Estructuras de Datos Básicas

Introducción

Escribí este tutorial con el objetivo de enseñar estructuras de datos a participantes de la OMI en las que hasta la fecha no he encontrado ningún
tutorial de estructuras de datos que hable con profundidad de los temas y a la vez este orientado a resolver problemas.
La mayoría de tutoriales de estructuras de datos que se encuentran estan orientados al desarrollo de aplicaciones, y los pocos que he encontrado orientados a resolver problemas de concursos de programación, no habla de los temas con la suficiente profundidad.
Otra buena razón para escribir este tutorial, es que las implementaciones que he encontrado en otros tutoriales(y en los libros tambien) de las estructuras de datos, son muy poco prácticas para concursos (aunque son lo mejor en el desarrollo de aplicaciones).
Este tutorial pretende enseñar a manejar eficazmente las estructuras de datos básicas, de las cuales se derivan muchas otras estructuras de datos mas avanzadas.
Además de que el participante las conozca y las sepa implementar, tambien es necesario que sepa hacer un uso inteligente de ellas, es por eso que este tutorial incluye una gran cantidad de problemas que involucran las estructuras de datos.
Como todo buen programador debe de saber comprobar rigursamente sus ideas o darse cuenta que está equivocado, este tutorial tambien incluye problemas de comprobación y de análisis matemático.

Requisitos para entender este tutorial

Para comprender este tutorial se requiere mas que nada:
- Conocer el lenguaje de programación C/C++.
- Conocimientos básicos de álgebra y teoría de conjuntos.
- Estar familiarizado con definiciones recursivas.
- Entender el análisis de complejidad de funciones y del tiempo de ejecución de un programa.
- Conocer y dominar las comprobaciones por medio de inducción matemática.

“modo de uso” del tutorial

Los problemas de este tutorial, como te darás cuenta, no tienen límites especificados, ni formato de entrada y salida; y no todos requieren de escribir un programa.
El hecho de que no tengan rangos especificados los problemas, no significa que se tenga que usar memoria dinámica, es decir, al resolver problemas de este tutorial, tu pon los límites del tamaño que creas conveniente, utiliza el formato de entrada que creas conveniente, y procura usar memoria estática.
Pero si debes tomar en cuenta la complejidad que especifica la descripción del problema.

El objetivo de todo esto es no perder tiempo con descripciones largas sino ir directo al problema.
Si encuentras un error, o hay algo que no se entiende y cumples con los requisitos para entender el tutorial, mandame un mensaje a luison.cpp@gmail.com

¿Qué son y para qué sirven las estructuras de datos?

En lo que se refiere a la resolución de problemas, muchas veces para plantear el problema imaginamos objetos y acciones que se relacionan entre si.
Por ejemplo, un mesero tiene platos de colores apilados; de vez en cuando el que lava los platos coloca un plato recién lavado sobre la pila de platos; y en otras ocaciones el mesero toma el plato que esta hasta arriba y sirve ahí la comida que ha sido preparada por el cocinero para posteriormente llevarla a su destino.
Si sabemos de qué color es la pila inicial de platos, y en qué momentos el que lava los platos colocó platos sobre la pila(y claro, tambien sabemos el color de estos), y en qué momentos el mesero retiró el plato que se encontraba hasta arriba; podemos saber de qué color será el plato que le toca a cada cliente.
Una manera de saberlo podría ser, hacer una representación dramatica de los hechos; pero esto no es necesario, ya que tambien podríamos tomar un lapiz y un papel, y escribir una lista de los colores de los platos, posteriormente, ir escribiendo los colores de los platos que se pusieron en la pila al final de la lista, y borrar el ultimo color de la lista cada que un plato se retire.
No se necesita ser un gran matemático para pensar en hacer eso, sin embargo, en el momento de querer implementar un programa en C que lo reprodusca, nos encontramos con que no tenemos ninguna lista donde se coloquen y se quiten cosas del final, tenemos solamente arreglos, variables, estructuras, apuntadores, etc. Claro que podemos simular esta lista con las herramientas que nos proporciona C, asi pues, los *objetos(*como la pila de platos) ligados a *operaciones*(como poner un nuevo plato o quitar un plato) que modifican al objeto son llamados estructuras de datos.
Una definición sencilla de estructura de datos: unión de un conjunto de datos y funciones que modifican dicho conjunto.
Es muy importante conocer las estructuras de datos mas comunes que se utilizan en la programación, ya que la estructura de datos es vital para plantear el problema y al resolverlo, poder implementar su solución eficazmente.

Pilas

Una pila, es la estructura de datos mencionada en el ejemplo anterior, es decir, un *altero* de objetos. O mas formalmente, una estructura de datos en la cual solo se pueden hacer 2 operaciones: colocar un elemento al final, o quitar un elemento del final.
Lo unico que se puede hacer en una pila es colocar un objeto hasta arriba, o quitar el objeto que esta arriba, ya que si se quita un objeto de abajo o del centro(lo mismo que si se intenta añadir uno), la pila colapsaría.
Si queremos programar algo similar, lo mas obvio es guardar la información de la pila en un arreglo.

Imaginemos que el restaurant tiene en total 10 platos(un restaurant bastante pobre), ello nos indicaría que un arreglo de tamaño 10 podría guardar todos los platos sin temor a que el tamaño del arreglo no alcance.
Suponiendo que inicialmente hay 2 platos, uno de color 1, y otro de color 2, el arreglo debería lucir algo asi:
2 1 0 0 0 0 0 0 0 0
Si repentinamente el que lava los platos pone un plato hasta arriba de color 2, luego de ello el arreglo debería de lucir asi:
2 1 2 0 0 0 0 0 0 0
Si luego pone hasta arriba un plato de color 3, entonces el arreglo debería de quedar asi:
2 1 2 3 0 0 0 0 0 0
Pero si el mesero toma un plato de arriba, el arreglo estará de nuevo de esta manera:
2 1 2 0 0 0 0 0 0 0
Si el mesero vuelve a tomar un plato de arriba, el arreglo quedará de esta manera:
2 1 0 0 0 0 0 0 0 0
Para lograr esto, basta con declarar un arreglo y una variable.
Tal que el arreglo diga especificamente qué platos hay en la pila, y la variable cuántos platos hay.
Entonces, podemos declarar una pila de la siguiente manera(suponiendo que la pila es de números enteros):

```
int pila[tamaño maximo];
int p=0;
```

Cada que queramos añadir un elemento en la parte superior de la pila, es suficiente con esta línea de código:
pila[p++]=*objeto*;
Y cuando queramos retirar el elemento que este en la parte superior.
pila[--p]=0;
Por último, como cultura general, en ingles a la pila se le llama stack, a la operación de poner un elemento en la parte superior se le llama push, y la de quitar un elemento se le llama pop.
Asi mismo, si la pila sobrepasa su tamaño máximo, el error devuelto es “stack overflow” o “desbordamiento de pila”(ahora ya sabemos qué quieren decir algunos errores comunes que saturan el monitor en pantallas azules).

