Con mis palabras

***Que es la Recursividad de datos**

Es el modo o forma que tiene una estructura de datos de ir almacenando y de que en algunas de estas se pueda recorrer esos datos. Se dice que la recursividad no es una estructura, como los de búsqueda. Es la propiedad mediante la cual un subprograma puede llamarse a si mismo en las estructuras de datos, en estas invocaciones se van almacenando, creando o administrando los datos.

***¿Porque se considera a una tabla hash más rápida, en muchas de las operaciones que realiza (guardar, recorrer, etc.) a una TAD lista simple?**

A la tabla hash se la considera más rápida en operaciones de almacenar, recorrer y búsqueda porque ubica el dato en un lugar indicado con índice o clave generada.

La tabla hash permite el acceso al elemento como por ejemplo a DNI, direcciones, teléfono, etc. Almacenados a partir de una clave generada (a través de un nombre o número de cuenta, etc.). Esto permite el acceso al elemento más rápido, ya que fue identificado gracias a la clave.

Esta funciona transformando la clave con una función hash, en un número que identifica la posición donde la tabla hash localiza el valor deseado.

Para poder recorrer, guardar o eliminar elemento de una lista simple, se realiza siempre desde el principio y utilizando algoritmos de búsqueda hasta hallar elemento deseado. Llegando a veces a recorrer todos los datos sin que el elemento sea encontrado o que se encuentre en la última posición de la lista simple o que no se conozca su posición.

Explica con sus palabras: ¿Qué se entiende por Back-Tracking?¿en qué situación se lo utiliza? ¿se podría reemplazar por otro mecanismo heurística o estructura?

Back-Tracking es un proceso que utiliza la recursividad para la búsqueda exhaustiva de todas las combinaciones posibles. Al poseer estructuras abstractas de datos, se necesita encontrar la forma de poder recorrer sus elementos de manera siempre optima, evitando los bucles infinitos o bien estar recorriendo elementos varias veces innecesariamente. Al Back-Tracking se lo conoce como un proceso en el que “volvemos sobre nuestros pasos” o ”hacia atrás” el ejemplo del laberinto es la mejor forma de entender este proceso, ya que, para poder encontrar la salida debemos recordar el y los caminos que aún no fueron “visitados” para poder al regreso ingresar a él. Este ejemplo es similar a las estructuras de grafos, al no tener este un nodo raíz y poder acceder a el desde cualquier nodo, es necesario utilizar el proceso de Back- Tracking para conocer que nodos ya fueron recorridos y cuales quedan por recorrer.

El back- Tracking se utiliza para recorrer estructuras abstractas como árboles y grafos. También en algoritmos que requieren la búsqueda de una solución, la cual requiere que se vaya evaluando varias veces una posible solución hasta hallar la correcta o deseada.

Otro mecanismo, heurística o estructura no realizarían el mismo trabajo.

Recursividad:

La recursividad es un proceso en el que se calcula un problema reiterada veces a si mismo hasta resolverlo.

La recursividad en programación es una caracterisitca que permite que un subprograma o un encontrar una solución, la recursividad es útil a la hora de resolver problemas definibles en sus propios términos.

La recursividad se la puede clasificar de dos formas; nivel de funciones o procedimiendos, la cueal puede ser

Directa
: una función o método se llama a si mismo una o varias veces ej serie Fibonacci

Indirecta: una función o método “A” llama a otro “B” y esta a su vez llama a “A” ej:

Números positivos y negativos

También esta a nivel de datos, que es la definición recursiva

La características

1. Es una llamada o invocaciona a si misma
2. Tiene una condición de fin
3. Se utiliza usalmente para reemplazar a una estructura repetitiva
4. Para utilizarla se realizan pocos pasos
5. Una de sus desventajas es que es un proceso lento, ya que al apulas los datos en memoria, luego es necesario desapilarlos
6. Se requiere mucha memoria para almacenar datos en una pila

La diferencia entre la recurividad y la iteración es que:

En la recursividad, la repetición de la tareas se controlan desde adentro, mientras que en la iteración desde afuera

Tipos abstractos de datos:

Abstracción la abstracción consiste en ocultar las características de un objeto y obviarlas. De manera que solamente utilizamos el nombre del objeto en nuestro programa

La abstracción funcional consiste en crear procedimientos y funciones e innovarlos mediante un nombre donde se destaca que hace la función y se ignora como lo hace

Abstracción de datos, tipos de datos: proporcionado por los lenguajes de alto nivel. La representación usada es invisible al programador, al cual solo se le permite ver las operaciones predefinidas para cada tipo

Tipos definidos, están definidos por el programador lo cual posibilitan la definición de valores de datos mas cercanos al problema que se prende resolver

TIPOS DE DATOS VS OOP

Opp como contenedores: en general no se sabe cuantos objetos se van a necesitar para resolver un problemas en particular, cuanto duraran o como almacenar esos objetos

¿Y como puede saber cuanto espacio crear si sa información no se conoce hasta el tiempo de la ejecución? Los arreglos y matrices triene este problema y la solución a la mayoría de estos problemas es creando otro objeto contenedor que tiene referencia a objetos internos

Este nuevo tipo de objeto se encarga de acomodar o ordenar todos los objetos que se coloquen dentro de el, sin saber cuantos serán. Simplemente crea un objeto contenedor y deja que se encargue de los detalles

Los contenedores proporcionan diferentes tipos de interfaces y comportamientos externos, además de permitir incluir operaciones que tienen diferentes eficiencias para ciertas operaciones.

