



UNIVERSIDAD DE COLIMA

Facultad de Telemática

Colima, col. 13 de Diciembre del 2019

Materia: Estructura de Datos

Catedrático: Jorge Rafael Gutiérrez Pulido

(“Memoria de apuntes”)

3D

Alumno:

- Cabellos Aguilar Juan Pablo

Contenido

1.- PRIMERA PARCIAL:

1.1.- Presentaciones sobre funciones de tiempos de complejidad..2

1.2.- Presentación sobre Omega.....3

1.3.- Presentaciones sobre algoritmos de ordenamiento.....3

2.- SEGUNDA PARCIAL:

2.1.- Estructuras de datos.....4

2.2.- Métodos de ordenamiento.....4

3.- TERCERA PARCIAL:

3.1.- Árboles.....5

3.2.- Grafos.....6

ANEXOS:

Anexo A.....8

Anexo B.....10

Anexo C.....11

1.- PRIMERA PARCIAL:

1.1.- Presentaciones sobre funciones de tiempos de complejidad:

Little o: Establece la cota inferior asintótica y la superior asintótica. Se usa para definir el crecimiento de las funciones.

Diferencia entre Big O y Little o:

En Big O existe una constante que sea mayor que la función. En little o, toda constante positiva es mayor a la función.

$f(n)$ se vuelve insignificante a medida que $g(n)$ se acerca a infinito.

Big O:

Es un tipo de notación para describir los grados de complejidad de ciertos algoritmos.

Tipos de notación:

- **Complejidad constante $O(1)$:**
No importa el input, siempre va a tardar lo mismo.
- **Complejidad lineal $O(n)$:**
Describe una complejidad que aumenta dependiendo del input de manera lineal.
- **Complejidad logarítmica $O(\log n)$:**
Depende de la cantidad de elementos de manera logarítmica.
- **Complejidad casi lineal $O(n \log n)$:**
Se parte en trozos dependiendo de las partes que tiene que analizar.
- **Complejidad cuadrática $O(n^2)$:**
Representa que el tiempo aumenta de forma exponencial.
- **Complejidad cúbica $O(n^3)$:**
Representa que el tiempo aumenta de manera exponencial y tiene una triple anidación de ciclos.
- **Complejidad factorial $O(n!)$:**
No se suele tomar en cuenta porque crece demasiado.

Theta:

Se usa para medir la cantidad de recursos que un algoritmo utiliza aunque es algo genérica.

¿Cómo funciona?:

Nos indica simultáneamente los límites inferior y superior de una función.

La notación asintótica es una función que representa a dónde quieres llegar sin llegar a esos límites, tiende a esos límites inferior o superiores pero sin llegar a esos límites.

El peor de los casos es dado por Big(O), que tarda más tiempo y el Omega(O) describe el mejor de los casos, que es la cantidad mínima de tiempo que toma si así acomodan los valores.

Omega en minúscula describe el promedio

Las notaciones matemáticas describen el cual es el mejor de los casos, el peor de los casos y el caso promedio.

Si en los ciclos tienes cosas que dependen de N entonces tomamos ese ciclo, si no depende de N no tiene porque aumentar el exponencial, si hay otro ciclo que termina en un número constante entonces es irrelevante y ese no se toma.

1.2.- Presentación sobre Omega:

Que tengan un $O(n^2)$ no significa que sea mayor a $O(n)$.

Lo que nos debemos de fijar en un algoritmo es su tasa de crecimiento, es su tasa de crecimiento, deberemos de eliminar las constantes menos significativas y quedarnos con aquellas que realmente importan.

1.3.- Presentaciones sobre algoritmos de ordenamiento:

Bucket sort:

Los elementos se ponen en buckets.

¿Cómo funciona?

Se inicializa un array, se van separando los arrays en cada uno de los buckets, se ordenan de manera individual.