Problema 1. Comprueba que es suficiente utilizar p-- en lugar de pila[--p]=0 para retirar un elemento de la pila.

Problema 2. En el álgebra, comunmente se utilizan signos de agrupación tales como ( ), [ ] o { }, y se pueden utilizar para indicar el orden en que se realizarán las operaciones aritmeticas. Ej. “[a(x+y)+c]d+(b(c+d))”, sin embargo, existen ciertas formas invalidas de utilizar dichos signos de agrupación Ej. “[a+b)” ó “(a” ó “(b))” o tambien “([a+b)c]”. Es decir, cada signo que “abre” debe tener un signo

correspondiente que lo "cierre", cada signo que "cierra" debe de tener un signo correspondiente qué cerrar, y los signos no se pueden traslapar.
Escribe un programa, que dada una expresión algebrarica, determine si tiene o no error en los signos de agrupación en tiempo lineal.

Problema 3. Una manera de encriptar mensajes(no muy segura), es colocar parentesis de manera arbitraria, y todo lo que esta dentro de un parentesis, ponerlo al reves, por ejemplo "Olimpiada de Informatica" se puede encriptar como "Olimpia(ad) de I(rofn)matica", los parentesis tambien se pueden anidar, es decir, otra forma de encriptar el mismo mensaje podría ser "Olimpia(am(nfor)I ed (da))tica".
Escribe un programa, que dado un mensaje encriptado, determine el mensaje original en tiempo lineal.
(9° Olimpiada Mexicana de Informática)

Problema 4. Hay *n* faros acomodados en línea recta, cada faro puede tener una altura arbitraria, y además, cada faro brilla con cierta intensidad.
La luz de cada faro ilumina unicamente a los primeros faros en cada dirección cuyas alturas sean estrictamente mayores a la del faro que emite la luz.
La iluminación total de un faro, es la suma de las luces que llegan al faro(sin incluir la propia).
Escribe un programa, que dadas las alturas de los *n* faros y las intensidades de cada uno, determine cual es el faro mas iluminado en tiempo lineal.
(USACO 2006)

Problema 5. Una cuadricula bicolorada, es aquella en la cual cada cuadro puede estar pintado de color negro o de color blanco(todos los cuadros estan pintados de alguno de estos 2 colores). Puedes asumir que cada casilla de la cuadricula tiene area 1.
Escribe un programa, que dada una cuadricula bicolorada, determine el área del mayor rectangulo (dentro de la cuadricula) pintado de un solo color. Tu programa deberá funcionar en tiempo lineal (lineal en función del número de casillas de la cuadricula, NO en función de la base o de la altura).
(Croatian Olympiad in Informatics)

Problema 6. Considera el siguiente algoritmo recursivo para generar permutaciones de *arreglo*[ ] desde *inicio* hasta *fin*:

```
1       funcion permuta(arreglo[ ], inicio, fin)
2               si inicio=fin entonces
3                       imprime arreglo[ ];
4                       fin_de_funcion;
5               para i=inicio hasta fin
6                       intercambia(arreglo[inicio], arreglo[i]);
7                       pemuta(arreglo[ ], inicio+1, fin);
8                       intercambia(arreglo[inicio], arreglo[i]);
9       fin_de_funcion;
```

Escribe una función no recursiva que genere todas las permutaciones de *arreglo*[ ] desde *inicio* hasta

fin, con la misma complejidad que la función que se muestra.

Colas

Imagina una conversación de chat entre 2 personas, aunque los conversantes no se den cuenta, existe algo llamado *lag*, es decir, el tiempo que tardan las 2 computadoras en mandarse y recibir los mensajes. Dependiendo de la conexión, el *lag* puede variar entre menos de un segundo o incluso mas de un minuto.
Si por un momento, por falla del servidor, una de las 2 computadoras pierde la conexión, y en ese momento un usuario está intentando mandar varios mensajes, el programa de chat guardará los mensajes que el usuario está tratando de mandar, y cuando se recupere la conexión, el programa de chat mandará los mensajes en el mismo orden que el usuario los escribió.
Obviamente, el programa de chat no usa una pila para eso, ya que si usara una pila, el receptor leería los mensajes en el orden inverso que el emisor los escribió.
Es decir, en una pila el ultimo que entra es el primero que sale. De ahí que a las pilas se les conozca como estructuras LIFO(*Last In, First Out*).
Existen otras estructuras llamadas colas, en una cola el primero que entra es el primero que sale. Su nombre deriva de las filas que se hacen en los supermercados, cines, bancos, etc. Donde el primero que llega, es el primero en ser atendido, y el último que llega es el último en ser atendido(suponiendo que no haya preferencias burocráticas en dicho establecimiento).
Las colas, son conocidas como estructuras FIFO(*First In, First Out*).
Una cola es una estructura de datos, en la cual sólo se pueden aplicar estas dos operaciones: colocar un elemento al final, o quitar un elemento del principio.
Para representar una cola, obviamente necesitamos tambien un arreglo; supongamos que hay varios participantes en la cola para el registro de la OMI. Cada participante tiene un nombre que consiste de un número entero mayor o igual que 1.(Son bastante populares esos nombres en nuestros días)
Y como faltan muchos estados a la OMI, solo habrá 10 participantes.
Entonces el arreglo podría lucir algo asi...
0 0 0 0 0 0 0 0 0 0
En ese momento llega 3 y se forma
3 0 0 0 0 0 0 0 0 0
Luego llega 5 y se forma
3 5 0 0 0 0 0 0 0 0
Despues llega 4 y se forma
3 4 5 0 0 0 0 0 0 0
Luego llega 9, y despues de eso llega 2
3 4 5 9 2 0 0 0 0 0
Entonces al encargado del registro se le ocurre comenzar a atender, y atiende a 3.
En la vida real, los participantes darían un paso hacia delante, pero en una computadora, para simular eso, sería necesario recorrer todo el arreglo, lo cual es muy lento; por ello, es mas practico dejar el

primer espacio de la cola en blanco.
0 4 5 9 2 0 0 0 0 0
Luego se atiende a 4
0 0 5 9 2 0 0 0 0 0
En ese momento llega 1 corriendo y se forma
0 0 5 9 2 1 0 0 0 0
etc.
Ya para este momento, te debes de estar imaginando que para implementar una cola, unicamente se requiere un arreglo y dos variables, donde las variables indican donde inicia y donde termina la cola.
En un problema de olimpiada, siempre podremos saber cuantos elementos "se formarán" en la cola. Sin embargo, suponiendo que se forma 1 e inmediatamente es atendido, se vuelve a formar y vuelve a ser atendido inmediatamente, y asi 1 000 veces, entonces, de esa manera se requeriría un arreglo de tamaño 1 000, cuando nunca hay mas de un elemento dentro de la cola.
Para evitar eso, podemos añadir elementos al principio de la cola si ya no hay espacio al final.
Por ejemplo, si luego de que se atiendió a muchos participantes, la cola está de esta forma:
0 0 0 0 0 0 0 0 7 3
Y para rectificar algo 5 vuelve a formarse, podemos colocar a 5 al principio
5 0 0 0 0 0 0 0 7 3
Luego si 4 vuelve a formarse
5 4 0 0 0 0 0 0 7 3
Puede parecer extraño esto, pero el programa sabrá que la cola empieza donde está 7 y termina donde está 4.
Asi que si el organizador atiende al siguiente, atenderá a 7, y la cola quedará de esta manera
5 4 0 0 0 0 0 0 0 3
Luego atenderá a 3
5 4 0 0 0 0 0 0 0 0
Despues atenderá a 5
0 4 0 0 0 0 0 0 0 0
Y asi sucesivamente...
Implementar la operación de meter un elemento a la cola es muy sencillo:

```
cola[fin++]=elemento;
if(fin>=tamaño de la cola)
        fin=0;
```