TAD

Como en el cas ode los arreglos, en este también se pueden crear nuevos tipos de datos con capacidades diferentes, que posean operaciones especiales que oculten la implementación de las mismas a los usuarios que lo quieren utilizar.

Un tip ode datos abstracto es un conjunto de datos u objetos al cual se le asocian operaciones

El TDA provee de una interfaz con la cual es posible realizar las operaciones pemitidades abstrayéndose de la manera en como están implementadas dichas operaciones.

Los obejtos permiten encapsular valores, métodos y funciones que pueden ser usados por otras partes de un sistema mediante una clara definición o interfaz que define a los mismos

Ejemplo de tad

Se define un nuevo tipo de dato que posee las mismas capacidades que un arreglo, pero que posee la capacidad crecer dinámicamente a medida que nuevos calores sean asignados

El nuevo tipo contendrá operaciones para agregar, actualizar y eliminar valores asi también como operaciones para recorrerlo y retonar la longitud del mismo

Mitad mitad = nes mitad=;

Mitad,agregar(5);

Mitad.legth();

Mitad.recorrer();

Estructuras de datos lineales

Estas pueden ser listas que pueden ser simples, doblemente enlazadas y circulares y estas ultimas pueden ser simples y doblemente enlazadas

Y también están las pilas y las colas

Listas:

Las Listas: son una lista enlazada de nodos, donde cada nodo tiene un único campo de de enlace. Una variable de referencia contiene una referencia aal primero nodo, a cada nodo(excepto al ultimo) enlaza con el nodo siguiente, y el enlace del ultimo nodo contiene null para indicar el final de la lista.

```
record Node {
   data // El dato almacenado en el nodo
   next // Una referencia al nodo siguiente, nulo para el último nodo
}

record List {
    Node PrimerNodo   // Apunta al primer nodo de la lista; nulo para la lista vacía
}

node := list.PrimerNodo
while node not null {
    node := node.next
}
```

Listas doblemente enlazadas: es un tipo de lista que contiene dos vias. Cada nodo tiene dos enlaces, uno que apunta al anterior o apunta al valor si es el primer nodo y otro que apunta al nodo siguiente o apunta al valor null si es el ultimo nodo

```
record Node {
   data // El dato almacenado en el nodo
   next // Una referencia al nodo siguiente, nulo para el último nodo
   prev // Una referencia al nodo anterior, nulo para el primer nodo
}

record List {
    Node firstNode   // apunta al primer nodo de la lista; nulo para la lista vacía
    Node lastNode    // apunta al último nodo de la lista; nulo para la lista vacía
}

node := list.firstNode
while node ≠ null
    <do something with node.data>
    node := node.next
```

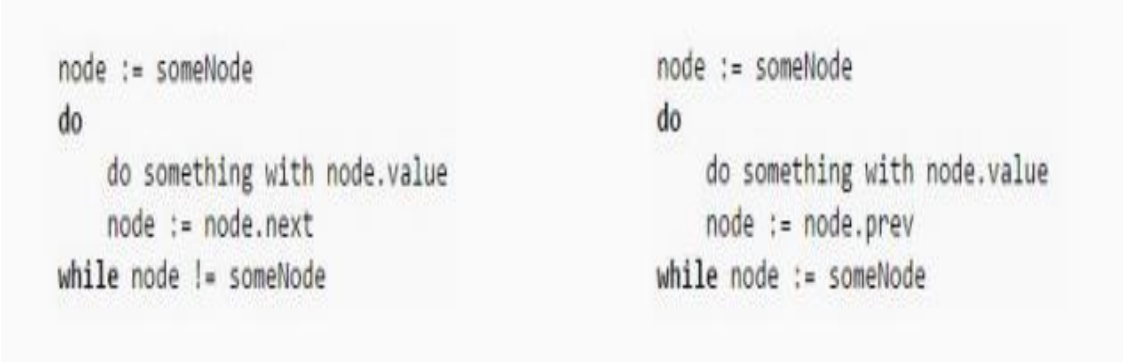
```
node := list.lastNode
while node ≠ null
    <do something with node.data>
    node := node.prev
```

Listas enlazadas circulares:

En las listas circulares, el primero y el ultimo nodo están unidos juntos, esto se puede hacer tanto para las listas simples como para las listas doblemente enlazadas, para recorrer una lista enlazada circular podemos empezar por cualquier nodo y seguir la lista en cualquier dirección hasta que se regrese al nodo original, este tipo de lista es la mas usada para visitar todos los nodos de una lista a partir de un nodo dado

Listas circulares simples

Cada nodo tiene un enlace, similar al de las listas enlazadas simples excepto que el siguiente nodo del ultimo apunta al primero, como en una lista simple, los nuevos nodos pueden ser insetados después de uno que ya tenemos refenciado. Es usual quedarse con una referencia solamente al ultimo elemento en una lista enlazada circular simple, esto nos permite rápidas inserciones al principio y también permite accesos al primer nodo desde el puntero del ultimo nodo



```
node := someNode
do
    do something with node.value
    node := node.next
while node != someNode
```

```
node := someNode
do
    do something with node.value
    node := node.prev
while node := someNode
```

Listas doblemente circulares

En una lista doblemente circularm cada bidi tuebe dis ebkaces similares a los de la lista doble, excepto que el enlace anterior del primer nodo apunta al ultimo y el enlace siguiente apunta al primero. Como en una lista doble, las inseciones y eliminaciones pueden ser hechas desde cualquier punto con acceso a algún nodo cercano. Un puntero de acceso externo puede establecer el nodo apuntando que esta en la cabeza o al nodo cola y asi mantener el orden tan bien como en una lista doble

Pilas

Estructura de datos lineal donde los elementos pueden ser añadidos o removiudos solo por un extremo, a esta estructura se la conoce como lifo -> last in – first out

Operaciones

● Push (insertar/apilar): Agrega elementos a la pila en el extremo Tope.

