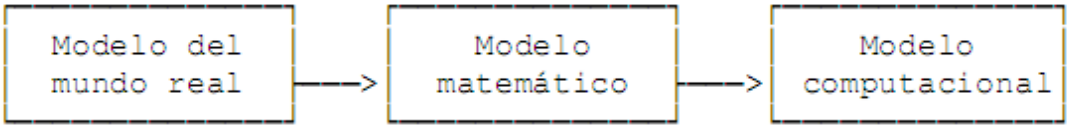
UNIDAD I: ESTRUCTURAS COMPUTACIONALES

GRAFOS

Los grafos sirven para modelizar matemáticamente una estructura de datos.

Su objetivo es obtener información para la toma de decisiones.

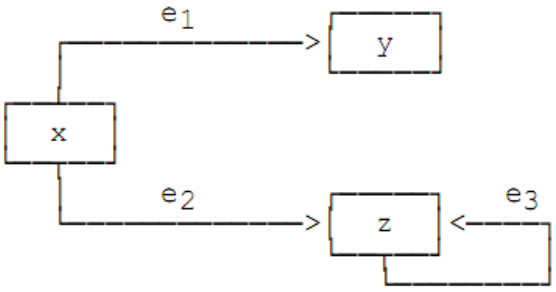
Entre el modelo de la realidad y el computacional existe un modelo matemático



Un grafo es un conjunto de puntos P, también llamados nodos y un conjunto de relaciones E, llamados arcos, se lo representa con G (P,E), donde:

P = {x, y, z}

E = {(x, y); (x, z); (z; z)}



Subgrafo parcial: G' es subgrafo parcial de G si P' = P y E' está incluido en E

Subgrafo: G' es subgrafo de G si P' está incluido en P y E' está incluido en E

Bucle: Son relaciones en la que el nodo de partida es igual al nodo de salida (x,x) donde x pertenece a E.

Maximal: Conjunto de nodos desde los cuales no parte ningún arco que llegue a otro nodo (los bucles no se tienen en cuenta)

Minimal: Conjunto de nodos hacia los cuales no llega ningún arco proveniente de otro nodo (los bucles no se tienen en cuenta)

Grado (+): Cantidad de arcos salientes del nodo

Grado (-): Cantidad de arcos entrantes al nodo

Grado: Hace referencia al grado positivo del nodo

Left L(x): Conjunto de nodos desde los cuales parte un arco que llega al nodo x

Right R(x): Conjunto de nodos hacia los cuales hay un arco desde el nodo x

In-Degree ° L(x) °: Es el grado de entrada del nodo, es decir la cantidad de arcos que entran al nodo

Out-Degree ° R(x) °: Es el grado de salida del nodo, es decir la cantidad de arcos que salen del nodo

Ideal izquierdo L(x): Conjunto de nodos desde los cuales hay paso para llegar al nodo x, se incluyen a sí mismos en el conjunto por lo tanto nunca puede ser vacío.

Ideal derecho R(x): Conjunto de nodos hacia los cuales hay paso partiendo desde el nodo x, se incluyen a sí mismos en el conjunto por lo tanto nunca puede ser vacío.

Ideal Principal: Ocurre cuando el ideal izquierdo o derecho está compuesto por un solo elemento.

Paso: Existe paso si se puede llegar desde el nodo x al nodo siguiendo la dirección de los arcos, solo se evalúa en grafos dirigidos.

Longitud de paso: Cantidad de arcos que tiene el paso desde un nodo x a un nodo y.

Camino: Existe un camino entre un nodo a y b si hay una vinculación directa o indirecta entre los nodos, no se tiene en cuenta la dirección de los arcos, se evalúa en grafos dirigidos o no dirigidos.

Ciclo: Es un paso en donde se recorren nodos distintos y el nodo inicial es igual al nodo final.

Ciclo Hamiltoniano: Es un paso en donde se recorren todos los nodos del grafo una sola vez y el nodo inicial es igual al nodo final.

Grafo restricto: Son aquellos grafos los cuales solo pueden modelizar relaciones que no cumplan con las propiedades de reflexión, simetría y transitividad.

Grafo irrestricto Son aquellos grafos los cuales pueden modelizar relaciones sin importar si cumplen o no ciertas propiedades.

Grafo simple: Es un grafo donde para cada par de nodos a lo sumo hay un arco que los une.

Grafo complejo: Es un grafo que no es simple (más de un arco puede unir un par de nodos).

Grafo conexo: Grafo en el cuál para cada par de nodos existe un camino que los une.

Grafo fuertemente conexo: Grafo en el cuál para cada par de nodos existe al menos dos caminos disjuntos tal que al eliminar un nodo el grafo no quede disconexo.

Grafo completo: Grafo en el cual para cada par de nodos existe un arco que los une.

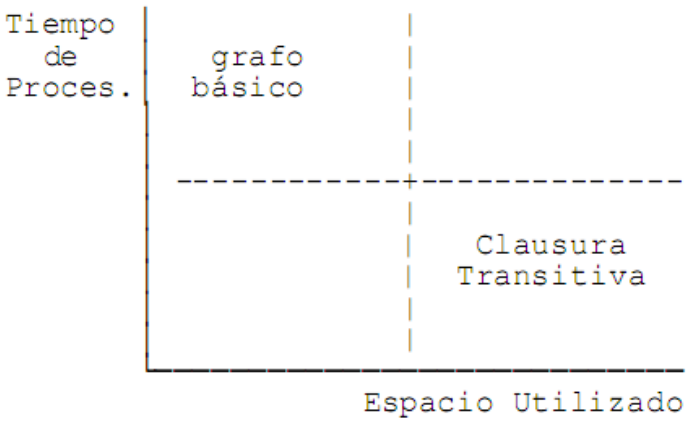
Grafo bipartito: Grafo en el cual existen dos conjuntos de nodos disjuntos (la intersección de ambos es igual al conjunto vacío) y los arcos deben ir desde un nodo de un conjunto a un nodo del otro conjunto, no pueden existir arcos que unan dos nodos de un mismo conjunto.

Grafo básico: Es un grafo en el cual se omiten las relaciones redundantes con el objetivo de disminuir el espacio ocupado pero a su vez aumenta el tiempo de búsqueda ya que es mayor la cantidad de arcos que deben recorrerse.

Clausura transitiva de un grafo: Es un grafo en el cual se detallan todas las relaciones posibles con el objetivo de disminuir el tiempo

de búsqueda (cantidad de arcos recorridos) pero como consecuencia el espacio ocupado aumenta.

Si se comparan ambos modelos se obtiene



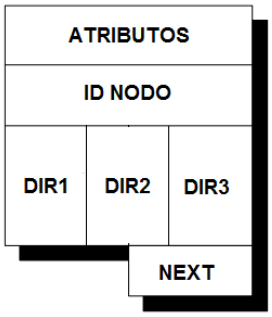
Representación estática: Se pueden utilizar matrices de adyacencia o incidencia las cuales tienen como ventaja permitir un acceso más fácil de forma directa por posición pero la desventaja es que puede llegar a ocupar espacio innecesario y no permite guardar atributos.

Matriz de incidencia: Matriz binaria que relaciona nodos y arcos, se coloca un 0 si el nodo no es un extremo del arco y un 1 si el nodo es un extremo del arco. Habrá un como máximo dos 1 para cada columna (arcos) y la cantidad de 1 en cada fila (nodos) indicará el grado del nodo.

Matriz de adyacencia: Matriz que relaciona nodos con nodos el número será la cantidad de arcos que unen ambos nodos, la matriz será binaria si el grafo es simple. 1 si existe relación entre nodos, 0 si no hay relación.

Representación dinámica: Podría utilizarse listas enlazadas o una representación de Pfaltz. Por ejemplo podría untilizarse una lista de nodos en los cuales se podría guardar información necesaria y además solo se ocuparía solo el espacio necesario por lo tanto el desperdicio sería mínimo, la desventaja sería el acceso a los datos ya que habría que recorrer toda la lista. Las listan enlazadas convienen para grafos dispersos.