Y casí lo mismo es sacar un elemento de la cola

```
inicio++;
if(inicio>=tamaño de la cola)
        inicio=0;
```

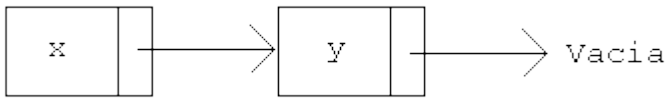
Problema 7. Comprueba que una cola esta llena o vacía si y solo si inicio=fin

Problema 8. En tu negocio tienes una secuencia de *n* sillas acomodas en línea recta y numeradas de 1 a

n, por cada persona que se siente en una silla con el número *x*, deberás pagar $*x* al gobierno, pero como quieres tener cómodos a los clientes, no tienes otra opción que pedirles que tomen asiento; sin embargo, el aprecio por el cliente aún no te ha quitado la codicia, por lo que puedes indicarle a cada cliente donde sentarse, pero el cliente deside cuando irse; desde el momento que un cliente llega hasta el momento que se va, el cliente pasará todo el tiempo sentado en una silla y no estará dispuesto a compartirla con nadie mas.
Escribe un programa que dado el historial de qué clientes llegan y qué clientes se van en orden cronológico, calcule que asientos asignarles a cada quien para que el dinero que debas pagar al gobierno sea el mínimo posible. Tu programa deberá funcionar en tiempo lineal.

Listas Enlazadas

Frecuentemente necesitamos tener almacenadas unas *k* listas de datos en memoria, sabiendo que el número total de datos en memoria no sobre pasa *n*.
Si disponemos de suficiente memoria, podemos guardar en memoria *n* arreglos de tamaño *k* o una matriz de tamaño *nk*, pero no siempre dispondremos de tanta memoria.
Tambien hay veces que se requieren tener listas de números y agregar números a dichas listas pero no al final ni al principio, sino en medio.
Para solucionar estos y otros problemas, existen las litas enlazadas.
Las listas enlazadas son estructuras de datos compuestas por una sucesión de elementos llamados nodos; en la que cada nodo contiene un dato y la dirección del proximo nodo, en caso de que haya próximo.
La siguiente imagen muestra una representación gráfica de una lista enlazada.



Una forma de definir un nodo es:
- Una estructura vacía ó
- un elemento de información y un elace a otro nodo.

La tarea de implementar una enlazada puede hacerse eficazmente con 2 arreglos: uno para guardar los datos y otro para guardar los enlaces, además se requiere una variable que diga el tamaño de la lista de la siguiente manera...

```
int dato[tamaño maximo de la lista];
int proximo[tamaño maximo de la lista];
int tam_lista=1; //Tamaño de la lista
```

Lo único que falta definir es el elemento vacío, para ello, podemos asumir que el dato 0 es el elemento vacío, y en el momento que nos encontremos con él, sabemos que la lista ya habrá terminado.
Insertar un nodo con un dato *x*, justo despues de otro nodo *k*, se puede hacer facilmente en tiempo

constante:

```
1       void insertar(int x, int k){
2               dato[tam_lista]=x;
3               proximo[tam_lista]=proximo[k];
4               proximo[k]=tam_lista++;
5       }
```

Lo que hace este código es colocar *x* en el primer espacio en blanco dentro del arreglo *datos*, luego colocar un enlace al sucesor de *k* en el primer espacio en blanco dentro del arreglo de *proximo*, y hacer que *k* apunte al nodo que se acaba de crear.
De esa forma *k* apuntará al nuevo nodo, y el nuevo nodo apuntará al nodo que apuntaba *k*.
El siguiente código imprime todos los datos contenidos en una lista, asumiendo que 1 es el primer elemento de dicha lista.

```
1       for(i=1;i!=0;i=proximo[i])
2               printf("%d", dato[i]);
```

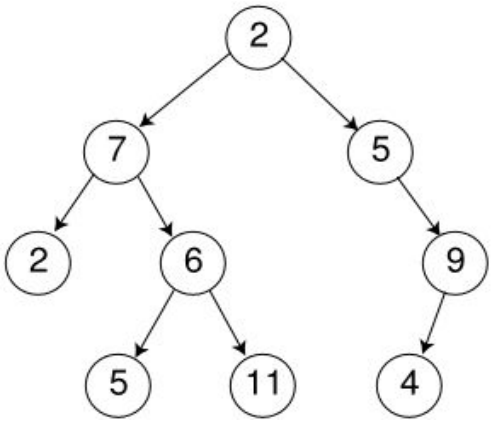
Problema 9. Implementa una lista donde se puedan realizar las siguientes operaciones: imprimir los datos de la lista en órden, agregar un nodo a la lista y eliminar un nodo especificado de la lista.

Problema 10. Si borras un nodo de la lista, simplemente borrando su enlace, no podras reutilizar ese espacio de memoria para guardar mas datos. Implementa una lista que permita insertar despues de x en orden constante, imprimir la lista en orden lineal, borrar un nodo x en orden constante, y que reutilize los espacios de memoria de los nodos borrados.
(*nota: es muy facil hacer esto con memoria dinamica, pero se insiste en que no es conveniente usarla en la olimpiada, hay una solución de este problema que no requiere memoria dinamica, aunque puede llegar a ocupar el doble de memoria que se requiere para guardar los datos en la lista; encuentrala.)

Problema 11. Un verdugo es mandado a exterminar a *n* prisioneros de guerra. El exterminio lo ejecuta de la siguiente manera: los prinsioneros forman un círculo al rededor del verdugo, el verdugo elige a quien fusilar primero, una vez muerto el primero, el verdugo cuenta, a partir del lugar donde estaba su ultima victima, k prisioneros en orden de las manesillas del reloj, y luego fusila al k-ésimo prisionero despues de su última víctima (a los muertos no los cuenta), y repite este proceso hasta que solo quede un prisionero.
El último prisionero podrá ser liberado.
El verdugo tiene un amigo entre los *n* prisioneros, escribe un programa que dado, n, k y la ubicación de su amigo, le diga a quien fusilar primero, para asegurar que su amigo sea el que quede libre.