Se hacen buckets de rangos de vectores de números y esos los ordenamos, después se ordenan individualmente por bucket, una vez que se termina de ordenar por buckets se sacan de los buckets por orden y ya están ordenados.

Counting sort:

Utiliza un método de conteo con vectores y vectores auxiliares $O(n + k)$ cuando k es el tamaño del vector auxiliar.

¿Cómo funciona?

Primero se averigua el intervalo, se crea un vector auxiliar con tantos números de intervalo, se crea un contador, se cuenta cuántas veces se repite cada elemento, después se hace una suma de lo que ya tenías con esas veces que los elementos se repiten, se colocan los números en la posición que el array de las sumas indiquen.

Una desventaja de esto es que va buscando por todo el array y encuentra el mínimo y lo va colocando desde la posición 0 hasta terminar, por lo tanto no es tan eficiente cuando se dan listas grandes.

2.- SEGUNDA PARCIAL:

2.1.- Estructuras de datos:

***Listas ligadas:**

- Valor
- Apuntador al siguiente valor

Lista doblemente ligada:

- Valor
- Apuntador al siguiente valor
- Apuntador al valor anterior

Lista doblemente ligada circular:

- Valor
- Apuntador al siguiente valor
- Apuntador al valor anterior
- El último nodo apunta al primero y viceversa

Para introducir un nuevo elemento se pone la referencia del nodo anterior a este elemento y este elemento apunta al que estaba en esta lista.

***Queue:**

Principio First In First Out.

Se utilizan 4 variables:

- Fila
- ArraySize
- Frente: First Element
- Posterior: The last element

Dos métodos:

- Agregar valores a la cola
- Eliminar valores de la cola.

***Stack:**

Estructura para guardar datos en forma LIFO (First In Last Out) para desplegar datos de forma cronológica de más a menos recientes

Métodos:

Push: Agrega un nuevo valor a la pila, ubicándolo al final de esta

Pop: Retorna el último valor ingresado a la pila, sacándolo de esta.

Peek: Retorna el valor del inicio del stack.

2.2.- Métodos de ordenamiento:

Heapsort:

Método de ordenamiento basado en comparación en forma de montículos.

Utiliza las propiedades de los montículos binarios.

No utiliza memoria adicional.

Quicksort:

Es rápido basado en la técnica divide y vencerás, originalmente recursivo, los algoritmos recursivos en general son más lentos.

Requiere pocos recursos en comparación a otros métodos.

No es bueno para cuando se le agregan nuevos números.

La función para calcular la altura de tu árbol es: " $h = \log_b n$ ", donde h representa la altura del árbol, b representa la base en la cual se está construyendo el árbol y n es la cantidad de nodos que el árbol va a tener.

De la misma manera existe una fórmula para calcular la cantidad de nodos que el árbol como tal va a tener, dicha función es: " $n = b^h$ ", donde n es la cantidad de nodos a calcular, b es la base en la cual se está construyendo el árbol y h los niveles que el árbol está teniendo.

Por último, hay una fórmula para conocer el esfuerzo individual de cada subnodo que se divide para cumplir con el enunciado de divide y vencerás, esta fórmula es: " $nF(e/b^h)$ ", donde n es la cantidad de nodos que estamos representando en ese momento, este puede cambiar según el nivel donde nos estemos posicionando. e es el esfuerzo total de el método o el árbol, por lo tanto lo que se está calculando es el esfuerzo de uno de sus subnodos que representa parte parcial de la cantidad total de esfuerzo. Por último está la b que representa la base en la que estamos construyendo el árbol, la cual está elevada a la h , por lo tanto representa el nivel en el que nos estamos posicionando.

3.- TERCERA PARCIAL:

3.1.- Árboles:

Es una estructura de datos que cuenta con un nodo raíz o padre, el cual va a tener más nodos hijos. Las leaves o hojas son los nodos que se encuentran en el punto más bajo del árbol y que no cuentan con hijos.