● Pop (remover/desapilar): Remueve el elemento de la pila que se encuentra en el extremo llamado Tope.

● Vacía: Indica si la pila contiene o no elementos.

● Llena: Indica si es posible o no agregar nuevos elementos a la pila.

Colas

Es una lista lineal de elementos en la que las operaciones de insertar y eliminar se realizan en diferentes extremos de la cola, y esta también se la conoce como FIFO -> First In - First Out -> el primer elemento en entrar es el primer elemento en salir.

Operaciones ● Enqueue (insertar): Almacena al final de la cola el elemento que se recibe como parámetro. ● Dequeue (eliminar): Saca de la cola el elemento que se encuentra al frente. ● Vacía: Indica si la cola contiene o no elementos. ● Llena: Indica si es posible o no agregar nuevos elementos a la cola

Estructura de datos no lineales

Arboles pueden ser binarios o multi caminos o narios

Y grafos dirigidos y no dirigidos, y después están conexos y no conexos

Arboles:

Un árbol es una estructura de datos formada por un conjunto de nodos conectados, el nodo es una unidad sobre la que se construye el árbol y puede tener cero o mas nodos hijos conectados a el. Se dice que un nodo A es padre de un nodo b si existe un enlace desde a hasta b (en ese caso, también decimos que b es hijo de a)

Árbol binario

Es una estructura de datos en la cual cada nodo puede tener un hijo izquiero y un hijo derecho. No pueden tener mas de dos hijos

Árbol N-ario

Posee un grado g mayor a dos, donde cada nodo de información del árbol tiene un máximo de g hijos

Terminologías utilizadas en árboles

● Raíz: El nodo superior del árbol.

● Padre: Nodo con hijos.

● Hijo: Nodo descendiente de otro nodo.

● Hermanos: Nodos que comparten el

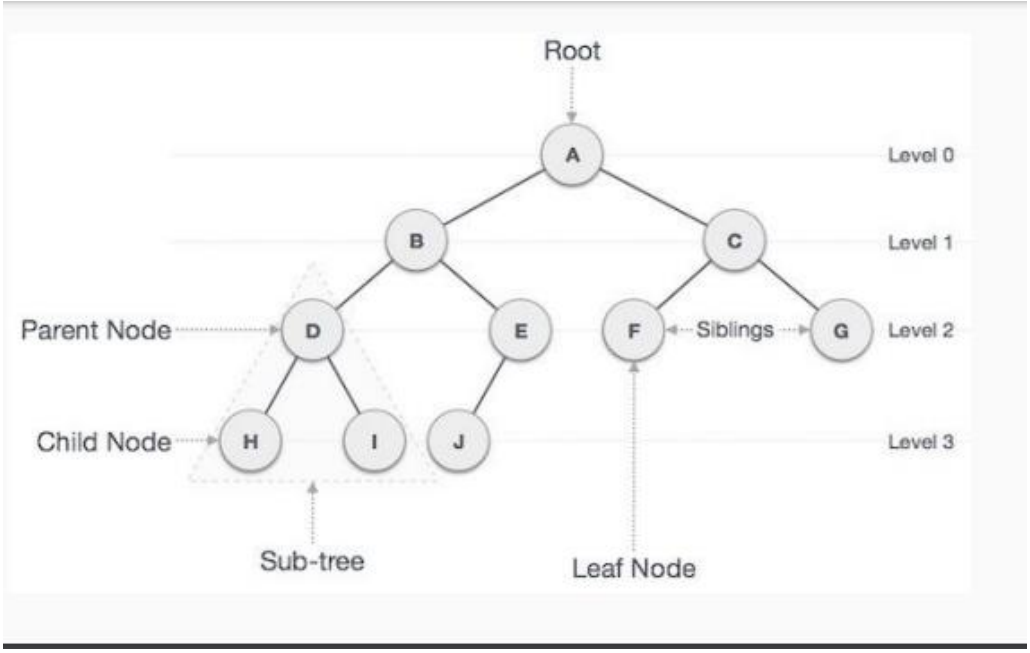
mismo padre.

● Hojas: Nodos sin hijos.

● Grado: Es el número de descendientes directos de un nodo.

● Longitud: El camino más largo de un nodo raíz hasta uno terminal.

● Profundidad: Es el número de nodos que se encuentran entre él y la raíz.



Nodo árbol binario

```
record Node {
   data // El dato almacenado en el nodo
   Node NodeIzq   // apunta al primer nodo de la izquierda.
   Node NodeDer   // apunta al último nodo de la derecha.
}
```