Árboles Binarios

Seguramente despues de haber visto las listas enlazadas, te llegó a la mente la idea de colocar mas de un enlace en cada nodo.
Pues no fuiste el primero en tener esa idea, pues los árboles binarios son estructuras de datos parecidas a las listas enlazadas, con la diferencia de que cada nodo puede tener hasta 2 enlaces(de ahí el nombre de binario).
Por ahora nos reservaremos la definición rigurosa de árbol binario, pues requiere de teoría de grafos para comprenderla.
La siguiente figura muestra un ejemplo de árbol binario:



Antes de seguir entrando en materia, será conveniente dar unas cuantas definiciones:
El primer nodo del árbol recibe el nombre de raíz, los enlaces reciben el nombre de aristas.

- Se dice que un nodo B es hijo de un nodo A, si existe alguna arista que va desde A hasta B. Por ejemplo, en la figura, 7 es hijo de 2, 4 es hijo de 9, 11 es hijo de 6, etc.
- Al mismo tiempo se dice que un nodo A es padre de un nodo B si existe una arista que va desde A hasta B. Ej. 9 es padre de 4, 6 es padre de 5 y de 11, etc.
- Se dice que un nodo es hoja, si no tiene hijos. Ej. 11 y 4 son hojas, pero 6, 7, y 9 no lo son.
- La rama izquierda de un nodo es el árbol que tiene como raíz el hijo izquierdo de tal nodo, por ejemplo {7, 2, 6, 5, 11} son los nodos de la rama izquierda de 2.
- La rama derecha de un nodo es el árbol que tiene como raíz el hijo derecho de tal nodo.
- Los nodos tambien pueden ser llamados vertices.

Los árboles binarios tienen muchas formas de implementarse, cada una con sus ventajas y desventajas, por ahora solo trataremos una.
Los árboles binarios pueden ser implementados utilizando 3 arreglos: clave(los datos que guarda cada nodo), hijo izquierdo e hijo derecho.
Y por supuesto otra variable entera: el número de nodos en el árbol.

int clave[*maximo de nodos*]; //Dato que guarda el nodo

int izq[*maximo de nodos*]; //Hijo izquierdo
int der[*maximo de nodos*]; //Hijo derecho
int nodos=0;

En seguida se muestran los datos que podrían contener los arreglos para obtener el árbol que se muestra el la figura:

nodo	***0***	***1***	***2***	***3***	***4***	***5***	***6***	***7***	***8***
Clave	2	7	2	6	5	11	5	9	4
Izq	1	2	-1	6	-1	-1	-1	8	-1
Der	4	3	-1	5	7	-1	-1	-1	-1

Como se observa en la tabla, se está tomando el 0 como raíz, y el -1 como vacío.
Los árboles binarios tienen diversas aplicaciones, las mas frecuentes implican ordenamiento.
Existen, además, 3 formas recursivas distintas de visitar los nodos de un árbol binario:

- Recorrido en Preorden. Tambien llamado Búsqueda en Profundidad en la teoría de grafos, consiste en visitar primero el nodo actual, luego el hijo izquierdo y despues el hijo derecho. Puede ser implementado de la siguiente manera:

```
1    void preorden(int nodo){
2            if(nodo==-1)
3                    return;
4            visita(nodo);
5            preorden(izq[nodo]);
6            preorden(der[nodo]);
7    }
```

  Si lo deseas, puedes visualizar graficamente el recorrido en preorden en la siguiente dirección:
  http://www.olimpiadadeinformatica.org.mx/material%20de%20estudio/Presentaciones/Arboles_Preorden.swf

- Recorrido en Orden. En algunos lugares se le menciona como inorden, consiste en visitar primero el hijo izquierdo, luego el nodo actual, y finalmente el hijo derecho. El siguiente código lo ilustra:

```
1    void orden(int nodo){
2            if(nodo==-1)
3                    return;;
4            orden(izq[nodo]);
5            visita(nodo)
6            orden(der[nodo]);
7    }
```

  Si lo deseas, puedes visualizar graficamente el recorrido en orden en la siguiente dirección:
  http://www.olimpiadadeinformatica.org.mx/material%20de%20estudio/Presentaciones/Arboles_Orden.swf

- Recorrido en Postorden. Como ya te imaginarás, consiste en visitar primero los nodos hijos y después el nodo actual.

```
1    void postorden(int nodo){
2            if(nodo==-1)
3                    return;;
4            postorden(izq[nodo]);
5            postorden(der[nodo]);
6            visita(nodo)
7    }
```

Si lo deseas, puedes visualizar graficamente el recorrido en postorden en la siguiente dirección:
http://www.olimpiadadeinformatica.org.mx/material%20de%20estudio/Presentaciones/Arboles_Postorden.swf

Quizá de momento no les encuentras mucha aplicación a dichos recorridos, pero cada uno tiene su utilidad específica.
Se puede crear un nuevo nodo facilmente con la siguiente función:

```
1       int crear_nodo(int dato){
2               clave[nodos]=dato;
3               izq[nodos]=-1;
4               der[nodos]=-1;
5               return nodos++;
6       }
```

Como se puede ver, la función recibe como parametro el dato que va a tener el nuevo nodo y regresa el lugar en el arreglo donde está ubicado el nuevo nodo, depende del programador saber dónde poner el enlace del nuevo nodo.

Problema 12. Comprueba que todo árbol binario tiene *n*-1 aristas, donde *n* es el número de nodos del árbol.

Problema 13. La sucesión de los números catalán se define de la siguiente manera:
c(0)=1
c(x)=c(x-1)*c(0)+c(x-2)*c(1)+c(x-3)*c(2)+...+c(x-1)*c(0)
Comprueba que c(*n*) es el número de árboles binarios que se pueden formar con *n* nodos (sin tomar en cuenta las claves).

Problema 14. Existe una manera de dibujar árboles binarios que requiere una cuadricula con *n* columnas numeradas de 1 a n(suponiendo que *n* es el número de nodos y hay un número ilimitado de filas). El nodo raíz se dibuja en la fila 1, sus hijos en la fila 2, los hijos de estos ultimos en la fila 3, etc. Además, para todo nodo A, todos los nodos de la rama izquierda de A deben de dibujarse en columnas menores a la columna donde se dibuja A, y todos los nodos de la rama derecha de A, deben dibujarse en columnas mayores a la columna donde se dibuja A.Escribe un programa que, dado un arbol binario, determine en qué columna dibujar cada nodo. (Croatian Olympiad in Informatics)

Como se dijo anteriormente, los árboles binarios comunmente se utilizan en algoritmos que requieren de una u otra manera ordenamiento.
Para ello, es muy frecuente utilizar árboles binarios de búsqueda.
Un árbol binario de búsqueda es un árbol binario con datos guardados en sus nodos, que cumple con las siguientes propiedades:

- El valor de cada nodo es mayor o igual que los valores de los nodos de su rama izquierda.
- El valor de cada nodo es mayor o igual que los valores de los nodos de su rama derecha.