Estructura de Celda de tipo "NODO"



Complejidad computacional: Tiempo de procesador utilizado

Complejidad espacial: Espacio utilizado (disco o memoria)

Un algoritmo es mejor que otro si utiliza menos recursos ya sea en tiempo de procesador o en espacio

ÁRBOLES

Un grafo G (P,E) es un árbol si cumple con las siguientes propiedades:

1) ACICLICO: El grafo no tiene ciclos
2) |P| = |E| + 1
3) Para todo a perteneciente a E = > a es desconectante
4) Para todo x,y pertenecientes a P => Existe un walk único (x,y)

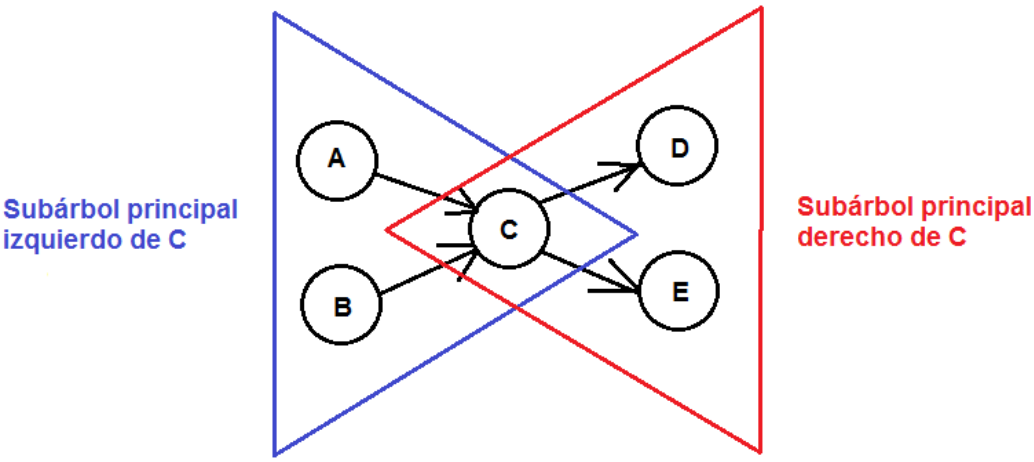
Un nodo solo puede ser un árbol

Subárbol: Es un subconjunto de un árbol que a su vez es también un árbol.

Subárbol principal izquierdo de un nodo: Es el subárbol de un nodo que contiene a los nodos desde los cuales hay un paso

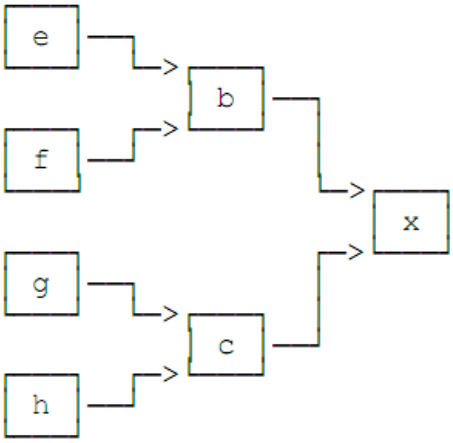
Subárbol principal derecho de un nodo: Es el subárbol de un nodo que contiene a los nodos hacia los cuales hay un paso

En los subárboles de un nodo se incluye a él mismo al conjunto



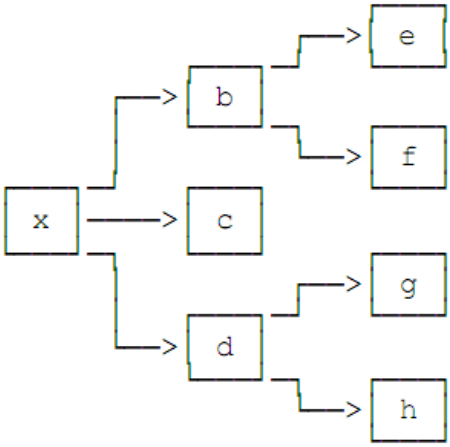
Árbol principal izquierdo: Un árbol es principal izquierdo si existe un único nodo cuyo subárbol principal izquierdo es todo el árbol y cumple con las siguientes condiciones

1) Si x perteneciente a P es maximal => / | R(x) | = 0
2) x es único
3) Para todo y perteneciente a P => | R(y) | = 1 con y != x



Árbol principal derecho: Un árbol es principal derecho si existe un único nodo cuyo subárbol principal derecho es todo el árbol y cumple con las siguientes condiciones

4) Si x perteneciente a P es mínimal => / | L(x) | = 0
5) x es único y se lo denomina raíz
6) Para todo y perteneciente a P => | L(y) | = 1 con y != x



Los árboles computacionales son árboles principales derechos, el nodo que es mínimal se lo denomina raíz y los nodos maximales se los denomina hojas

Grado de un árbol: Está dado por el grado de salida (arcos salientes) del nodo con mayor grado de salida

- Si el árbol es de grado 2 se lo denomina árbol binario
- Si el árbol es de grado 3 se lo denomina árbol ternario
- Si el árbol es de grado r se lo denomina árbol r-ario

Profundidad de un nodo: Es la distancia entre el nodo y la raíz (resta de niveles)

Niveles: Es la clase de equivalencia determinada por los nodos que tienen igual profundidad, la raíz tiene nivel 0

Nivel de un árbol: Es la profundidad más alta es decir el nivel más alto que exista en el árbol

Árbol completo: Un árbol es completo cuando todos sus nodos no maximales (nodos que no son hojas) tienen igual grado de salida

Árbol lleno: Un árbol es lleno cuando es completo y todos sus nodos maximales tienen la misma profundidad, es decir todos están al mismo nivel

Cardinalidad de un árbol lleno: La cardinalidad

$$|P| = \sum_{i=0}^{N} R^i$$

Dónde:

- | P | = Cardinalidad árbol lleno
- N = Nivel del árbol
- R = Grado del árbol

Barrido: Un barrido es un orden definido sobre un conjunto de nodos, es una operación que aplicada sobre un árbol principal derecho devuelve una estructura lineal

Los tipos de barridos que existen son:

Preorden:

1) Informar raíz
2) Visitar subárbol izquierdo
3) Visitar subárbol derecho

Simétrico

1) Visitar subárbol izquierdo
2) Informar raíz
3) Visitar subárbol derecho

Postorden

1) Visitar subárbol izquierdo
2) Visitar subárbol derecho
3) Informar raíz

Niveles: Se recorre el árbol de izquierda a derecho comenzando desde la raíz (nivel 0) hasta el último nivel del mismo

El barrido simétrico solo puede ser aplicado en árboles binarios, los demás barridos pueden aplicarse a cualquier árbol.

Arboles de expresión: Se agrupan los términos según el orden de precedencia de los operadores de forma que queden en las hojas los operandos y los nodos que no son hojas serán los operadores.

Si se barre el árbol de forma simétrica o infija se obtiene la expresión original.

Arboles r-arios: En estos árboles se cumple que para todos los nodos ‘x’ pertenecientes a P

$$|R(x)| <= r$$

Para calcular el espacio desperdiciado se utiliza:

$$(r-1) * | P | + 1$$

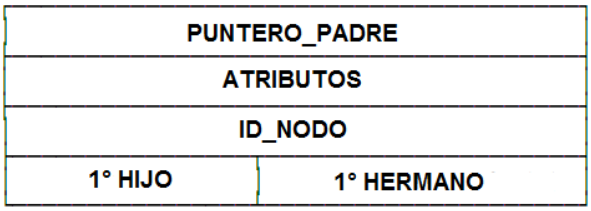
Donde r = Grado del árbol y | P | = N° de nodos

Transformada de Knuth: Se utiliza para transformar árboles r-arios en árboles binarios para lograr esto a cada nodo se le coloca como hijo izquierdo el primer nodo de su Right R(x) en el árbol original y como hijo derecho el hermano más próximo, es decir el nodo más próximo que tiene el mismo padre y está al mismo nivel en el árbol original

Una vez transformado el árbol se obtiene que

- Preorden original = Preorden Knuth
- Postorden original = Simetrico Knuth

Los registros se representan de la siguiente forma:



Arbol de busqueda binario

Se realizan las búsquedas en archivos organizados como árboles para esto se ordena el árbol de forma que los descendientes izquierdos sean menores al nodo especificado y los descendientes derecho sean mayores, si se recorre el árbol de forma simétrico se obtiene una lista ordenada, este método permite la búsqueda, eliminación e inserción de forma eficiente

Insercion: Si la búsqueda no tiene éxito inserta a la izquierda si es menor a la derecha si es mayor

Eliminación: Si el nodo no tiene hijos se elimina de forma directa, si tiene un hijo, este toma su lugar, si tiene dos hijos toma su lugar el predecesor de menor valor del subárbol derecho y se actualiza el subárbol según las condiciones mencionadas

Cardinalidad: Cantidad de nodos

Árbol balanceado: Árbol que para todo nodo se cumple que la Cardinalidad del subárbol izquierdo es igual a la Cardinalidad del subárbol derecho o difiere en 1

Arbol AVL: Arbol que para todo nodo se cumple que el nivel del subárbol izquierdo es igual al nivel del subárbol derecho o difiere en 1

Nodo critico: Nodo en el cual el árbol se desbalancea

Para calcular el “desbalance” de un nodo se resta ya sea la Cardinalidad (Balanceado) o el nivel (AVL) según lo que sea el árbol si no hay desbalance el resultado da 0

La rotación de arboles se utiliza para balancear un árbol en un punto critico determinado

Si el árbol se encuentra desbalanceado a izquierda se realiza una rotación a derecha

Si el árbol se encuentra desbalanceado a derecha se realiza una rotación a izquierda

La rotación garantiza que el barrido simétrico del árbol rotado sea igual al barrido simétrico del árbol original desbalanceado

UNIDAD II: ALGORITMOS

QUICKSORT

Los algoritmos se dividen en dos tipos: Algoritmos de ordenamiento externo e interno. Los algoritmos externos sirven para ordenar un conjunto muy grande de datos que no pueden ser manejados en memoria y los internos se aplican en conjuntos que pueden manejarse completamente en memoria

El algoritmo Quicksort es un método de ordenamiento de intercambio con partición (ordenamiento rápido).