Nodo árbol N-ario

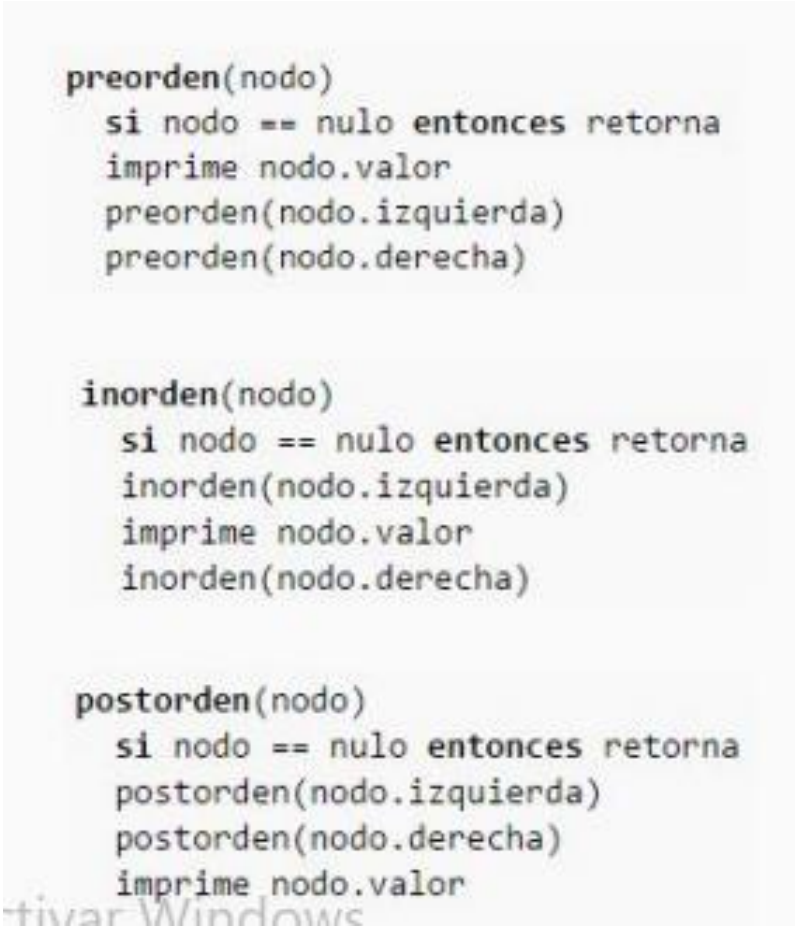
```
record Node {
   data // El dato almacenado en el nodo
   Node[...] NodeHijos   // apunta a una lista de nodos hijos.
}
```

Tipos De búsqueda de arboles:

Búsqueda en profundidad (DFS o Depth First Search): Es un algoritmo que permite recorrer todos los nodos de un grafo o árbol de manera ordenada pero no uniforme. Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodo que va localizando de forma recurrente en un camino concreto. Cuando ya no quedan mas nodos que visitar en dicjo camino, regresa (Backtracking) de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado

Búsqueda en anchura (BFS - Breadth First Search): Es un algortimo para recorrer o buscar elementos enun árbol el cual comienza en la raíz y se explora todos los hijos de este nodo. A continuación para cada uno de los hijos se exploran sus respectivos hijos y asi hasta que se recorra todo el árbol. Formalmente, bfs es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un árbol sistemáticamente para buscar una solución, el algoritmo no usa ninguna estrategia heurística

Pre-orden 1. Visite la raíz 2. Atraviese el sub-árbol izquierdo 3. Atraviese el sub-árbol derecho
In-orden 1. Atraviese el sub-árbol izquierdo 2. Visite la raíz 3. Atraviese el sub-árbol derecho
Post-orden 1. Atraviese el sub-árbol izquierdo 2. Atraviese el sub-árbol derecho 3. Visite la raíz



```
preorden(nodo)
  si nodo == nulo entonces retorna
  imprime nodo.valor
  preorden(nodo.izquierda)
  preorden(nodo.derecha)


inorden(nodo)
  si nodo == nulo entonces retorna
  inorden(nodo.izquierda)
  imprime nodo.valor
  inorden(nodo.derecha)


postorden(nodo)
  si nodo == nulo entonces retorna
  postorden(nodo.izquierda)
  postorden(nodo.derecha)
  imprime nodo.valor
```

Profundidad primero

● Secuencia de recorrido de preorden (raíz, izquierda, derecha)

F, B, A, D, C, E, G, I, H

● Secuencia de recorrido de inorden (izquierda, raíz, derecha)

A, B, C, D, E, F, G, H, I

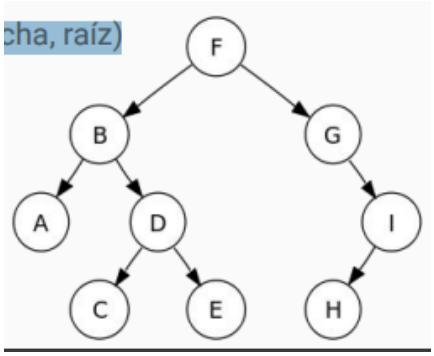
● Secuencia de recorrido de postorden (izquierda, derecha, raíz)

A, C, E, D, B, H, I, G, F

Anchura primero

● Secuencia de recorrido de orden por nivel

F, B, G, A, D, I, C, E, H



Pasos para insertar nodos

1. Se toma el dato a ingresar X

2. Partiendo de la raíz preguntamos: Nodo == null (o no existe) ?

3. En caso afirmativo X pasa a ocupar el lugar del nodo y ya hemos ingresado nuestro primer dato.

4. En caso negativo preguntamos: X < Nodo

5. En caso de ser menor pasamos al Nodo de la IZQUIERDA del que acabamos de preguntar y repetimos desde el paso 2 partiendo del Nodo al que acabamos de visitar

6. En caso de ser mayor pasamos al Nodo de la DERECHA y tal cual hicimos con el caso anterior repetimos desde el paso 2 partiendo de este nuevo Nodo.

- Insertar Nodos en Árboles Es un proceso RECURSIVO en el cual al final por más grande que sea el árbol el dato a entrar ocupará un lugar. Ejemplo: Insertar el valor 7.

Eliminar Nodos en Árboles

No es un proceso sencillo, debido a que pueden se presentan diferentes situaciones:

1. Borrar un Nodo sin hijos

2. Borrar un Nodo con un subárbol hijo

3. Borrar un Nodo con dos subárboles hijos

Pasos para Borrar un Nodo sin hijos

El caso más sencillo, lo único que hay que hacer es borrar el nodo y establecer el apuntador de su padre a nulo.