Para crear un árbol binario de busqueda, basta con colocar cualquier nodo como raíz, y luego insertar nodos de la siguiente manera:

```
1       void insertar(int nodo, int dato){
2               if(clave[nodo]>dato){
3                       if(izq[nodo]==-1) {
4                               izq[nodo]=crear_nodo(dato);
5                       }else{
6                               insertar(izq[nodo], dato);
7                       }
8               }else{
9                       if(der[nodo]==-1){
10                              der[nodo]=crear_nodo(dato);
11                      }else{
12                              insertar(der[nodo], dato);
13                      }
14              }
15      }
```

La función anterior, crea un nuevo nodo con clave *dato* y lo inserta en un árbol(o subárbol) que tiene raiz en *nodo*.
Asi que si queremos generar un árbol de búsqueda con las claves 5, 4, 8, 2, 1, lo unico que tenemos que hacer es:

```
raiz=crear_nodo(5);
insertar(raiz, 4);
insertar(raiz, 8);
insertar(raiz, 2);
insertar(raiz, 1);
```

Problema 15. Demuestra que la función *insertar* que se acaba de mostrar realmente inserta un nodo con clave *dato* en un árbol binario con raíz en *nodo,* de manera que si el árbol binario es de búsqueda, seguirá siendolo.

Problema 16. ¿Cual es la complejidad de insertar un nuevo nodo en un árbol binario de búsqueda en el caso promedio?, ¿y en el peor de los casos?

Problema 17. Muestra que si un arbol binario de búsqueda se recorre en inorden los nodos se visitarán en orden no descendente. Una vez demostrado esto, ¿Cual es la complejidad de ordenar una secuencia de n números creando un árbol binario de búsqueda?

Problema 18. Escribe un programa que dado un entero k y un árbol binario de busqueda, determine si existe un nodo con clave k en dicho árbol, tu programa deberá realizar esta operación en O(log n) en caso promedio y O(n) en el peor caso.

Programa 19. Escribe un programa que dados los recorridos en postorden y orden, determine el recorrido en preorden en O(n log n).

Grafos

Muchos problemas, nos presentan algun conjunto, en el cual algunos pares de elementos de dicho conjunto se relacionan entre sí.

Ejemplo: En una reunión hay n invitados, se asume que si un invitado a conoce a un invitado b, b tambien conoce a a, comprueba que en la reunión hay al menos 2 invitados que conocen al mismo número de invitados. Asúmase tambien que todos conocen por lo menos a alguien de la reunión.

En el ejemplo anterior, tenemos un conjunto de n personas, y como vemos, algunos pares de personas se conocen entre sí.
Para resolver este tipo de problemas es muy útil usar grafos, tambien llamados “graficas” por algunos, sin embargo, aquí se utilizará el término “grafo”, para evitar que se confunda con el concepto de la grafica de una función.
Una forma un tanto pobre de definir un grafo(poco después trataremos la definición formal) es:
“ conjunto de puntos llamados vértices, en el cual algunos vertices pueden estar unidos por líneas llamadas aristas”.
Volviendo al ejemplo anterior, es posible dibujar un punto por cada persona de la reunión, y entre cada par de personas que se conozca entre sí, colocar una línea.
Por ejemplo si 6 conoce a 4, 4 conoce a 5, 5 conoce a 1, 5 conoce a 2, 2 conoce a 1, 3 conoce a 2 y 4 conoce a 3, se podría dibujar un grafo similar a este:

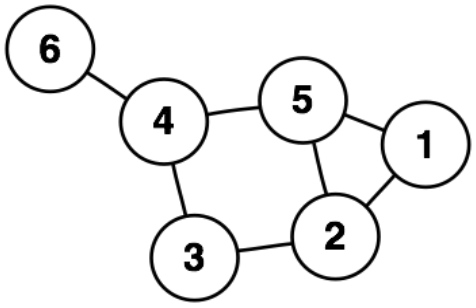


Figura G1

Ahora, una vez que tenemos visualizada de cierta manera la solución, podemos ver que cada nodo puede estar conectado con a lo menos un nodo(cada invitado conoce por lo menos a otro invitado) y a lo mas *n*-1 nodos.

Es decir, el número de conocidos de cada persona va desde 1 hasta *n*-1. Como cada persona puede tener *n*-1 números distintos de conocidos y hay *n* personas, por principio de las casillas al menos dos personas deberán tener el mismo número de conocidos.

Los grafos son muy útiles en problemas que involucran estructuras tales como carreteras, circuitos electricos, tuberías de agua y redes entre otras cosas, y como lo acabamos de ver, con los grafos incluso se pueden representar las relaciones sociales en una sociedad.

Si eres observador, tal vez ya te diste cuenta que tanto las listas enlazadas como los árboles son grafos.

Hay que hacer notar que no importa cómo se dibujen los grafos, si las aristas estan curvas o rectas, o si los vertices estan en diferente posición no es relevante, lo unico que importa qué pares de vertices estan unidos.

Por ejemplo:

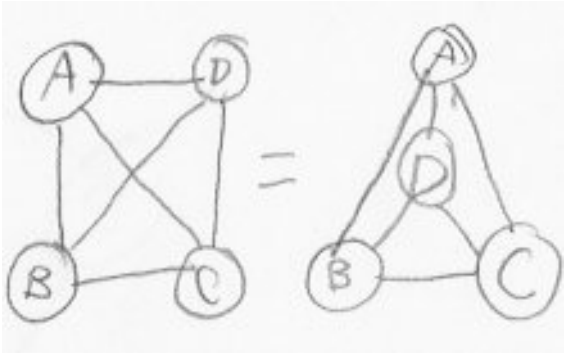


Figura G2

Es decir, un grafo, mas que un conjunto de puntos unidos por líneas, es un conjunto de objetos en el cual algunos pares de estos se relacionan entre sí, mas formalmente:

"Un grafo es un conjunto G={V, A}, donde los elementos de V son llamados vertices, y A es un conjunto de pares de vértices llamadas aristas."