El algoritmo utiliza un vector con una cierta cantidad de elementos y consiste en seleccionar un elemento del mismo que será tomado como pivote y se asumirá que los elementos que estén a su izquierda son menores que este y los que están a la derecha son mayores al mismo, para esto se utiliza dos punteros UP y DOWN y un FLAG que indica el puntero que está activo (es decir el que se esta moviendo) inicialmente FLAG = UP. El algoritmo irá comparando los valores de los elementos del vector y en base a eso ira moviendo uno de los dos punteros, el algoritmo finaliza cuando los punteros apuntan a la misma posición es decir UP = DOWN. Para obtener el vector ordenado se deberá ejecutar el algoritmo en todos los subvectores que vayan apareciendo, esto se realiza utilizando recursividad en caso de que el subvector ya se izquierdo o derecho tenga elementos.

El código es el siguiente:

```
WHILE (DOWN < UP)
        IF FLAG = UP
                IF (VEC (DOWN) < VEC (UP))
                        UP--
                ELSE
                        INTERCAMBIO
                        DOWN++
                        FLAG = DOWN
        ELSE
                IF (VEC (DOWN) < VEC (UP))
                        DOWN++
```

```
ELSE
        INTERCAMBIO
        UP--
        FLAG = UP
```

HEAPSORT

Árbol de orden parcial implementado dentro de un vector por medio de un barrido por niveles. Para ejecutar el algoritmo primero debe realizarse:

Carga del árbol

Dado un conjunto de elementos se carga el árbol leyendo un elemento del conjunto que se lo pondrá como raíz y el siguiente elemento será el hijo izquierdo del mismo si este se encuentra en NULL o será el hijo derecho si este el izquierdo está ocupado y el derecho está en NULL, si ambos hijos están ocupados se repite el proceso para el hermano más próximo (mismo padre y mismo nivel) o se desciende hacia abajo en caso de que estos tampoco tengan hijos disponibles. Se debe cumplir la condición de que el padre sea mayor que sus hijos por lo tanto a medida que se van colocando los nodos se van intercambiando las posiciones de forma que quede un árbol de orden parcial

Carga del vector asociado

A medida que se van ingresando elementos al árbol se va completando el vector.

Se lee el primer elemento del conjunto y se lo coloca en la primer posición libre (1), luego se lee el otro elemento y se lo coloca en la posición libre más cercana (2), se procede a comparar los elementos para esto se busca el padre de (2) y se los compara, si el padre es mayor o igual que el hijo se deja como está el vector, si el hijo es mayor que el padre se realiza un intercambio. Los hijos deben compararse con sus padres hasta que se verifique estos son menores a sus padre. Para determinar el padre y los hijos de un nodo se utiliza:

- Padre = int [i / 2] (se toma el entero)

- Hijo izquierdo = i * 2
- Hijo derecho = (i * 2) + 1

Donde i es la posición ACTUAL del nodo que se va a comparar, el conteo de elementos empieza desde 1 hasta N (NO desde 0)

De está forma se obtendrá un árbol barrido por niveles implementado en un vector sin haber realizado el barrido, lo cual disminuye la complejidad

Ejecución del algoritmo

1) Realizar un barrido por niveles de donde obtengo un vector
2) Intercambiar el vector(1) con el vector(N)
3) N = N – 1 (el elemento en N queda “fijo”)
4) Rearmar el árbol para que sea de orden parcial de 1 a N
5) Si N != 1 repetir el proceso desde 1)
6) Finalización del algoritmo

Orden de complejidad: Funciones aproximadas, estas funciones suelen ser ascendentes

Quicksort = O(N log N)

Heapsort = $\log_2 (N+1) * N$

- El Quicksort es más sencillo y tiene menos líneas de codigo que el Heapsort aunque tiene un orden de complejidad más alto
- El Heapsort generalmente es más rápido que el Quicksort, aunque no en todos los casos
- El Quicksort puede llegar a ejecutarse menos veces que el Heapsort en el mejor de los casos

SORT TOPOLOGICO

El algoritmo consiste en transformar un conjunto de precedencias en una lista lineal simple ordenada en la cual ningún elemento posterior precede a uno anterior dado

Se lo utiliza para determinar periodos de tiempo o cantidad máxima/mínima de recursos utilizados. Utiliza el concepto de paso.

Dado un grafo finito G(P,E)

- S es Sort Topologico de G si solo sí G es ACICLICO
- S será único <=> G es de orden total por lo tanto S será el grafo básico de G

El algortimo se ejecuta buscando un mínimal (x) de P y colocándolo en la estructura de salida, se procede a eliminar el nodo y sus relaciones del grafo G, ahora se busca otro mínimal, en caso de exisitir varios se elige uno de forma arbitraria y se procede a repetir el proceso hasta que todos los nodos se encuentren en la lista lineal.

Para implementar el TSORT (algoritmo del Sort Topologico) se podrían utilizar estructuras estáticas como por ejemplo una matriz de adyacencia pero al existir actualizaciones dinámicas no es adecuado ya que requiere conocer de antemano la cantidad de nodos del grafo, por lo tantos se utilizará estructuras dinámicas. La representación de Pfaltz podría utilizarse ya que utiliza una lista de nodos y arcos pero la desventaja es que su implementación es muy compleja así que se utiliza otra estructura que mantiene la filosofía de Pfaltz.

Si los nodos fueran tareas y estas tuvieran duración en caso de haber tareas que pueden realizarse independientemente, es decir son minimales se elegirá el que tenga mayor tiempo

ALGORITMO DE HUFFMAN

Los problemas que originan el uso del algoritmo son:

1) Insuficiente espacio en discos o memoria
2) Tiempo de transmisión prolongados y costos elevados

El algoritmo de Huffman utiliza un código variable para cada carácter sin utilizar un prefijo. Los caracteres que tengan más frecuencia en un alfabeto tendrán una codificación que usará la

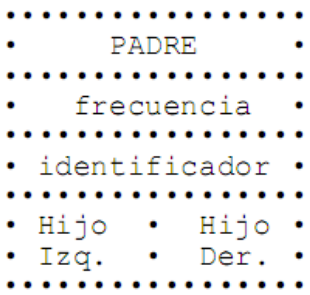
menor cantidad de bits posible y los que menos se repitan usarán una cantidad de bits mayor.

Para esto Huffman utiliza como estructura de datos

1) Una tabla de frecuencias donde se guardarán los caracteres y la cantidad de veces que se utilizaron en el texto, en la tabla también puede guardarse un FLAG que indica si el nodo es hijo (1) o raiz (0) y la dirección del nodo se recomienda ordenar la tabla de forma descendente por el campo frecuencia (los que tienen más frecuencia están arriba)

2) Un árbol binario donde
    - Las hojas son los caracteres
    - Cada hijo izquierdo distinto de NULL representa un 0 binario
    - Cada hijo derecho representa distinto de NULL un 1 binario

El registro que representa al nodo en el árbol binario es el siguiente



El algoritmo funciona de la siguiente manera

1) Buscar los 2 caracteres con frecuencia más baja de la tabla de frecuencias comenzando por los 2 últimos.
    - Si un carácter tiene frecuencia 0 no se lo tiene en cuenta
    - Si 2 o más caracteres tienen igual frecuencia se eligen 2 cualquiera
2) Armar un árbol binario donde la raíz tendrá como frecuencia la suma de las frecuencias de los dos caracteres, sus hijos serán los caracteres, el campo padre e identificador estarán en blanco. Los caracteres serán las hojas y se guardara el

carácter, la frecuencia y la dirección del padre y sus hijos estarán en NULL.

- Si los 2 caracteres tiene la misma frecuencia el que más arriba este en la tabla será el hijo izquierdo y el otro el hijo derecho.
- Si los 2 caracteres tienen distinta frecuencia el mayor será el hijo derecho y el menor el izquierdo.
- Primero se consideran los caracteres originales de frecuencia más baja, si ya no hay más o hay pero de frecuencias superiores se considerarán los caracteres “alpha” empezando por el que este más abajo en la tabla

3) Repetir desde 1) pero esta vez también debe considerarse los nuevos caracteres “especiales” creados, se repite el proceso hasta que quede un único árbol binario donde las hojas sean caracteres y la raíz tenga como frecuencia la suma de las frecuencias de los caracteres originales

Una vez obtenidos las estructuras, la tabla de frecuencias y el árbol a partir de esta se puede codificar y decodificar una secuencia de bits.