Ejemplo: Eliminar el valor 7

Pasos para Borrar un Nodo con un subárbol hijo

Este caso tampoco es muy complicado, únicamente tenemos que borrar el Nodo y el

subárbol que tenía pasa a ocupar su lugar.

Ejemplo: Eliminar el valor 5

Pasos para Borrar un Nodo con un subárbol hijo

Este caso tampoco es muy complicado, únicamente tenemos que borrar el Nodo y el subárbol que tenía pasa a ocupar su lugar.

Ejemplo: Eliminar el valor 5

Pasos para Borrar un Nodo con dos subárboles hijos

Este es un caso algo complejo:

1. Tenemos que tomar el hijo derecho del Nodo que queremos eliminar y recorrer hasta el hijo más a la izquierda (hijo izquierdo y si este tiene hijo izquierdo repetir hasta llegar al último nodo a la izquierda)
2. Reemplazamos el valor del nodo que queremos eliminar por el nodo que encontramos (el hijo más a la izquierda), el nodo que encontramos por ser el más a la izquierda es imposible que tenga nodos a su izquierda pero sí que es posible que tenga un subárbol a la derecha
3. Para terminar solo nos queda proceder a eliminar este nodo de las formas que conocemos (caso 1, caso 2) y tendremos la eliminación completa.

Grafos:

Un grafo es una representación de un grupo de objetos conectados donde algunos pares están conectados por aristas. Es un conjunto de objetos llamados vertices o nodos unidos por enlaces llamados aristas o arcos, que permiten representar relaciones binarias entre elementos de un conjunto

Un grafo {G} es un par ordenado {G=(V,E)}, donde:

- {V} es un conjunto de vértices o nodos, y

● {E} es un conjunto de aristas o arcos, que relacionan estos nodos.

● {E} se encuentra incluido en V x V.

● Normalmente {V} suele ser finito.

Ejemplo

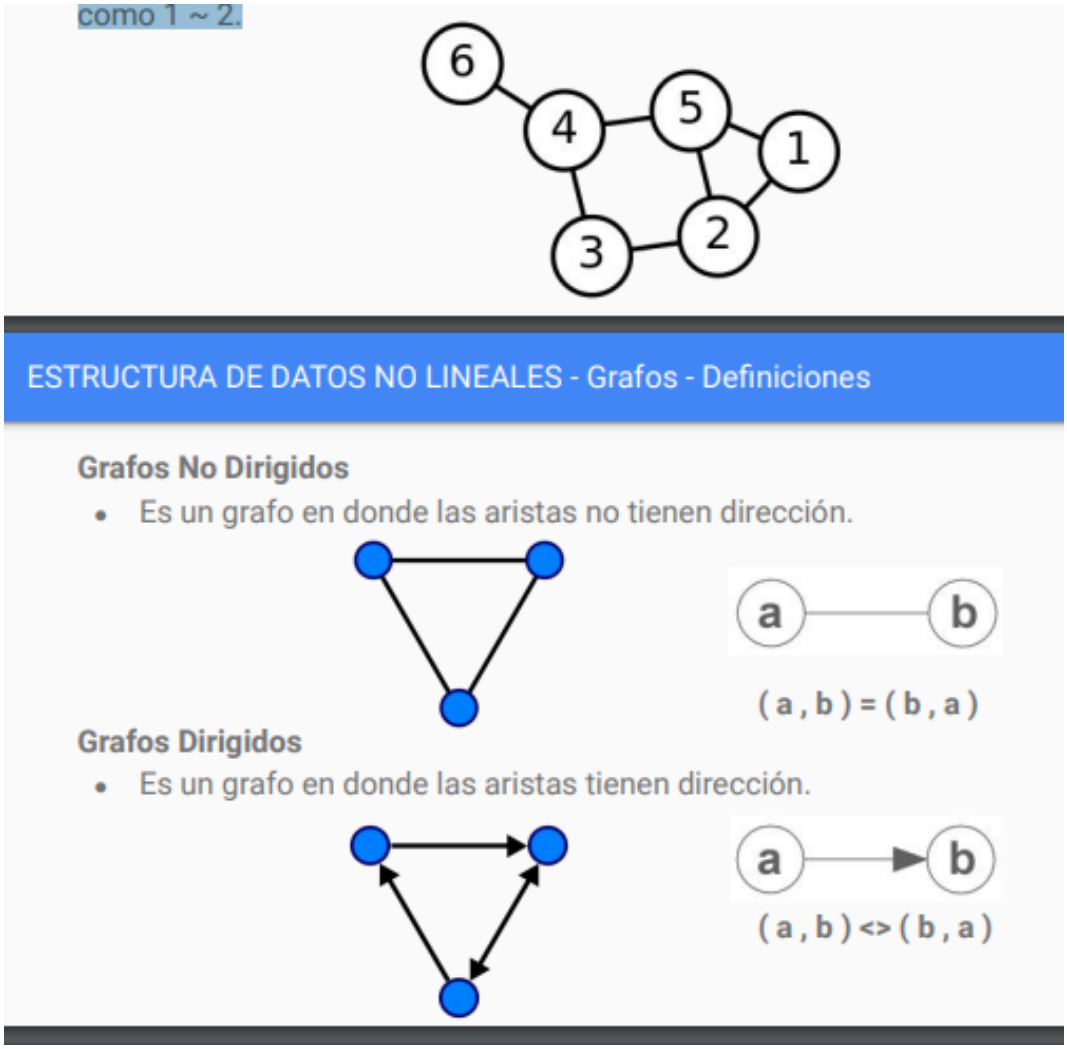
La imagen es una representación del siguiente grafo:

● V:={1,2,3,4,5,6}

● E:={{1,2},{1,5},{2,3},{2,5},{3,4},{4,5},{4,6}}

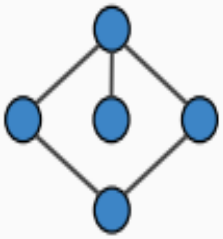
El hecho que el vértice 1 sea adyacente con el vértice 2 puede ser denotado

como 1 ~ 2.