De esa forma, el grafo que se dibuja arriba se puede expresar como:
G={{A, B, C, D}, {AB, AD, DB, DC, BC, AC} }

Esto no quiere decir que para manejar grafos hay que olvidarse completamente de dibujarlos como puntos unidos por líneas, ya que en un dibujo a simple vista se pueden distinguir detalles del grafo que no serían tan facil distinguirlos sin el dibujo.
Antes de proseguir es conveniente ver un poco de la notación basica en los grafos:

- El grado de un vertice es el número de aristas conectadas con él. Por ejemplo, en la figura G1 el nodo etiquetado como 1 tiene grado 2, y el nodo etiquetado como 4 tiene grado 3, mientras que en la figura G2 todos los nodos tienen grado 3.
- El número de aristas en un grafo G={V, A} se denota como |A|.
- El número de vertices en un grafo G={V, A} se denota como |V|.
- Un camino C es una sucesión de vertices $C=\{v_1, v_2, v_3, ..., v_n\}$, tal que para toda $0 \leq i < n$ existe una arista que conecta v_i con v_{i+1}.
- Un ciclo es un camino $C=\{v_1, v_2, v_3, ..., v_n\}$ donde $v_1= v_n$.
- Un grafo conexo es un grafo G={V, A} donde para todo v_i, v_j que pertenezcan a V, existe un camino $C=\{v'_1, v'_2, v'_3, ..., v'_n\}$ donde $v'_1=v_i$ y $v'_n=v_j$. Otra forma de decirlo es, es posible ir de cualquier vertice a cualquier otro.
- Un grafo dirigido es aquel en el cual las aristas tienen dirección, es decir una arista entre v_i y v_j no implica una arista entre v_j y v_i (Figura G3).

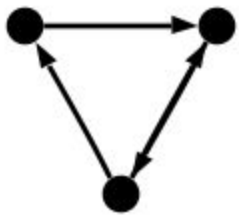


Figura G3 – Un grafo dirigido

- Un grafo no dirigido es aquel en el cual las aristas no tienen dirección, es decir una arista entre v_i y v_j implica una arista entre v_j y v_i (Figura G4).

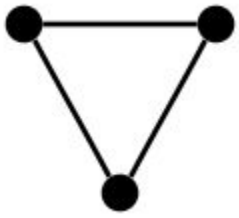


Figura G4 – Un grafo no dirigido

- Un árbol es un grafo conexo sin ciclos.
- Un bosque es un grafo sin ciclos.

Problema 20. Comprueba que en todo grafo no dirigido $G=\{V, A\}$, el número de vértices con grado impar siempre es par.

Problema 21. Comprueba que la suma de todos los grados de todos los vertices en un grafo no dirigido

G={V, A} es 2|*A*|.

Problema 22. Comprueba que todo árbol tiene al menos 2 vertices de grado 1.

Problema 23. Sea *G* un grafo conexo con 2*n* vertices, demuestra que borrando un subconjunto de aristas, es posible obtener un grafo conexo tal que todos sus vertices tengan grado impar.
(Amir2, Mathlinks forums)

Problema 24. Comprueba que para todo árbol de *n*-nodos en el que todos sus nodos, a excepción de las hojas, tienen exactamente *k* hijos, entonces *n* mod *k* = 1.
(Ian PARBERRY, Problems on Algorithms)

Problema 25. Un árbol de expansión de un grafo *G*={*V*, *A*} es un árbol *T*={*V*, *B*} donde *B* es subconjunto de *A*. Prueba que un grafo de *n* nodos no puede tener mas de (*n*-1)! árboles de expansión.

Problema 26. *n* pueblos estan conectados por 2*n*-1 senderos de una sola dirección. Es posible ir desde cualquier pueblo a todos los demas. Muestra que podemos borrar un sendero de tal manera que la condición anterior se siga cumpliendo.
(Labiliau, Mathlinks forums)

Problema 27. Prueba que es posible colorear los *n* vertices de un grafo(cada uno tiene un grado a lo mas de 5) con 2 colores, tal que al menos 3/5 de sus aristas tengan extremos de diferente color.
(Ukraina 2004)

Problema 28. Algunos pares de pueblos en un país estan unidos por senderos, hay una unica ruta para ir de cualquier pueblo a cualquier otro sin pasar 2 veces por el mismo pueblo. Exactamente 100 pueblos tienen un solo sendero. Muestra que es posible construir 50 nuevos senderos, de tal manera que si algun sendero(nuevo o viejo) es cerrado por mantenimiento, aún habrá alguna ruta entre cualquier par de pueblos del país.
(Russian 2001)

Luego de haber visto la naturaleza de los grafos, seguramente estas pensando en cómo implementar una estructura de datos similar a un grafo.
Hay muchas maneras de implementar grafos, pero solo se tratarán las 2 mas usadas: matriz de adyacencia y listas de adyacencia.

- Esta es quizá la forma mas común de representar un grafo en un lenguaje de programación debido a su sencilles, sin embargo, requiere $|V|^2$ de memoria, y en la mayoría de los casos es un poco lenta.
  Un grafo G={V, A} puede ser representado como una matriz M de |V| x |V|, donde $M_{i,j} \neq 0$ si existe una arista en A que une a los nodos v_i y v_j. Por ejemplo, el grafo de la Figura G1 puede ser

representado como:

M={
0 1 0 0 1 0
1 0 1 0 0 0
0 1 0 1 0 0
0 0 1 0 1 0
1 1 0 1 0 0
0 0 0 1 0 0
}

- Las listas de adyacencia consisten en |V| listas enlazadas, es decir, una lista enlazada para cada vertice *v* que pertenzca a V, dicha lista consta de los nodos que estan unidos a *v* por medio de una arista. La ventaja fundamental de las listas de adyacencia sobre la matriz de adyacencia es que las listas de adyacencia solo requieren |A|+|V| de memoria.

La implementacion de un grafo por matriz de adyacencia es bastante simple:

int m[*número_de_vertices*][*número_de_vertices*];

Sin embargo, la representación de un grafo por listas de adyacencia es un poco mas compleja y muchos se ven tentados a usar memoria dinamica para estos casos; nuevamente se hace incapie que la memoria dinamica no es para usarse en la olimpiada.
Si tenemos un grafo de 100 000 nodos, y a lo mas un millón de aristas, es obvio que no podemos 100 mil listas reservando espacio para 100 mil nodos en cada una, ya que requeriría una gran cantidad de memoria que es muy probable que carezca de ella la computadora donde se vaya a ejecutar el programa.
La solución para esto, es guardar los datos de todas las listas en un solo arreglo, y guardar cual es el primer elemento de cada lista en otro arreglo.
Además, será necesario tener una variable que indique el número de aristas de la lista.
En el siguiente código ilustra cómo implementar un grafo con a lo mas 100 000 nodos y a lo mas un millón de aristas:

```
int dato[1000001]; //El nodo hacia el cual apunta la arista
int proximo[1000001]; //El siguiente elemento de la lista
int primero[100001]; //El primer elemento de la lista de cada nodo
int aristas=1; //El número de aristas
```

Aquí se declaró como una arista inicialmente para dejar libre el 0 para expresar el elemento vacío que indica el final de toda lista.
El procedimiento de insertar una arista entre 2 nodos también es algo tedioso con listas de adyacencia comparandolo a lo simple que es con matriz de adyacencia, la siguiente función inserta una arista entre los nodos *v* y *w* en un grafo dirigido.

```
1       void insertar_arista(int v, int w){
2               dato[aristas]=w;
3               proximo[aristas]=primero[v];
4               primero[v]=aristas++;
5       }
```

Como se ve en el código, la arista siempre se inserta al principio de la lista de adyacencia, para que de esa forma, el conjunto de todas las aristas se inserte en tiempo lineal; ya que si se recorriera la lista de adyacencia cada que se quisiera insertar una nueva arista, insertar todo el conjunto de aristas se realizaría en tiempo cuadratico, queda como tarea para el lector demostrar por qué.