Codificación: Para codificar un carácter primero se requiere obtener la dirección del nodo que contiene al carácter que se quiere codificar, se lo puede obtener de la tabla de frecuencias. Una vez obtenida la dirección se recorre el árbol de forma ascendente desde la hoja hasta la raíz de forma que a medida que se asciende se insertará en una pila un 0 si el hijo es izquierdo o un 1 si el hijo es derecho así hasta llegar a la raíz donde padre = NULL, en este momento se tiene la codificación de forma invertida pero al vaciar la pila se obtendrá el código binario que luego se utilizará en la decodificación.

Decodificación: Para decodificar un carácter se parte de la raíz del árbol y luego se va descendiendo hacia las hojas en caso de leer un 0 se desciende por el hijo izquierdo y si es 1 por el hijo derecho, así hasta llegar a un nodo que tenga sus hijos en NULL, el cual tendrá el carácter buscado.

Resumiendo se puede decir que el algoritmo de Huffman está compuesto por 4 etapas

1) Generación de la tabla de frecuencias
2) Generación del árbol binario
3) Compresión o compactación de los datos
4) Descompresión o descompactacion de los datos

Las primeras 2 hacen referencia a las estructuras en memoria y las otras 2 al tratamiento de los datos

Para obtener el porcentaje de compactación se realiza:

*Bits originales = N° Caracteres * 8*

*(Bits utilizados / Bits originales) * 100*

REPRESENTACION DE PFALTZ

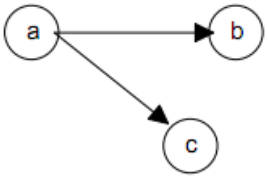
Un grafo es bipartito si el conjunto de nodos está dividido en subconjuntos disjuntos de forma que no existan nodos que se relacionen con otros nodos del mismo conjunto

Sea G (P,E) y Q, R incluidos en P, G es bipartito si:

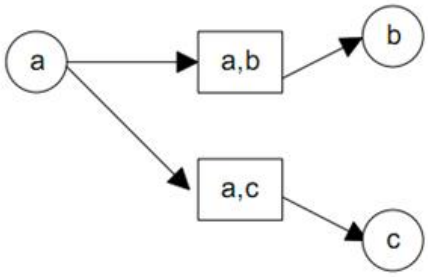
- Q Union R = P
- Q Interseccion R = { }
- Para todo (x,y) perteneciente a E =>
    - (x pertenece a Q ^ y pertenece a R) ó
    - (x pertenece a R ^ y pertenece a Q)

La representación algebraica de Pfaltz consiste en generar un grafo G' (P', E') a partir de un grafo G (P, E) irrestricto donde cada arco de E representará un nodo de P', que también incluye a los nodos de P y E' tendrá el doble de elementos de E ya que tendrá relaciones donde la primer componente del arco se relacionará con el nodo arco y el nodo arco se relacionará con la segunda componente

Sea:
G(P,E) una $E_d(P,E, f_1,f_2,\ldots,f_n, g_1,g_2,\ldots,g_n)$.
P={a,b,c}
E={(a,b),(c,d)}



Entonces su representación algebraica de Pfaltz G'(P',E')
P'= P U E = { a, b, c, (a,b), (a,c) }
E'={ (a,(a,b)), ((a,b),b), (a,(a,c)), ((a,c),c) }

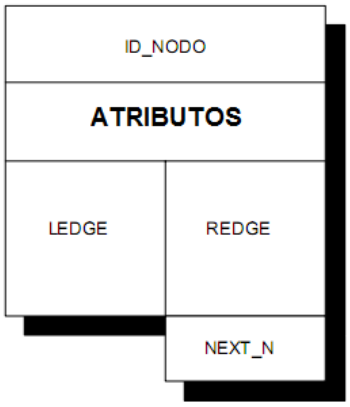


Para representar un grafo irrestricto hasta ahora se conocían dos formas: las matrices de adyacencia e incidencia la desventaja de este es que hay que conocer previamente la cantidad de arcos y nodos y en caso de que se requiera un funcionamiento dinámico no será muy eficiente ya que a la hora de agregar o quitar nodos o arcos hay que crear nuevamente las matrices.

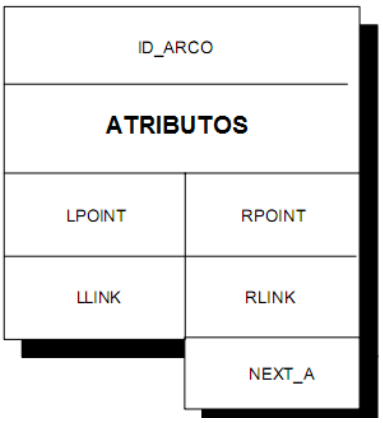
Para esto Pfaltz utiliza listas enlazadas en las cuales se agregan o quitan nodos de forma dinámica lo cual es más eficiente que las matrices en caso de que haya una gran cantidad de nodos en el grafo a representar

La representación computacional de Pfaltz utiliza como estructura de datos una lista de nodos y otra de arcos

Estructura de Celda de tipo "NODO"



Estructura de Celda de tipo "ARCO"



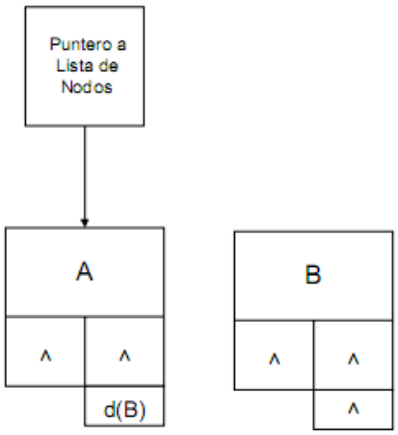
NODO

1) ID_NODO: Identificador único del nodo
2) ATRIBUTOS: Atributos del nodo
3) LEDGE: Puntero a la direccion del primer arco de una lista de arcos que llegan al nodo
4) REDGE: Puntero a la direccion del primer arco de una lista de arcos que parten del nodo
5) NEXT_N: Puntero al siguiente nodo de la lista de nodos

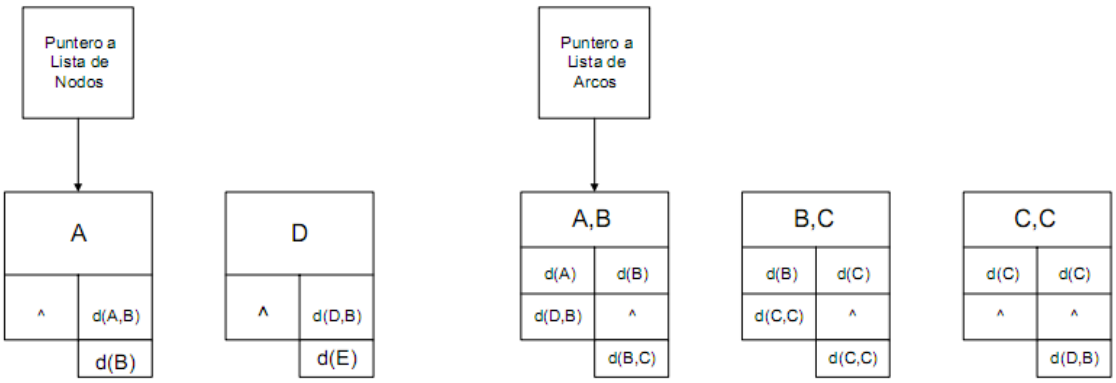
ARCO

1) ID_NODO: Identificador único del arco
2) ATRIBUTOS: Atributos del arco
3) LPOINT: Puntero a la direccion del nodo de donde parte el arco
4) RPOINT: Puntero a la direccion del nodo hacia donde llega el arco
5) LLINK: Puntero a la proxima direccion de un arco una lista de arcos con igual RPOINT, es decir que llegan al mismo nodo
6) RLINK: Puntero a la proxima direccion de un arco una lista de arcos con igual LPOINT, es decir que llegan al parten del mismo nodo
7) NEXT_N: Puntero al siguiente arco de la lista de arcos

En la implementacion de Pfaltz se realiza primero el ingreso de nodos donde se recibe una secuencia de nodos que corresponden a un grafo, en el caso de los nodos solo se actualiza el campo NEXT_N por ahora



Una vez finalizado el ingreso de nodos se realiza el ingreso de arcos el cual debe tener un orden especifico



A medida que se ingresan los arcos se van actualizando los LEDGE y REDGE de los nodos y los LLINK y RLINK de los arcos las direcciones de los nodos se representa con d(X) y de los arcos con d(X,Y).

Siempre arrancar desde el arco a completar para actualizarlo, es decir no tengo que tener en cuenta los nodos anteriores y también actualizar los nodos anteriores a medida que se agregan nodos posteriores.