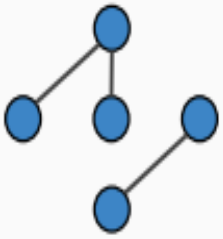
Grafos Conexos

- Es un grafo en donde cada par de vértices están conectados, existiendo una conexión entre todos ellos.



Grafos No Conexos

- Es un grafo en donde existen al menos dos subgrafos conexos.



Terminologías utilizadas en Grafos

Para un grafo G = {(V, E)}

● Orden: Se llama orden del grafo G a su número de vértices V.

● Grado: El grado de un vértice o nodo V es igual al número de arcos que lo tienen como extremo.

● Nodos adyacentes: Vértices conectados por una arista.

● Aristas/Arcos adyacentes: Aristas/Arcos con un nodo en común.

● Bucle: es una arista que relaciona al mismo nodo; es decir, una arista donde el nodo inicial y el nodo final coinciden.

● Camino: Sucesión de arcos adyacentes tal que el vértice final de cada arco coincide con el inicial del siguiente.

● Circuito: Camino que empieza y acaba en el mismo vérticeTipos de Grafos

● Grafo Etiquetado: Cada arista y/o vértice tiene asociada una etiqueta o valor.

● Grafo Ponderado o Grafo con pesos: Grafo etiquetado en el que existe un valor numérico asociado a cada arista o arco.

● Multigrafo: Grafo en el que se permite que entre dos vértices exista más de una arista o arco.

● Árbol: Grafo conexo que no tiene ciclos.

Como representar un grafo

Pares de adyacencia una lista de pares donde cada par representa una arista, se representa usando un arreglo donde cada posición corresponde a una arista que une un par de vertices

Ventajas : es simple de parsear

Desventajas: hacer cualquier operación es difícil y lento, por lo general solo se usan para la entrada en los problemas de grafos

Matrices de Adyacencia

● Una matriz donde cada el elemento en (i; j) indica

si hay una arista entre el nodo i y el nodo j.

● Se representa utilizando una matriz cuadrada

● de V x V.

● Se necesita conocer la cantidad de vértices, luego

se completa la matriz.

● Es ideal para cuando el número de vértices |V| es

pequeño y hay que encontrar relaciones entre

pares de nodos muy seguido, o cuando el grafo es

denso.

Ventajas:

● Saber si dos nodos están conectados tiene complejidad

simple, una lectura en la matriz.

● Se pueden hacer operaciones sobre esta matriz para

encontrar propiedades del grafo.

● Una matriz donde cada el elemento en (i; j) indica

si hay una arista entre el nodo i y el nodo j.

Desventajas:

● Buscar los nodos adyacentes a otro nodo tiene

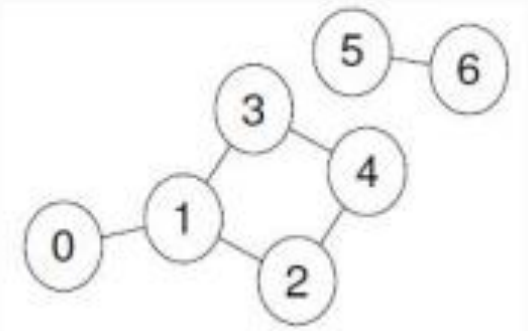
mayor complejidad, sin importar cuántos nodos

adyacentes tenga el primer nodo.

● Recorrer todo el grafo implica gran complejidad.

● Si la matriz es grande, ocupa mayor memoria.

	0	1	2	3	4	5	6
0	0	1	0	0	0	0	0
1	1	0	1	1	0	0	0
2	0	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	0	0
5	0	0	0	0	0	0	1
6	0	0	0	0	0	1	0



Listas de Adyacencia

● Una lista con |V| elementos, donde el elemento número i tiene la lista de nodos adyacentes al nodo i.

● Para cada nodo de |V| se tiene una lista de aristas que parten de ese nodo.

● Las listas están guardadas en un array de nodos cabecera.

● Si el grafo no es dirigido entonces cada arista será representada dos veces.

● Por lo general se usa en grafos esparsos, cuando |V| es grande, o cuando los multi-ejes son significativos.

Listas de Adyacencia

● Una lista con |V| elementos, donde el elemento número i tiene la lista de nodos adyacentes al nodo i.

● Para cada nodo de |V| se tiene una lista de aristas que parten de ese nodo.

● Las listas están guardadas en un array de nodos cabecera.

● Si el grafo no es dirigido entonces cada arista será representada dos veces.

● Por lo general se usa en grafos esparsos, cuando |V| es grande, o cuando los multi-ejes son significativos.

Búsqueda en profundidad (DFS o Depth First Search)

● Es un algoritmo que permite recorrer todos los nodos de un grafo o árbol de manera ordenada, pero no uniforme.

● Su funcionamiento consiste en ir expandiendo todos y cada uno de los nodos que va localizando, de forma recurrente, en un camino concreto.