El height o altura se refiere a la cantidad de niveles que tiene el árbol.

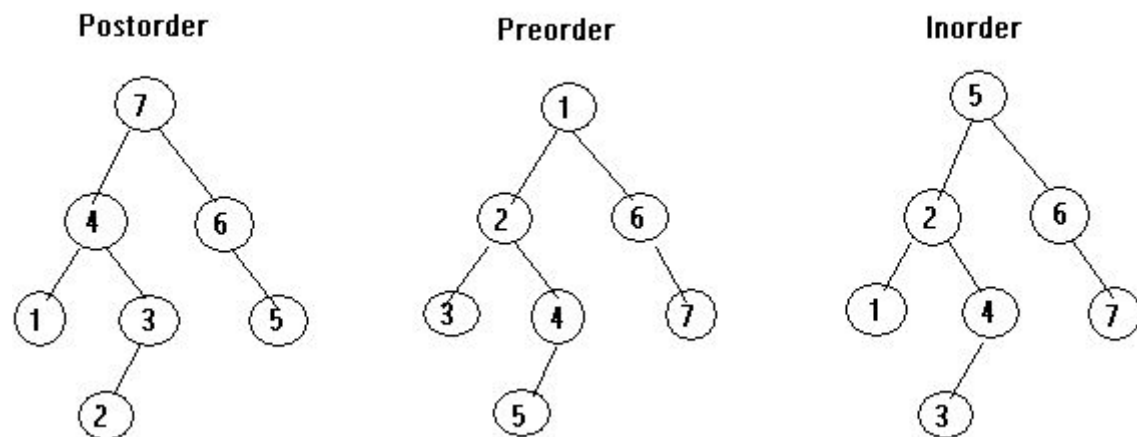
El nivel del árbol determinará cuántos nodos van a tener los hijos de cada nivel, ya que se tiene que poner como exponente el nivel del árbol, y " n " sería la base en la que está basada el árbol, así sabremos cuántos nodos hijos tendrán los nodos padre a partir de el nodo raíz. El nivel de profundidad de la raíz siempre será 0.

Tree transversals:

Estas son las formas de recorrer árboles binarios. El "pre", "in", "post" nos indica la posición en donde está colocada la N .

Un árbol binario es aquel que debe cumplir con que cada nodo tiene que tener una cantidad máxima de dos hijos. Lo ideal es que este árbol esté balanceado.

Balanceado: Significa que tenga la misma cantidad de nodos a través de los niveles del árbol, que no haya demasiados nodos cargados a la izquierda o a la derecha del árbol.



Splay tree:

El propósito de este árbol es mejorar la eficiencia en los datos que el usuario accede constantemente.

Árboles 2-3-4:

Es una extensión del árbol 2-3.

En todas sus operaciones tiene un tiempo de $O(\log N)$

Es muy bueno para la búsqueda pero la implementación del mismo puede llegar a ser algo bastante complejo.

Red-Black Tree:

Los nodos tienen un atributo los cuales tienen una propiedad llamada color.

Cada nodo debe ser rojo o negro.

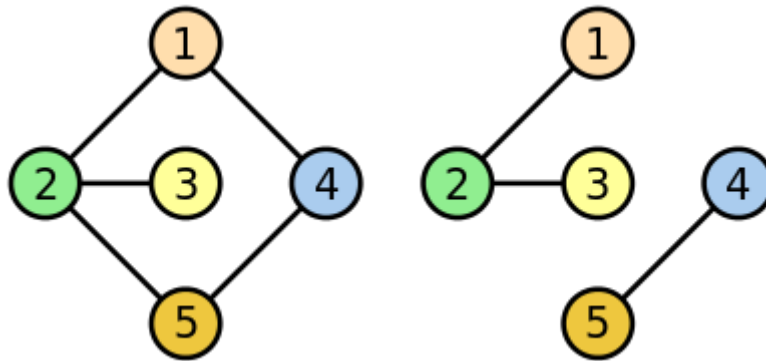
Siempre tiene como complejidad $O(\log n)$ para sus operaciones.