ALGORITMO DE SIKLOSSY

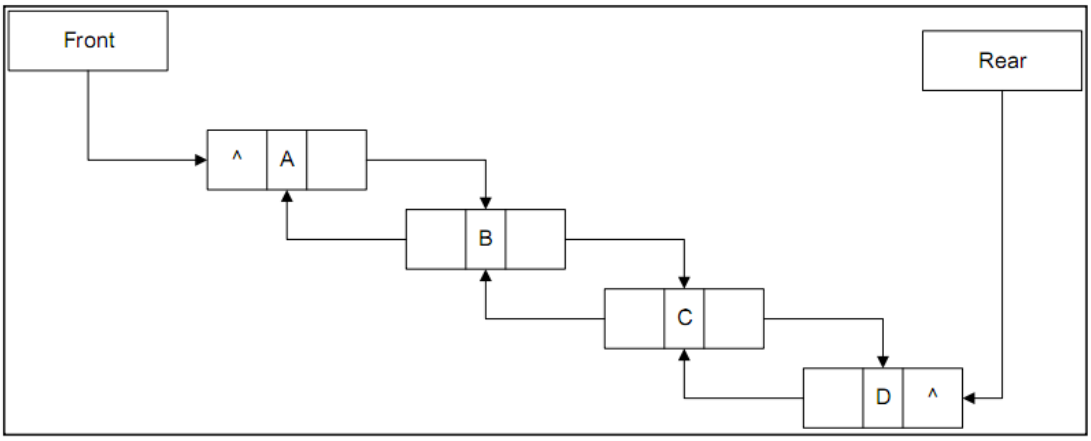
Anteriormente se utilizaron listas con linkeo simple las cuales trataban de minimizar el espacio innecesario por lo tanto es natural pensar que una lista de doble linkeo ocupará mayor espacio que una de simple linkeo al tener dos punteros (al anterior y siguiente registro de forma de poder recorrer la lista en ambos sentido. El

algoritmo de Siklossy permite minimizar el espacio utilizado haciendo uso de un solo campo link y sin aumentar el tiempo de acceso mediante el concepto del or exclusivo.

p	q	p + q
0	0	0
0	1	1
1	0	1
1	1	0

Donde se obtiene que:

(p + q) + p = q
(p + q) + q = p



La lista tendría un FRONT que apuntaría al primer registro de la lista y un REAR que apuntaría a la cola del registro y cada registro tendría dos punteros apuntando al anterior y al siguiente.

Para arrancar a dar de alta los nodos hay que tener en cuenta que las direcciones de FRONT y el REAR son = 0000. Tener en cuenta lo siguiente:

- pX = Dirección del nodo anterior
- qX = Dirección del nodo posterior

Por lo tanto se pide una dirección al administrador de memoria, este devuelve la dirección 0101 El FRONT y el REAR apuntarán a esta dirección, el registro en esa dirección tendrá un campo con el valor de A y en el campo dirección tendrá la suma con el OR EXCLUSIVO que representará el (p + q) en este caso:

O también se dice que pA = 0000 y qA = 0000 => (p + q) = 0000

Ahora se ingresará un nuevo nodo por lo tanto se pide una nueva dirección de memoria, ahora devolverá el administrador un 1000 en esa dirección habrá un registro con un campo que tendrá el valor de B y un campo con la dirección (p + q) que en este caso

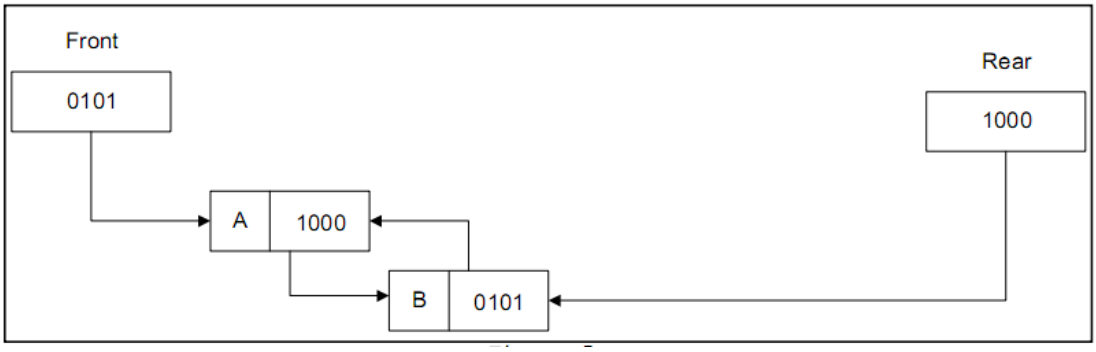
pB = 0101 y qB = 0000

El nodo anterior se lo puede obtener del REAR antes de que se actualice y el Q siempre será 0000 para un nuevo nodo. De esta forma se obtiene que

pB + qB = 0101 + 0000 (OR EXCLUSIVO) = 0101

El campo del registro A también deberá actualizarse ya que ahora

pA = 0000 y qA = 1000 => (pA + qA) = 1000



Ahora se ingresa el nodo C con dirección 0111 por lo tanto el nodo B se actualiza ya que

pB = 0101 y qB = 0111 => (pB + qB) = 0010

Y el registro de C tendrá un campo con el valor de C y un campo con (pC + qC) donde

pC = 1000 (REAR) y qC = 0000 => (pC + qC) = 1000

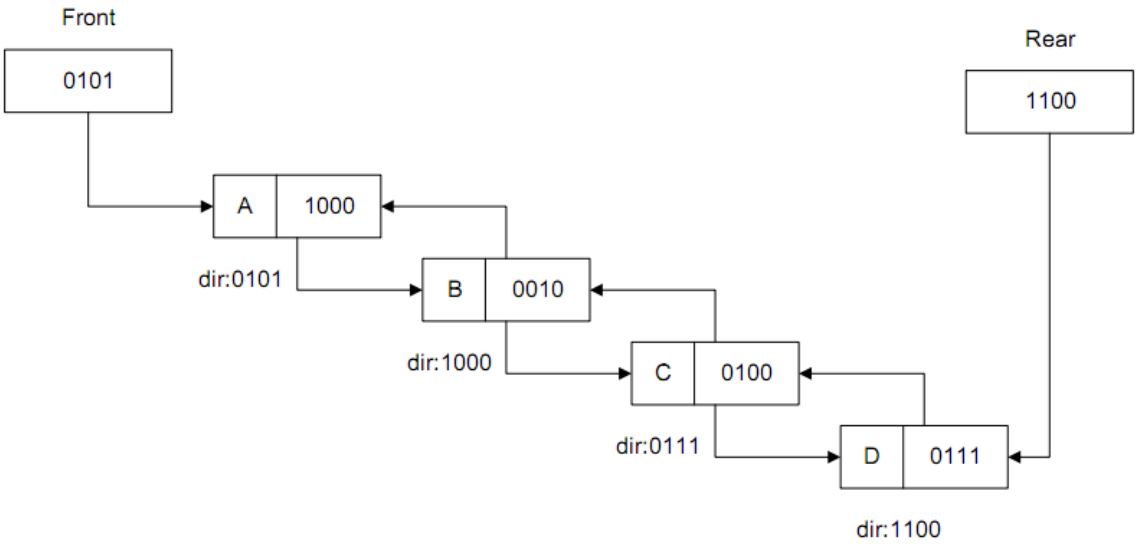
Luego se ingresará el nodo D con dirección 1100 por lo tanto

pC = 1000 y qC = 1100 => (pC + qC) = 0100

Y para el nodo D, el registro tendrá el campo con el valor D y el campo (p + q) con

pD = 0111 y qC = 0000 => (pC + qC) = 0111

Quedando finalmente:



Con este nueva estructura creada se puede acceder a los nodos posteriores y anteriores utilizando el campo (p + q) sabiendo que

- pX = Dirección del nodo anterior
- qX = Dirección del nodo posterior

por lo tanto

- (pX + qX) + qX = Dirección nodo anterior
- (pX + qX) + pX = Dirección nodo posterior

Por ejemplo como quedaron estas direcciones

- Dir A = 0101
- Dir B = 1000
- Dir C = 0111
- Dir D = 1100

Si analizo el nodo B y quiero conocer sus nodos anterior y posterior hago

- (pB + qB) + qB = dir(A) = Nodo anterior (i-1)
- (0010) + 0111 = 0101 = dir(A)
- (pB + qB) + pB = dir(C) = Nodo posterior (i+1)
- (0010) + 0101 = 0111 = dir(C)

UNIDAD III: ACCESO A DATOS

HASHING

Uno de los principales objetivos de las computadoras es procesar grandes volúmenes de datos en el menor tiempo posible u obtener a un registro por medio de una clave en el menor tiempo posible por lo tanto un algoritmo será considerado ineficiente si el tiempo de espera es mayor a lo que el usuario considera lógico.