Problema 29. ¿Qué sucede si se eleva al cuadrado una matriz de adyacencia que consta solo de 1s y 0s?, ¿si se eleva al cubo?, ¿si se eleva a la n?

Problema 30. Multiplicar una matriz tamaño n x n por otra matriz de n x n requiere de n^3 multiplicaciones. Encuentra un algoritmo de tiempo $O(n^3 \log k)$ para elevar una matriz M de n x n a la k.

Recorridos en Grafos

Ahora que ya conocemos los grafos y cómo representarlos en una computadra comenzaremos con una de sus aplicaciones mas básicas: los recorridos en grafos o búsquedas.
Básicamente las búsquedas sirven para visitar todos los vertices de un grafo conexo, o bien de un subgrafo conexo de manera sistemática.
Es decir, iniciamos en el vertice A, y queremos saber desde ahí, a cuáles vertices se puede llegar, y la manera de hacerlo es... ¡Buscando! :D.
El primer recorrido que veremos será la busqueda en profundidad.
La busqueda en profundidad es un recorrido de grafos de naturaleza recursiva(aunque se puede implementar de manera no recursiva).
¿Recuerdas la antigua leyenda del minotauro? En caso de que no la recuerdes aquí va la parte que interesa:
Había un minotauro(un hombre con cabeza de toro) dentro de un laberinto, un valiente joven, llamado Teseo se dispuso a matar al minotauro(el minotauro mataba gente, asi que no tengas lastima del minotauro ;)), pero para no perderse en el laberinto, tomó una cuerda para ir marcando por donde ya había pasado, y saber qué recorrido había seguido, para así poder regresar. Luego cuando mató al minotauro regresó siguiendo su rastro, como lo había planeado, y el resto ya no tiene que ver con el tema.
¿Qué moraleja nos deja esta historia?
Basicamente que si nos encontramos en un laberinto(o en un grafo), es necesario ir marcando el camino por el que ya pasamos y saber por dónde veníamos.
La idea de la búsqueda en profundidad es dejar una marca en el nodo que se esta visitando; y despues

visitar recursivamente los vecinos de dicho nodo que no hayan sido visitados.
El siguiente codigo es una implementacion de la búsqueda en profundidad en un grafo con *N* nodos y con matríz de adyacencia *g*:

```
1       void visitar(int nodo){
2               visitado[nodo]=1;
3               for(i=1;i<=N;i++)
4                       if(g[nodo][i]!=0 && visitado[i]==0){
5                               visitar(i);
6                       }
7       }
```

Hay que mencionar que en un grafo $G=\{V, A\}$, la búsqueda en profundidad funciona con matriz de adyacencia en $O(|V|^2)$ mientras que con listas de adyacencia funciona en $O(|V|+|A|)$.
El otro recorrido que se va a tratar aquí es la búsqueda en amplitud(algunos lo llaman búsqueda a lo ancho). Lamentablemente la búsqueda en amplitud no fue idea de Teseo, y como te imaginarás, es un poco mas difícil que a alguien se le ocurra cómo hacela si no la conoce.
Para comenzar a adentrarnos en la búsqueda en amplitud es conveniente que imagines la siguiente situación:
Eres miembro de una banda de bándalos(valga la redundancia), y la banda tiene como centro de reunión una esquina de la ciudad. Los miembros de la banda quieren que toda la ciudad sepa a cuántas cuadras se encuentran en cada momento de "su terrirtorio", asi que deciden dejar pintados mensajes en la paredes de cada esquina de la ciudad diciendo a cuántas cuadras de encuentran del "territorio prohibido".
Ahora, el jefe de la banda te ha encargado a ti determinar qué número qué número se debe de pintar en cada esquina de la ciudad.
La tarea no es facil, ya que si haces rodeos, podrías llegar a creer que estas mas lejos de lo que realmente te encuentras, y la ciudad no es muy rectangular que digamos.
Piensa un poco cómo resolver ese problema y luego continua leyendo.
.....
Bueno, si no se te ocurrió la solución(ojalá si la hayas pensado un poco), lo que deberías hacer en ese caso, es pintar algo asi como "Alejate, este es nuestro territorio" en la esquina donde la banda tiene su centro de reunión, luego caminar una cuadra en cualquier dirección y pintar un algo asi como "estas a una cuadra de territorio prohibido", después regresar al punto de encuentro y caminar una cuadra en otra dirección y otra vez pintar lo mismo, repitiendo esa operación hasta que ya se haya caminado en todas las direcciones.
Después, para cada esquina donde hayas escrito "estas a una cuadra de terrirtorio prohibido", caminas una cuadra en todas las direcciones y escribes "estas a dos cuadras de territorio prohibido"(claro, si no habías pintado nada ahí anteriormente).
Y continúas así hasta haber pintado toda la ciudad o hasta que te agarre la policía.
Nota: la solución anterior no pretende fomentar el hábito del grafiti, sino enseñar la búsqueda en

amplitud.
La idea de la búsqueda en amplitud es visitar un nodo inicial *v*, y luego visitar los nodos que se encuentren a una arista de *v*, después los que se encuentren a 2 aristas de *v* y asi sucesivamente.
De esa forma, la búsqueda en amplitud visita todos los nodos que pueden ser alcanzados desde *v*, y para cada uno de ellos, computa su distancia(mínimo número de aristas) con *v*.
En la situación que imaginamos del bándalo pintando paredes, el bándalo tenía que caminar mucho despues de pintar cada esquina, sin embargo, como nosotros usaremos una computadora para realizar un proceso similar, no necesitamos estar *caminando*, o recorriendo todo el grafo en busca del siguiente vertice que hay que marcar.
Una forma mas rápida de obtener los mismos resultados es teniendo una cola, y metiendo a *v* en la cola, luego. mientras la cola no este vacía, sacar un vertice de la cola, visitarlo, y meter a los vecinos no visitados a la cola.
Una manera de hacerlo(asumiendo que la cola nunca se llenará) es la siguiente:

```
1       cola[fin++]=v; //Meter v a la cola
2       visitado[v]=1; //Inicializar
3       distancia[v]=0;
4       while(inicio!=fin){ //Mientras la cola no este vacía
5               a=cola[inicio++]; //Sacar de la cola
6               for(i=1;i<=N;i++) //Meter vecinos a la cola
7                       if(g[a][i] && visitado[i]==0){
8                               visitado[i]=1;
9                               distancia[i]=distancia[a]+1;
10                              cola[fin++]=i;
11                      }
12      }
```

El código anterior usa una matriz de adyacencia *g*, con *N* nodos.
Como nodo se visita una vez y cada arista se visita 1 o 2 veces dependiendo de si el grafo es dirigido o no dirigido, este algoritmo funciona en tiempo $O(|V|^2)$ con matriz de adyacencia y con listas de adyacencia funciona en tiempo $O(|V|+|A|)$.

