure en inglés) es un conjunto de instrucciones SQL que se almacenan en una base de datos y que pueden ser llamadas y ejecutadas de forma repetida.

```sql
-- Crear un procedimiento almacenado
DELIMITER //
CREATE PROCEDURE ObtenerEmpleadosConSueldoSuperior(
    IN SueldoLimite DECIMAL(10, 2)
)
BEGIN
    SELECT * FROM Empleados WHERE Sueldo > SueldoLimite;
END //
DELIMITER ;

-- Llamar al procedimiento almacenado
CALL ObtenerEmpleadosConSueldoSuperior(60000.00);
```

Suponiendo que tenemos una base de datos con una tabla llamada Empleados, el procedimiento almacenado ObtenerEmpleadosConSueldoSuperior toma un parámetro SueldoLimite y devuelve todos los empleados cuyo salario sea superior al límite proporcionado.

- **FUNCTION:**

FUNCTION

Una función en SQL es un conjunto de instrucciones que realizan una tarea específica y devuelven un valor. Al igual que los procedimientos almacenados, las funciones pueden encapsular lógica de negocio, pero a diferencia de los procedimientos almacenados, las funciones devuelven un valor y se pueden utilizar en expresiones SQL.

```sql
-- Crear una función
CREATE FUNCTION CalcularSueldoTotal (
    @SueldoBase DECIMAL(10, 2),
    @Bonificacion DECIMAL(10, 2)
)
RETURNS DECIMAL(10, 2)
AS
BEGIN
    DECLARE @SueldoTotal DECIMAL(10, 2);

    -- Calcular el sueldo total
    SET @SueldoTotal = @SueldoBase + @Bonificacion;

    RETURN @SueldoTotal;
END;


-- Utilizar la función en una consulta
SELECT Nombre, SueldoBase, Bonificacion, dbo.CalcularSueldoTotal(SueldoBase, Bonificacion)
AS SueldoTotal
FROM Empleados;
```

La función CalcularSueldoTotal toma dos parámetros (@SueldoBase y @bonificacion), calcula el salario total sumando el salario base y la bonificación, y luego devuelve ese valor.

Luego, la consulta utiliza esta función para determinar el "sueldo total" considerando los datos de SueldoBase y Bonificación.

- **CURSOR:** es un objeto que permite recorrer filas de un conjunto de resultados y realizar operaciones en cada fila manera secuencial y se utilizan principalmente en procedimientos almacenados o bloques de código anidados para procesar registros uno a uno.

- **TRIGGERS:** son bloques de código que se ejecutan automáticamente en respuesta a ciertos eventos en una tabla o vista. Estos eventos pueden ser operaciones de modificación de datos, como INSERT, UPDATE o DELETE. Y son útiles para automatizar acciones o aplicar lógica de negocio en respuesta a cambios en los datos.

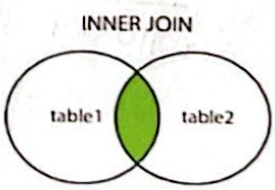
3.5 Cláusula JOIN

La cláusula JOIN se utiliza para **combinar filas de dos o más tablas en función de una condición de relación entre ellas.** Los más comunes son INNER JOIN, LEFT JOIN (o LEFT OUTER JOIN) y RIGHT JOIN (o OUTER JOIN).

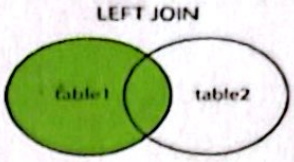
SINTAXIS:

SELECT atributos
FROM tabla1
INNER/PUTER/LEFT/RIGTH JOIN tabla2 ON igualdades de ids.

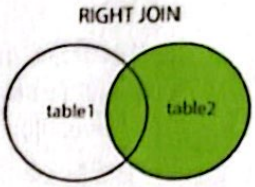
- **INNER JOIN:** devuelve sólo las filas que tienen coincidencias en ambas tablas. En otras palabras, sólo se incluyen las filas que cumplen con la condición de unión especificada.

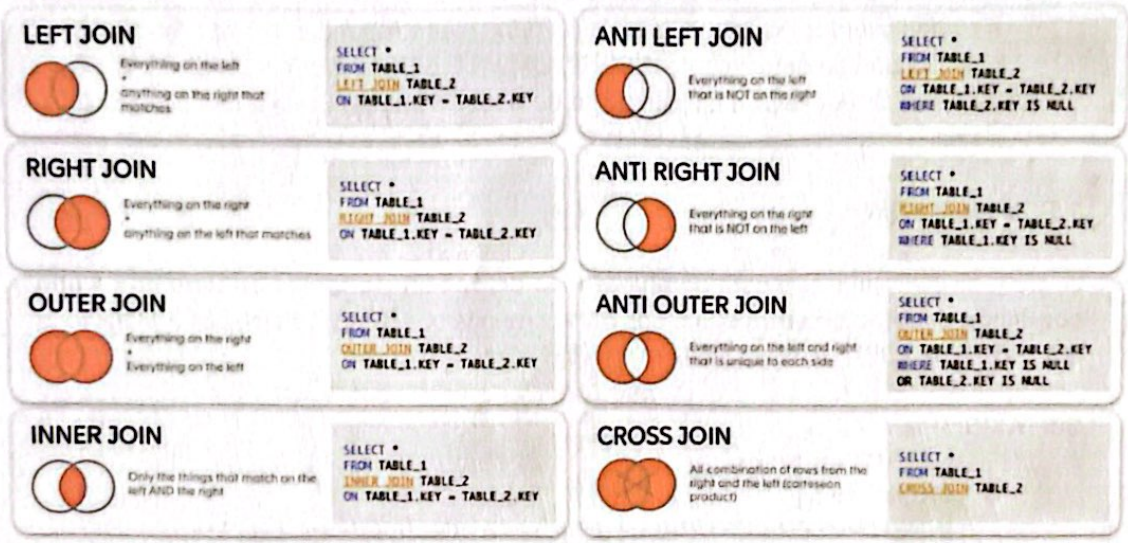


- **LEFT JOIN:** devuelve todas las filas de la tabla izquierda y las filas coincidentes de la tabla derecha. Si no hay coincidencias, se devuelven NULL para las columnas de la tabla derecha.



- **RIGHT JOIN**: devuelve todas las filas de la tabla derecha y las filas coincidentes de la tabla izquierda. Si no hay coincidencias, se devuelven NULL para las columnas de la tabla izquierda.





SQL: Ejemplo.

VideoJuego (NombreJuego (PK), consola(PK), precio, genero, año).
Tiene (Jugador_ID (PK, FK), NombreJuego (PK, FK), consola (PK, FK), fecha).

Se quiere conocer los IDs de los jugadores que tengan juegos del tipo Soulslike en PC.

→ ¿Que atributos deseamos obtener? ID_jugador.

→ ¿Desde que tablas deben ser seleccionadas las tuplas? Videojuegos, tiene.

→ ¿Cual es la condicion de seleccion?
NombreJuego y consola deben ser iguales, consola = "PC" y genero = "soulslike".

→ CONSULTA.

```
SELECT ID_jugador FROM Videojuegos, tiene
Where Tiene.NombreJuego = Videojuego. NombreJuego And Tiene.consola = videojuego.cons
AND videoJuego.consola = "PC" AND videojuego.genero = "Soulslike".
```

CONSULTA:
SELECT <atributos, lo que se busca>.
FROM <tablas>.
WHERE <condiciones, igualar id de tablas>.