Operaciones:

- Rotación
- Almacena una referencia
- Elige pivote
- Se actualizan las referencias de los padres.

3.2.- Grafos:

Un grafo es una estructura de datos no lineal que consiste de nodos y líneas. Los nodos están a veces referenciados a los vértices mediante una línea o arco que conecta cualquiera de los dos nodos de un grafo.



Los **grafos completos** son aquellos que los nodos tienen una conexión la cual está ligada al resto. Este tiene el máximo número de aristas posibles.

Hay grafos que son **dirigidos** y **no dirigidos**, los dirigidos se refieren a aquellos que tienen una dirección, mientras que los no dirigidos no la tienen.

Para representar los grafos con matrices se toma que cada fila es un nodo, y los 0 representan que no hay conexión, mientras que los 1 significan que si la hay con el número de nodo, este número de nodo se representa por la columna.

Para que haya un grafo se debe cumplir que:

- Una liga no se apunte a sí misma
- Que no haya más de una liga para unir los mismos nodos.

Si no cumple con alguna de estas, entonces es un **multigrafo**.

Para saber si es un subgrafo de otro tenemos que considerar dos cosas:

- Que los nodos sean iguales.
- Que todos los pares del conjunto estén presentes en el otro grafo más grande.

***Matrices de adyacencia:**

Una matriz de adyacencia es definida cuando solamente se representan las conexiones entre vértices en un grafo, es decir, cuáles vértices están conectados con cuales. Sus características son:

- Es cuadrada
- Es simétrica

Un ejemplo sería el siguiente:

Matriz de adyacencia

	a	b	c	d	grado
a	0	1	1	0	= 2
b	1	0	1	2	= 4
c	1	1	0	1	= 3
d	0	2	1	0	= 3

Es la matriz $|V| \times |V|$ cuyas entradas A_{ij} cuentan el número de aristas que unen v_i con v_j .

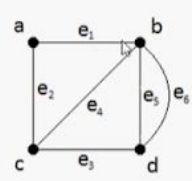
*Matrices de incidencia:

Una matriz de incidencia es cuando se señala el número de conexiones que un vértice tiene en general, es decir, cuando un vértice en un grafo tenga una conexión, no importa a qué otro vértice se conecte, se marcará un uno en la matriz que lo represente. Sus características son:

- Nunca es cuadrada
- Nunca es simétrica

Ejemplo:

Matriz de incidencia



Es la matriz $|V| \times |E|$ con entradas:
0 si e_i no es incidente con v_i
1 si e_i es incidente con v_i

	e_1	e_2	e_3	e_4	e_5	e_6
a	1	1	0	0	0	0
b	1	0	0	1	1	1
c	0	1	1	1	0	0
d	0	0	1	0	1	1

ANEXOS

ANEXO A:

Ordenamiento de un vector:

Let vect = [3, 5, 8, 1, 9]

Dado el array de arriba, tenemos que proceder a ordenarlo de una manera manual. La idea principal de este array es ir tomando elementos de a dos para ir comparándolos e intercalándolos de ser necesario para vida de que el array nos quede ordenado.

Una manera sencilla de hacer esto es la siguiente.

Lo primero que vamos a hacer es establecer un elemento que recorra nuestro array desde el primer elemento hasta el penúltimo elemento.

for($i=0$; $i < n-1$; $i++$){} = tomando “i” como el elemento que va a recorrer el array y “n” como el último de los dígitos que se introdujeron en el array.

Ahora lo que haremos será un bucle interno que realice un recorrido del vector desde el elemento “i+1” hasta el elemento n.

for($s = i+1$; $s < n$; $s++$){} = aquí se ve que por cada iteración del bucle externo se van reduciendo cada vez más los números a comparar, ya que estos debieron de haber sido comparados en iteraciones anteriores.