Problema 31. Dado un grafo $G=\{V, A\}$ y dos vertices *s* y *t* en *V*, escribe un programa que encuentre el camino mas corto(el de menos aristas) entre *s* y *t* en $O(|V|+|A|)$.

Problema 32. ¿Es posible generalizar la solución del problema anterior para encontrar el camino mas largo en $O(|V|+|A|)$?, comprueba tu respuesta.

Problema 33. En una extensa red de carreteras, cada carretera admite el pase de automoviles de un peso máximo, tienes un carro de peso *w*, y quieres saber si puedes ir de la ciudad *i* a la ciudad *f*. Escribe un programa que dado un grafo $G=\{V, A\}$(los vertices como ciudades y las aristas como carreteras)

representando la red de carreteras, el peso máximo que soporta cada carretera, *w*, *i* y *f,* determine si puedes viajar desde *i* hasta *f* con tu carro de peso *w* en tiempo O(|*A*|+|*V*|).

Problema 34. Considera la misma situación del problema anterior, escribe un programa que funcione en tiempo O((|*A*|+|*V*|) log *A*) y que dados los mismos datos, determine cual es el carro mas pesado con el que se puede viajar desde *i* hasta *f*. (Programing Challengues, UVa)

Un grafo torneo es un grafo dirigido *G*={*V*, *A*}, en el cual para todo par de vertices *v*, *w* pertenecientes a *V*, existe una arista, ya sea que vaya desde *v* hacia *w* o desde *w* hacia *v*.

Problema 35. Comprueba que en todo grafo torneo *G*={*V*, *A*} existe un vertice *v* en *V*, al cual es posible llegar desde cualquier otro vértice.
(ARTHUR ENGEL, Problem-Solving Strategies)

Un camino hamiltoniano es un camino que pasa por todos los vértices.

Problema 36. Comprueba que en todo grafo torneo existe a lo menos un camino hamiltoniano. (IAN PARBERRY, Problems on Algorithms)

Problema 37. Escribe un programa que dado un grafo torneo, encuentre un camino hamiltoniano en dicho grafo en tiempo O(|*V*| log |*V*|).

Un grafo bipartito es cualquier grafo G={V, A}, en el cual V esta compuesto por nodos blancos y negros V={B, N}, y no existe arista alguna con extremos a y b, tal que a y b pertenezcan a B ó, a y b pertenezcan a N, es decir, un garfo bipartito es aquel en el que sus vertices se pueden colorear de blanco o de negro de tal forma que los nodos negros solamente esten unidos con nodos blancos y los nodos blancos solamente esten unicamente conectados con nodos negros.

Problema 38. Escribe un programa que determine si un grafo *G*={*V*, *A*} es bipartito en O(|*V*|+|*A*|).

Otras Estructuras de Datos

En este tutorial se trataron las estructuras de datos básicas mas usadas, pero hay otras estructuras de datos que se derivan de éstas, sólo algunas de estas se mencionan a continuación:

- Colas dobles. Estructuras de datos que constan de una sucesión de elementos y permiten insertar un elemento al principio, quitar un elemento del principio, insertar un elemento del final, y quitar un elemento del final. Todas estas operaciones en O(1).
- Colas de prioridad. Estructuras de datos que constan de las siguients operaciones: saber cual es el elemento mas grande, quitar el elemento mas grande e insertar un nuevo elemento.

- Listas circulares. Las listas circulares son listas de adyacencia en las cuales el ultimo nodo tiene un enlace al primero.
- Listas doblemente enlazadas. Practicamente iguales a las listas enlazadas, pero además de que cada elemento apunte al siguiente, cada elemento apunta al siguiente y al anterior.
- Árboles AVL. Un árbol AVL es un árbol binario de búsqueda en el cual, para cada vértice, la diferencia entre las alturas de su rama izquierda y su rama derecha es menor o igual a 1. Hay algoritmos que permiten inserción y eliminación de nodos en un árbol AVL en O(*log* n).
- Montículos. Árboles binarios, en los cuales cada nodo es mayor que sus hijos; además para cada nodo la diferencia de alturas entre la rama izquierda y la rama derecha es menor o igual que 1, y la rama izquierda siempre tiene igual o mas nodoso que la derecha, o mejor dicho, todos los nodos del último nivel estan "tan a la izquierda como puedan" en el árbol. Generalmente se utiliza para colas de prioridad o para ordenamiento ya que permite inserción en tiempo logarítmico, saber cual es el mayor en tiempo constante, y eliminación del mayor en tiempo logarítmico.
  Para mas información sobre montículos visitar:
  http://ce.azc.uam.mx/profesores/franz/omi/monticulo.html
  En esta presentación se ilustra el funcionamiento de un montículo:
  http://www.olimpiadadeinformatica.org.mx/material%20de%20estudio/Presentaciones/Heap.swf
- Bosque de conjuntos. Un bósque de conjuntos es un bosque en el que cada árbol representa un conjunto, cada nodo apunta unicamente a su padre, y si no tiene padre apunta a sí mismo. Permite saber si dos elementos son del mismo conjunto y unir pares de conjuntos en tiempo O(log *n*)

Equivalente en Inglés de la Terminología

Término en Español	**Término en Inglés**
Estructura de Datos	Data Stucture
Pila	Strack
Cola	Queue
Lista enlazada o lista ligada	Linked List
Nodo ó vértice	Vertex
Árbol	Tree
Árbol binario	Binary Tree
Hoja	Leaf
Raíz	Root
Preórden	Preorder
Inórden	Inorder
Posorden	Postorder
Árbol binario de búsqueda	Binary Search Tree
Grafo ó gráfica	Graph
Grado	Degree
Camino	Path
Ciclo	Cycle

Grafo dirigido	Directed graph
Grafo no dirigido	Undirected graph
Grafo conexo	Connected graph
Bosque	Forest
Matríz de adyacencia	Adjacency matrix
Listas de adyacencia	Adjacency lists
Búsqueda en profundidad	Deph first search
Búsqueda en amplitud o búsqueda a lo ancho	Breath first search
Torneo	Tournament
Camino hamiltoniano	Hamiltonian path
Cola doble	Deque
Cola de prioridad	Priority queue
Lista circular	Circularly linked list
Lista doblemente enlazada	Doubly linked list
Árbol AVL	AVL tree
Montículo	Heap
Conjunto	Set

Fuentes

Cada problema que fue extraido de alguna fuente en específico, tiene la fuente anexada en la descripción.
Algunas imágenes fueron extraidas de www.algoritmia.net, y la mayoría de www.wikipedia.org.
Algunas definiciones fueron extraidas de IAN PARBERRY, Problems on Algorithms.