Las claves pueden estar formadas por uno o más campos los cuales están asociados con el registro de forma que pueda identificarse unívocamente, las claves pueden ser:

- **Internas**: Están contenidas dentro del registro
- **Externas:** Se encuentran en archivos externos al archivo de datos

Los algoritmos de recuperación de datos o de búsqueda pueden devolver un registro o un puntero a estos. Estos algoritmos también deben verificar que en caso de insertar un nuevo registro la clave de este no exista previamente ya que la clave debe ser única por lo tanto no puede ser duplicada (DUPKEY) o en caso de querer obtener un registro que la clave del mismo exista (INVALID KEY). Existen distintos tipos como por ejemplo:

Búsqueda secuencial: Es la más sencilla de implementar consiste en recorrer el archivo de claves en forma secuencial hasta hallar la clave buscada o hasta llegar a la última posición de la misma. La principal desventaja es que en el peor de los casos se deberá leer el casi todo el archivo varias veces.

Búsqueda secuencial indexada: Consiste en el archivo de datos y una tabla de índices, la cual contiene el valor de la clave y la dirección de los datos de esa clave, primero se busca la clave en la tabla de índices y si la clave no se encuentra en la tabla se busca la clave inmediata superior (i+1) para posicionarse en la clave (i) y recorrer secuencialmente el archivo de datos hasta encontrar la clave buscada o hasta llegar a la clave inmediata superior (i+1) lo cual producirá que devuelva que no existe la clave (INVALID KEY). La ventaja de este método es que reduce el tiempo de búsqueda pero el espacio ocupado será mayor ya que requiere una tabla adicional. Pueden utilizarse múltiples tablas para acceder de forma más rápida a los datos pero requerirá más espacio.

Función de Hash

Los algoritmos de búsqueda están constantemente realizando comparaciones por lo tanto para aumentar la eficiencia del algoritmo se busca que se realice la menor cantidad de comparaciones posibles, una forma de solucionar este problema sería que exista una relación directa entre la clave del registro y la dirección donde se alojan los datos del mismo. ***En base a esta idea se creó la función de Hash la cual consiste en una función que dada una clave devuelve el subíndice de la tabla donde se encuentra la dirección con los datos del registro buscado.***

Sin embargo problema más común que presentan las funciones de hash es devolver un mismo valor para dos claves diferentes, lo cual se denomina colisión.

Colisión: Sean K1 y K2 claves pertenecientes a los registros R1 y R2 con K1 != K2 y sea h(K) una función de hash. Si h(K1) = h(K2) entonces se dice que se produjo una colisión.

Las colisiones no pueden evitarse tarde o temprano ocurrirán ya que la cantidad de claves posibles siempre es mayor a la cantidad de posiciones de la tabla si bien existen métodos para reducir la probabilidad de que se produzca una colisión está nunca será de 0%. Por lo tanto se puede decir que una función de Hash será más eficiente cuando minimiza el N° de colisiones y ocupa los elementos de la tabla de manera uniforme. Si se aumenta el tamaño de la tabla

disminuye la probabilidad de que se produzcan colisiones pero como contraparte se ocuparía un mayor espacio.

El principal objetivo de las funciones de hash es disminuir el número de colisiones utilizando la menor memoria posible. A continuación de mostrarán algunas funciones de ejemplo pero en la realidad no se utiliza una sola sino que se utilizan combinaciones de varias

Método de la división o del módulo

Consiste en obtener un numero entre 1 y MAXTAB (Tamaño máximo de la tabla) a partir del resto de la división entre la clave y MAXTAB.

Sea

- MAXTAB = 947
- CLAVE = 2866
- hash(K) = $\text{mod}_{\text{MAXTAB}}$

hash(CLAVE) = 25

Método del cuadrado medio

Consiste en obtener un número entre 1 y MAXTAB a partir de los dígitos del valor que se obtiene al multiplicar la clave por sí misma, si la tabla tiene N dígitos máximos, el numero obtenido deberá tener N dígitos.

Sea

- MAXTAB = 947
- CLAVE = 2866
- hash(K) = Dígitos medios de K^2

CLAVE^2 = 8213956 => hash(CLAVE) = 139

Sea MAXTAB = 1013 y CLAVE = 2866

hash(CLAVE) = 2139 o hash(CLAVE) = 1395

Método de dobles

Consiste en dividir los dígitos la clave en dos o más partes iguales y realizar la operación OR exclusivo entre los valores binarios de cada una de esas partes y finalmente el resultado pasarlo a decimal para obtener un N° entre 1 y MAXTAB

Sea K = 2866

K1 = 28 = > Binario (K1) = 11100

K2 = 66 => Binario (K2) = 1000010

Aplicando operación OR EXCLUSIVO entre 11100 y 1000010 se obtiene1011110.

Decimal(1011110) = 94 => hash(K) = 94

Para tratar las colisiones los métodos que existen pueden dividirse en dos

1) Rehashing (Direccionamiento abierto)
2) Chaining (Encadenamiento)

Rehashing

Consiste en aplicar una segunda función de hash hasta encontrar una posición libre, la solución más trivial o más simple es que si la posición de la tabla que coincide con el valor que devuelve la función de hash está ocupada se ocupa la siguiente posición libre de la tabla, es decir que la función de rehashing depende exclusivamente del valor que devuelva la función de hashing

Si hash(k) = dir => rehash(dir)

Ejemplo rehash(dir) = $(dir + c) \bmod_{MAXTABLE}$ donde c y MAXTABLE son números primos

Resumiendo dada una clave K si se le aplica una funcion de hash hash(K) = dir y se obtiene un valor de subíndice (dir) y el elemento de la tabla table(dir) se encuentra ocupado se aplicará una función de reasignación rhash(dir) de forma recursiva hasta encontrar una posición libre en la tabla, debe evitarse los casos en que:

1) La funcion de rehash quede buscando eternamente una posición libre de la tabla.
2) La función de rehash informe que no quedan espacios libres erróneamente por ejemplo con rhash(dir) = dir + 2 solo tendrá en cuenta las posiciones pares por lo tanto cuando se llene informará que no quedan espacios libres descartando las posiciones impares
3) La función de rehash produzca que un elemento de la tabla tenga más probabilidad de ser ocupado que otros, esto se lo conoce como clustering y puede evitarse a través del doble hashing el cual consiste en utilizar una función definida por el programador dentro de la función de rehashing que toma como valor el valor de la clave (k) por lo tanto ahora la función de rehash dependerá del valor de la clave y del subíndice que devuelve la función de hash como por ejemplo
   Rhash(dir) = (dir + f(key)) $\text{mod}_{\text{MAXTABLE}}$ donde f(key) = 10 * key

Una funcion de reasignación rhash será óptima cuando la recursividad de la misma cubra la totalidad de elementos de la tabla

Otra forma de evitar la colisiones en las funciones de rhash es hacer que dependan del N° de veces que se utilizó

Rhash(dir, i) = (i+1) * dir $\text{mod}_{\text{MAXTABLE}}$ con i inicializada en 1

Los principales problemas al utilizar funciones de rehashing (direccionamiento abierto) es que:

1) Dependen de tablas estáticas es decir en caso de que están queden chicas deberán expandirse y por lo tanto se deberán actualizar las funciones en base a los nuevos valores de MAXTAB
2) En caso de dar de baja un clave la posición correspondiente a la misma quedará libre por lo tanto cuando se quiera recuperar otra clave que haga que la función de hash retorne el valor del subíndice que acaba de liberarse no se podrá saber si esa clave que se quiere recuperar existe o está en otras posiciones siguientes generadas por la función de rehash

Chaining o encadenamiento

Consiste en agrupar en una lista enlazada los pares clave/dirección que colisionen entre sí de forma de agruparlos para esto se utiliza Hashing into buckets el cual utiliza como estructura de datos una tabla denominada área base y una lista denominada área de overflow que tiene un registro con 3 campos uno para la clave, otro para la dirección y otro para el puntero al siguiente registro, cuando una función de hash produce una colisión para una clave k2 dada ya que el subíndice fue ocupado por otra clave k1 se enlaza un registro con los datos correspondientes a la lista correspondiente a k1 que tiene sus datos en la area base, si se quiere dar de baja una clave solo se elimina un nodo de la lista. El principal problema de esta solución es la cantidad de espacio ocupada por los punteros pero aun así reduce los problemas de direccionamiento abierto como iterar muchas veces hasta encontrar una posición libre, para medir la eficiencia se lo hace en base a la cantidad de nodos accedidos por lo tanto si el N° de nodos a recorrer es muy grande se perderá mucho tiempo para recuperar una clave

Hash solo es eficiente en accesos directos de claves

ÁRBOL B

Los arboles B se utilizan para obtener un registro por medio de una clave en un tiempo acotado, tambien para recorrer secuencialmente un conjunto de datos entre dos valores de clave dados sin necesidad de consultar todo el archivo.