Ahora, lo que se quiere es ordenar todos los elementos, por lo cual se va a comparar el elemento "i" con los elementos siguientes del vector, por lo cual irá intercalándolos cuando sea mayor que un elemento que se encuentre en la parte inferior del vector.

Al final al código le agregué una validación para que comprobara que solamente se estaban introduciendo valores numéricos en mi array, ya que con el método `isNaN` se introduce dentro de un ciclo, el cual recorre el array elemento por elemento para comprobar que sean números, en el primer elemento no numérico que encuentre se saldrá del bucle y lanzará un error al usuario para indicarle que no puede introducir valores no numéricos dentro de un array de números. De no contener valores no numéricos dentro de mi array, entonces lanza un `true`, el cual voy a evaluar en una condicional para que continúe con el proceso de ordenamiento de mi array.

El código quedaría algo así:

```
function sort(array) {
  let numerico = false;
  for (let j = 0; j < array.length; j++) {
    let element = array[j];
    if (!isNaN(element)) {
      numerico = true;
    }
    else{
      numerico = false;
      break;
    }
  }
  if (numerico == true) {
    for(i=0; i < array.length-1; i++){
      for(s = i+1; s < array.length; s++){
        if(array[i] > array[s]){
          aux = array[i];
          array[i] = array[s];
          array[s] = aux;
        }
      }
    }
  }
}
```

```

        console.log(array)
    }
    else{
        console.error("Sólo se permiten valores numericos en el
array");
    }
}

let array=[3, 5, 8, 1, 2];
sort(array);

```

ANEXO B:

Representación de matriz adyacente en forma de lista:

```

a={data: 'a'};
b={data: 'b'};
c={data: 'c'};
d={data: 'd'};

a.l=[];
b.l=[];
c.l=[];
d.l=[];

a.l.push(a);
a.l.push(b);
a.l.push(c);
b.l.push(a);
b.l.push(c);
b.l.push(d);
b.l.push(d);
c.l.push(a);
c.l.push(b);
c.l.push(d);
d.l.push(b);
d.l.push(b);
d.l.push(c);

```

Es este código de una matriz adyacente, se puede alcanzar a apreciar que un vértice se apunta a sí mismo cuando corremos el código “a.l” en la consola. Vemos

que este vértice tiene dentro de su array de conexiones a otros vértices una conexión con sí mismo.

También podemos ver que hay una doble liga en nuestro código, cuando ejecutamos “b.l” ó “d.l”, podremos ver que en cualquiera de los dos, la “b” o la “d” están repetidas, eso indica una doble conexión.

ANEXO C:

Representación de una matriz incidente en forma de lista:

```
a={data: 'a'};  
b={data: 'b'};  
c={data: 'c'};  
d={data: 'd'};  
e0={data: 'e0'};  
e1={data: 'e1'};  
e2={data: 'e2'};  
e3={data: 'e3'};  
e4={data: 'e4'};  
e5={data: 'e5'};  
e6={data: 'e6'};  
  
a.l=[];  
b.l=[];  
c.l=[];  
d.l=[];  
  
a.l.push(e0);  
a.l.push(e1);  
a.l.push(e2);  
b.l.push(e1);  
b.l.push(e4);  
b.l.push(e5);  
b.l.push(e6);  
c.l.push(e2);  
c.l.push(e3);  
c.l.push(e4);  
d.l.push(e3);  
d.l.push(e5);  
d.l.push(e6);
```

Para comprobar que hay un nodo apuntando a sí mismo, corremos el código “a.l”, veremos que el vértice “a” cuenta con el enlace “e0”, el cual indica que apunta a sí mismo, ya que ningún otro vértice cuenta con este enlace.

También podemos ver que tenemos una doble conexión cuando corremos en conjunto el código “b.l” y “d.l” los cuales tienen repetida la conexión “e5” y “e6”. Esto indica que tienen una doble conexión.