● Cuando ya no quedan más nodos que visitar en dicho camino, regresa (Backtracking), de modo que repite el mismo proceso con cada uno de los hermanos del nodo ya procesado.

```
DFS(grafo G)
   PARA CADA vértice u ∈ V[G] HACER
           estado[u] ← NO_VISITADO
           padre[u] ← NULO
   tiempo ← 0
   PARA CADA vértice u ∈ V[G] HACER
           SI estado[u] = NO_VISITADO ENTONCES
                   DFS_Visitar(u,tiempo)

 DFS_Visitar(nodo u, int tiempo)
    estado[u] ← VISITADO
    tiempo ← tiempo + 1
    d[u] ← tiempo
    PARA CADA v ∈ Vecinos[u] HACER
            SI estado[v] = NO_VISITADO ENTONCES
                    padre[v] ← u
                    DFS_Visitar(v,tiempo)
    estado[u] ← TERMINADO
    tiempo ← tiempo + 1
    f[u] ← tiempo
```

Búsqueda en anchura (BFS - Breadth First Search)

● Es un algoritmo para recorrer o buscar elementos en un grafo el cual comienza eligiendo algún nodo como elemento raíz y se exploran todos los vecinos de este nodo.

● A continuación para cada uno de los vecinos se exploran sus respectivos vecinos adyacentes, y así hasta que se recorra todo el grafo.

● Formalmente, BFS es un algoritmo de búsqueda sin información, que expande y examina todos los nodos de un grafo sistemáticamente para buscar una solución.

● El algoritmo no usa ninguna estrategia heurística.

```
BFS(grafo G, nodo_fuente s)
{
   // recorremos todos los vértices del grafo inicializándolos a NO_VISITADO,
   // distancia INFINITA y padre de cada nodo NULL
   for u ∈ V[G] do
   {
      estado[u] = NO_VISITADO;
      distancia[u] = INFINITO; /* distancia infinita si el nodo no es alcanzable */
      padre[u] = NULL;
   }
   estado[s] = VISITADO;
   distancia[s] = 0;
   padre[s] = NULL;
   CrearCola(Q); /* nos aseguramos que la cola está vacía */
   Encolar(Q, s);
   while !vacia(Q) do
   {
      // extraemos el nodo u de la cola Q y exploramos todos sus nodos adyacentes
      u = extraer(Q);
      for  v ∈ adyacencia[u]  do
      {
         if estado[v] == NO_VISITADO then
         {
              estado[v] = VISITADO;
              distancia[v] = distancia[u] + 1;
              padre[v] = u;
              Encolar(Q, v);
         }
      }
   }
}
```

Matriz de Adyacencia

● Será necesario inicializar la matriz con valores

● Que indiquen que la misma está vacía (0 o false)

● A medida que se indican los pares de vértices adyacentes, se procederá a completar en la matriz, en la fila y columna

correspondiente, que existe una arista entre el par de vértices, por ejemplo con un 1 o true.

● En el caso de que el Grafo posea pesos en sus aristas, se podrá insertar dicho valor de forma similar al punto anterior.

● Para eliminar conexiones entre vértices, sólo implica volver a setear el valor en la matriz con 0 o false.

Listas de Adyacencia

● Será necesario inicializar un arreglo de Listas de Nodos (TAD Lista)

● El arreglo no tendrá valores iniciales (las Listas están vacías)

● Cada posición del arreglo corresponderá a un vértice del grafo, por lo tanto será necesario crear una función de mapeo entre los índices del arreglo y los vértices del grafo. (arreglo[1] => Nodo 1, arreglo[2] => Nodo 2, etc)

● A medida que se indican los pares de vértices adyacentes, se procederá a buscar el índice del arreglo que corresponde con el vértice inicial del par, y luego agregar un nodo en la Lista con el valor del vértice final.

● Para eliminar conexiones entre vértices, será necesario eliminar los nodos de la Lista correspondiente al vértice inicial del par.

TABLA HASH

Una función hash es cualquier función que se puede utilizar para asignar datos de tamaño arbitrio a datos de tamaño fijo. Los valores devueltos por una función hash se denominan valores hash codigos hash, digest o simplemente hashes.

Es una estructura de datos que asocia llaves o claves con valores. Funciona trasformando la clave con una función hash en un hash, un numero que identifica la posición(casilla o campo) donde la tabla hash localiza el valor deseado. La operaicon principal que soporta de manera eficiente es la búsqueda

La inserción: para almacenar un elemento en la tabla hash se ha de convertir su clave a un número. Esto se consigue aplicando la función hash a la clave del elemento. El resultado de la función resumen ha de mapearse al espacio de direcciones del vector que se emplea como soporte. Tras este paso se obtiene un índice valido para la tabla. El elemento se almacena en la posición de la tabla obtenido en el paso anterior; si en la posición de la tabla ya había otro elemento, se ha producido una colisión. Este problema se puede solucionar siguiendo distintas técnicas.

Busqueda: para recuperar los datos, es necesario únicamente conocer la clave del elemento, a la cual se le aplica la función hash. El valor obtenido se mapea al espacio de direcciones de la tabla. Si el elemento existente en la posición indicada en el paso anterior tiene la misma clave que la empleada en la búsqueda, entonces es el deseado. Si la clave es distinta, se ha de buscar el elemento según la técnica empleada para resolver el problema de la colisiones al almacenar el elemento.

Colisiones: si dos llaves generan un hash apuntando al mismo índice, los registros correspondientes no pueden ser almacenados en la misma posición. Cuando una casilla ya esta ocupada, debemos encontrar otra ubicación donde almacenar el nuevo registro, y hacer de tal manera que podamos encontrar cuando se requiera

Direccionamiento Cerrado, Encadenamiento separado o Hashing abierto: Cada casilla en el array referencia una lista de los registros insertados que colisonan en la misma casilla. La inserción consiste en encontrar la casilla correcta y agregar al final de la lista correspondiente. El borrado consiste en buscar y quitar de la lista.