El árbol B parte del concepto de árbol R – arios y su principal característica es que los datos se encuentran en el último nivel (todos sus maximales están en el mismo nivel), de esta forma se consigue uniformidad en el acceso a los datos

Condiciones de un árbol B

1. Cada nodo del árbol tiene como máximo R hijos
2. Cada nodo del árbol tiene mínimo R/2 hijos a excepción de la raíz y las hojas

3. Todas las hojas se encuentran al mismo nivel
4. Un nodo con R hijos tiene R -1 claves

Un split puede producirse al dar de alta claves en en un árbol B

Si se quiere dar de alta una clave en un nodo maximal lleno este deberá dividirse en 2 nodos maximales colocando la mitad de las claves en uno y la otra mitad en otro, la clave media deberá llevarse al nodo nivel superior

En el peor de los casos el efecto se propagara hasta la raíz del árbol haciendo que este aumente su nivel

El factor de carga es un porcentaje de carga de los nodos de un árbol B, el cual se aplica solo en el proceso de carga inicial del árbol

Un factor de carga al 100% se utiliza cuando el Arbol B se representa un índice asociado a una tabla que sea de consulta con pocas actualizaciones o que sea histórica

Un factor de carga al 75% - 85% se utiliza cuando el Arbol B se representa un índice asociado a una tabla que tenga muchas actualizaciones (inserts, updates, deletes) en forma online

Arbol B mas eficiente en búsquedas secuenciales y en búsqueda de claves por rango

UNIDAD IV: BASES DE DATOS

BASES DE DATOS

Una Base de datos es un conjunto de datos persistentes e interrelacionados que utilizan los sistemas de aplicación de empresas, los datos se almacenan en conjuntos independientes y sin redundancia

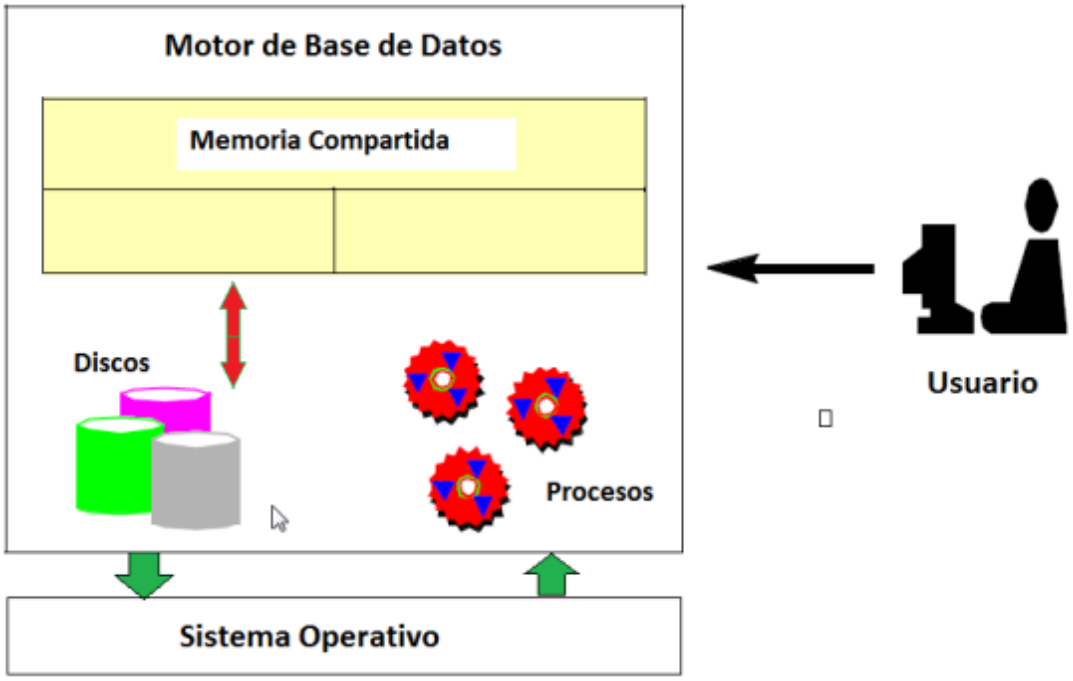
Un sistema de administración de base de datos (DBMS = Data Base Manager System) es un programa que permite administrar todos los contenidos de una BD almacenada en disco, maneja las operaciones de usuario en el nivel más alto e interpreta y ejecuta todas las instrucciones SQL

Ejemplos de DBMS son Microsoft SQL Server, Oracle, MySQL, PostgreSQL

Componentes de una base de datos

1. **Datos:** Los datos pueden ser integrados (minimiza la redundancia) o compartidos (pueden ser accedidos de forma concurrente)
2. **Programas – Procesos:** El DBMS es el programa más importante, los procesos controlan las actividades del DBMS y abstraen al usuario de los detalles a nivel de HW
3. **Usuarios**: Hacen referencia a los programadores de la aplicación, al DBA (Administrador de BD), a los usuarios finales y a otros grupos funcionales
4. **Tecnología:** Hacen referencia al Hardware (Memoria) y el Software (Sistema operativo)

Los componentes de un DBMS son los siguientes



- La memoria compartida se usa para cachear los datos de forma de acceder más rápido o para guardar información necesaria que utilizarán los procesos
- Los procesos controlan las actividades del DBMS y proveen funciones específicas relacionadas con la seguridad, integridad, etc.

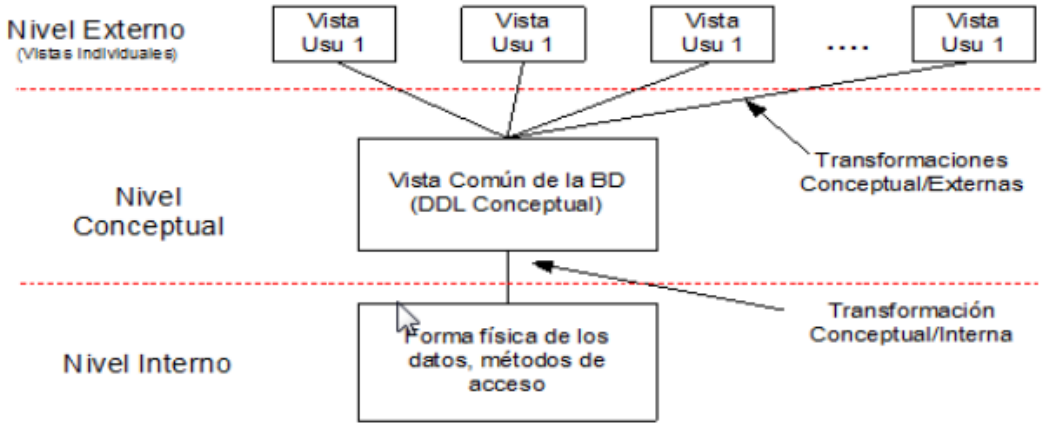
- Los discos guardan la información de la BD y toda la información propia del sistema para mantener la DBMS

A la hora de realizar consultas desde las aplicaciones de usuario las peticiones las atiende el DBMS a través de instrucciones SQL, es decir que no lo hace el SO.

Ventajas de este enfoque

1. Concurrencia en acceso a datos
2. Consistencia de datos (transacciones)
3. Seguridad
4. Integridad (constraints, vistas, stored procedures, triggers)
5. Independencia
6. Redundancia mínima
7. Criterios y normativas para la organización de datos
8. Estándares de documentación y normalización de las nomenclaturas

La arquitectura de una BD puede decirse que se ajusta a:



1. **Nivel externo o VISTAS**: Esta es la percepción que tienen los usuarios respecto a la BD por ejemplo los DBA, programadores, usuarios finales, para el usuario la vista es la base de datos
2. **Nivel conceptual o LOGICO:** Representa el esquema conceptual de toda la información contenida en una BD. Contiene definiciones del contenido de la BD, restricciones, reglas de integridad, tipos de datos

3. **Nivel interno o FISICO:** Se define como se almacenan los datos en disco, es una representación a bajo nivel de la BD, por ejemplo se especifica las estructuras que se usan en disco o memoria, se especifica la forma de acceder a los datos
4. **Transformación Externa – Conceptual:** Define la relación entre una vista externa y la vista conceptual
5. **Transformación Conceptual – Interna:** Define la relación entre la vista conceptual y la BD almacenada

Las funciones del motor de base de datos (DBMS) pueden ser

Diccionario de datos: Es un conjunto de tablas propias de la BD del sistema (metadatos) y no pueden ser alteradas por ningún usuario, por ejemplo si se define una tabla en el diccionario se guardan los parámetros, restricciones y relaciones de la misma. Las sentencias SQL relacionadas con el diccionario son CREATE, ALTER, DROP <Objeto de la BD>

Control de seguridad: La seguridad implica que los usuarios tengan autorización para realizar una determinada acción para esto el sistema de BD utiliza un catálogo donde figuran entidades e interrelaciones. Las sentencias SQL relacionadas con esto son GRANT (otorga permisos) y REVOKE (quita permisos) y alguno objetos de seguridad son (vistas, SP, triggers)