Técnicas:

● Encadenamiento separado con Listas Enlazadas: El Array de índices hash posee

un puntero a una lista enlazada que es donde se almacenan los valores.

● Encadenamiento separado por Cabecera de Celdas: Almacena el valor del primer

registro directamente junto a la clave Hash y luego utiliza una lista para almacenar

el resto de los valores.

● Encadenamiento separado por otras estructuras: Las listas pueden ser

reemplazadas por árboles auto-balanceables, por ejemplo, el tiempo teórico del

peor de los casos disminuye de O(n) a O(log n).

Encadenamiento separado con Listas Enlazadas: El Array de índices hash posee un puntero a una lista enlazada que es donde se almacenan los valores.

Encadenamiento separado por Cabecera de Celdas: Almacena el valor del primer registro directamente junto a la clave Hash y luego utiliza una lista para almacenar el resto de los valores.

Direccionamiento abierto o Hashing cerrado:

Las tablas hash de direccionamiento abierto pueden almacenar los registros directamente en el array. Las colisiones se resuelven mediante un sondeo del array, en el que se buscan diferentes posiciones del array (secuencia de sondeo) hasta que el registro es encontrado o se llega a una casilla vacía, indicando que no existe esa llave en la tabla.

Las secuencias de sondeo más socorridas incluyen:

● Sondeo lineal: en el que el intervalo entre cada intento es constante

(frecuentemente 1).

● Sondeo cuadrático: en el que el intervalo entre los intentos aumenta linealmente

(por lo que los índices son descritos por una función cuadrática).

● Sondeo hashe: en el que el intervalo entre intentos es constante para cada registro

pero es calculado por otra función hash.

```
registro par { llave, valor }
var vector de pares casilla[0..numcasillas-1]


function buscacasilla(llave) {
    i := hash(llave) módulo de numcasillas
    loop {
        if casilla[i] esta libre or casilla[i].llave = llave
            return i
        i := (i + 1) módulo de numcasillas
    }
}

function busqueda(llave)
    i := buscacasilla(llave)
    if casilla[i] está ocupada   // llave está en la tabla
        return casilla[i].valor
    else                    // llave no está en la tabla
        return no encontrada
```

```
function asignar(llave, valor) {
    i := buscacasilla(llave)
    if casilla[i] está ocupada
        casilla[i].valor := valor
    else {
        if tabla casi llena {
            hacer tabla más grande (nota 1)
            i := buscacasilla(llave)
        }
        casilla[i].llave := llave
        casilla[i].valor := valor
    }
}
```

COMPLEJIDAD ALGORITMICA:

La complejidad algortimica es una metrica teorica que se aplica a los algortimos como una forma de comparar una solución algorítmica con otras y de esta forma conocer como se comportara al atacar problemas complejos. La complejidad no es un numero: es una función.

● Complejidad Temporal: representa la idea del tiempo que consume un

algoritmo para resolver un problema.

● Complejidad Espacial: representa la cantidad de memoria que consume

un algoritmo para resolver un problema.

Entonces lo que vamos a medir es cómo crece el número de instrucciones

necesarias para resolver el problema con respecto al tamaño del problema.

```
//pseudocódigo
funcion ejemplo(n: entero):entero
empieza
   variables: a, j, k enteros
   para j desde 1 hasta n hacer
      a=a+j
   fin para
   para k desde n hasta 1 hacer
      a=a-1
      a=a*2
   fin para
   devolver a
termina
```

1 instrucción -> 1 ejecución.

1 bucle -> n veces cada instrucción.

1 x n + 2 x n + 1 = **3n + 1**

Asignar 1 unidad de tiempo a **n**

Algoritmo responde de manera lineal

```
//pseudocódigo
funcion ejemplo3(n:entero): entero
empieza
   variables i, j, k enteros
   para i=1 hasta n hacer
      para j=n hasta 1 hacer
         k=k+j
         k=k+i
      fin para
   fin para
    devolver k
termina
```

1 instrucción -> 1 ejecución.

1 bucle -> n veces cada instrucción.

2 x n x n + 1 = **$2n^2 + 1$**

Asignar 1 unidad de tiempo a **n**

Algoritmo responde de manera cuadrática.

```
//pseudocódigo
funcion EstaEn(v: array[1..n] de enteros, x: entero):booleano
empieza
   variables: i:entero, encontrado:booleano;
   i=1;
   encontrado=false;
   mientras(NO(encontrado) Y x<=n) hacer
      si v[i]==x entonces
        encontrado=true;
      fin si
      i=i+1
   fin mientras
   devolver encontrado
termina
```

1 instrucción -> 1 ejecución.

1 bucle -> n veces cada instrucción.

Peor caso -> se ejecuta hasta el final el bucle.

Mejor caso -> sólo se ingresa a la primera posición del array.

Orden de Complejidad: Debido a que analizar cada algoritmo de manera independiente para obtener la complejidad real sería muy complejo, lo que podemos hacer es agruparlos dependiendo de la complejidad que presenten, por ejemplo: lineal, cuadrática, cúbica, etc. ● A esta agrupación la vamos a llamar Orden de Complejidad. ● Veremos que realmente lo que influye en la complejidad de los algoritmos son los bucles y las sentencias de selección que

tienen efecto en esos bucles. ● Los algoritmos recursivos tienen un análisis de complejidad distinto y complejo.