Integridad de los datos: El motor de BD cuenta con mecanismos para garantizar la integridad de los datos los cuales pueden ser creados y el motor se encargará de que se cumplan su función. Por ejemplo existen las restricciones o constraints (DEFAULT, CHECK, NOT NULL, UNIQUE, PRIMARY KEY, FOREIGN KEY), también los objetos de la BD se encargan de esto (vistas, SP, triggers, indices)

Consistencia de los datos: Para garantizar la consistencia de los datos el DBMS utiliza transacciones, logs transaccionales y métodos de recuperación ante caídas

Las transacciones son instrucciones SQL que se ejecutan de manera atómica es decir se ejecutan completamente o no se ejecutan por lo tanto una transacción siempre debe llevar a un estado correcto de la BD ya sea que falle o no. Las instrucciones de

INSERT, UPDATE y DELETE son transacciones. Para utilizar estas se utiliza el comando BEGIN TRAN que indica el comienzo de una transacción, para actualizar los datos se utiliza COMMIT TRAN, en caso de que haya una falla para revertir se usa ROLLBACK TRAN y si se quiere revertir desde un determinado punto se usa SAVE TRAN todos los comandos al terminar de ejecutarse dejan a la BD en un estado correcto o consistente

Los logs transaccionales sirven para guardar toda la información relacionada con las operaciones que se realizaron sobre los datos

Los recovery son un mecanismos que se ejecutan automáticamente al inicio de la BD como método de tolerancia a fallas en caso de caídas y sea necesaria recuperar la información sus objetivos son:

1. Retornar la BD a un punto consistente donde la memoria este sincronizada con el disco.
2. Utilizar logs transaccionales para retornar la BD a un estado lógico consistente verificando las operaciones que se hayan realizado con éxito (rolling foward) y deshacer las que no hayan tenido éxito (rolling back)

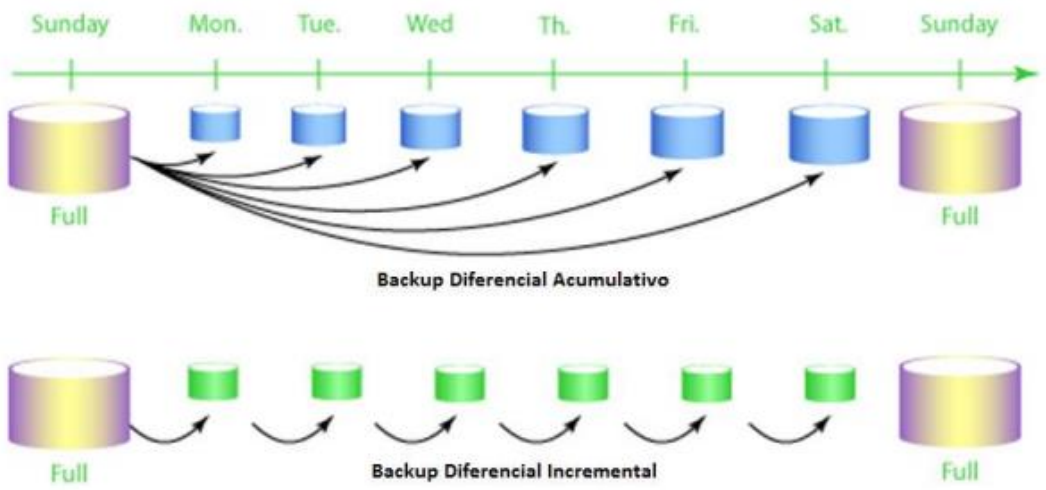
Propiedades de un DBMS: Se hace foco en la ejecución de transacciones y se busca que cumplan con ACID

1. A = Atomicidad: las transacciones se realizan completamente o no se realizan
2. C = Consistencia: Los datos se actualizan o no de forma correcta, es decir no se romperá la integridad de los datos
3. I = Isolation o Aislamiento: Una transacción no puede verse afectada por otras transacciones concurrentes.
4. D = Durabilidad: Una operación que se realiza correctamente se persistirá y no podrá deshacerse aunque falle el sistema

Las BD no relacionales no cumplen con consistencia

Backup y Restore: Backup es copiar total o parcialmente la información de una BD y almacenarla en otro sistema de almacenamiento masivo. Los tipos de backups son

1. **Backup completo o full:** Se guarda toda la información que se actualizo
2. **Backup incremental diferencial:** Se guarda la información que se actualizo a partir de una fecha tomando como punto de partida (delta T) la fecha del ultimo backup incremental
3. **Backup incremental acumulativo:** Se guarda la información que se actualizo a partir de una fecha tomando como punto de partida (delta T) la fecha del ultimo backup full
4. Backup en caliente: Se realiza cuando la aplicación está funcionando
5. Backup en frio: Se realiza cuando la aplicación no esta en uso
6. Backup de logs transaccionales: Se realiza sobre logs transaccionales



Restore es restaurar la información a partir de un backup realizado sobre una BD

Control de concurrencia: Los motores controlan el acceso concurrente a los datos a través de lockeos y niveles de aislamiento

Tipos de lockeos

1. Compartido: Si una transacción realiza un cambio sobre un recurso que tiene lockeos compartidos y está siendo utilizado por otras transacciones está se bloquea hasta que las demás transacciones liberen el bloqueo, no durán durante toda la transacción.

2. Exclusivo: Una transacción solo podrá usar el recurso en un determinado tiempo, durán hasta que finaliza la transacción
3. Promovido

Granularidad de lockeos

1. Lockeos a nivel de BD
2. Lockeos a nivel tabla
3. Lockeos a nivel pagina
4. Lockeos a nivel fila
5. Lockeos a nivel indice

Niveles de aislamiento

1. **Read uncommited:** Leer datos que no fueron confirmados, no aseguran lockeos por select por lo que se producirán lecturas sucias, no repetibles y lecturas fantasmas
2. **Read commited:** Leer solo datos que fueron confirmados (por defecto en varios motores de BD) evita que se produzcan lecturas sucias pero no garantiza lecturas repetibles
3. **Repeatable read:** Asegura lecturas repetidas y evita lecturas sucias pero aun así pueden aparecer lecturas fantasmas en caso de inserción de nuevos registros. Su uso es muy particular.
4. **Serializable read:** Asegura lecturas repetibles y evita lecturas sucias y lecturas fatansma, la desventaja es que aplica un bloqueo que puede afectar a los usuarios en los sistemas multiusuarios, realiza un bloqueo de un rango de índice según donde se ponga el where y si no es posible bloquea toda la tabla. Puede producir interbloqueos.

Para especificar un nivel de aislamiento se usa

SET TRANSACTION ISOLATION LEVEL <TIPO ISOLATION>

Otras funciones de la DBMS

- Facilidad de auditoria: Se pueden crear logs donde figuren el uso de los recursos de la BD para auditar posteriormente
- Logs del sistema: Se utilizan para determinar la falla que produjo una caída del sistema

- Modificaciones o cambios en el esquema: El motor permite los cambios en las tablas constantemente
- Creacion de objetos de BD
- Mirroring de discos
- Encriptacion de discos
- Replicacion de discos

UNIDAD V: DATAWAREHOUSE

DATAWAREHOUSE

Es una colección de datos de las cuales se extrae información para la toma de decisiones, los datos pueden provenir de diferentes partes del sistema, incluso sistemas externos.

Características de un Datawarehouse

1. **Soporta el proceso de toma de decisiones:** Es la principal función de un DW.
2. **Orientada a sujetos**: El DW está orientado a los mayores sujetos de la empresa para que tomen las decisiones más convenientes.
3. **Integrada:** Como los datos provienen de diferentes sistemas que pueden utilizar distintas BD, se integra los datos para que los mismos sean consistentes.
4. **Variante en el tiempo:** Los datos pueden ser útiles en determinados momentos, no necesariamente en la actualidad.
5. Simple de utilizar: Solo se realizan dos operaciones carga inicial y consultas ya que no existen modificación de datos.
6. **No volátil:** Los datos utilizados para la toma de decisiones deben ser estables por lo tanto no deben modificarse una vez almacenados.

Datamart

Es un componente de un DW que se centra en un área específica de la empresa. Son menos costosos pero tienen un alcance limitado ya que solo se enfoca en las necesidades del área en cuestión y no

en las necesidades de la organización. También se dice que el Datamart es un subconjunto de un DW.

BD Multidimensional

Es una BD en donde los datos están asociados a dimensiones, es decir su valor varía según los “ejes” definidos.

1. **Hipercubo:** Se guardan todas las dimensiones en un cubo, soporta la dispersión de los datos (no todas las celdas del cubo están completas y los datos están alejados entre sí), cada intersección del cubo es otro cubo.
2. **Multicubo:** Implementa el modelo de forma dinámica ya que solo tiene punteros a las caras que existen, es decir se “saltean” las caras inexistentes. Es recomendable cuando hay una alta dispersión de datos.