

GENERICIDAD Definir el comportamiento de una entidad independiente de los datos que maneja.

Genericidad con comparable

```
public class MiClase <T extends Comparable>
public class MiClase <T extends Comparable<T>>
```

} Comparable.

```
public class Estudiante implements Comparable<Estudiante>
```

} ComparableTo

Genericidad con comparator

```
public static class ComparadorEstPorNombre implements Comparator<Estudiante>
```

```
public class ListaOrdenada <T extends Comparable<T>>
```

```
{ public boolean agregar(T pElem, Comparator<T> comparador)
{
}
}
```

```
public class Curso
```

```
{
```

```
    Estudiante.ComparadorXNombre compx = new Estudiante.ComparadorXnombre();
```

```
}
```

ANÁLISIS DE ALGORITMOS

Operaciones primitivas.

Declaración de variables.

Asignación de valores.

Comparación ints.

Acceso a un arreglo.

longitud de un arreglo.

Operaciones comunes

Sumar dos ints.

Busqueda binaria

Recorrido Sencillo

MergeSort

Doble Recorrido

Triple Recorrido

Busqueda exhaustiva.

NanoSec.

K_1

K_2

K_3

K_4

K_5

OOG.

K

$\log N$

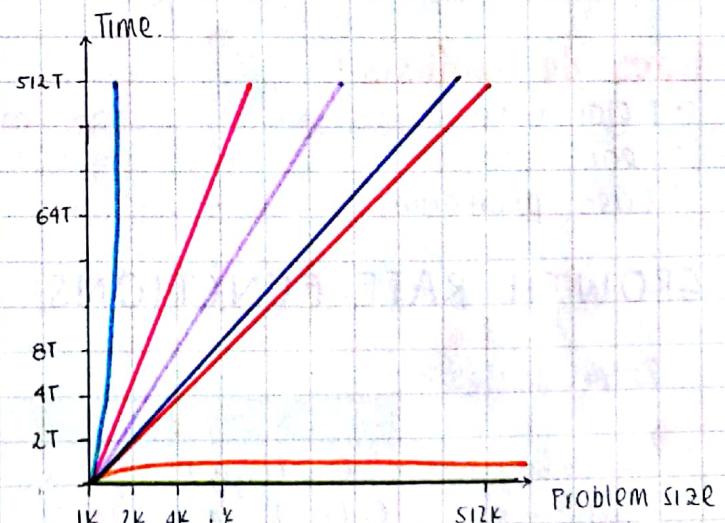
lineal

$N \log N$

N^2

N^3

2^N



Constante

logarítmica

lineal

linearitídica

Exponencial

Cubic

Quadratic

K

$\log N$

N

$N \log N$

N^3

N^2

Busqueda Binaria:

Código:

```
public static int busquedaBinaria( int[ ] arreglo , int elem ).
```

```
{  
    int inicio = 0 ;  
    int final = arreglo.length() - 1 ;  
  
    while ( inicio <= final )  
    {  
        int mitad = ( inicio + ( final - inicio ) ) / 2  
        if ( elem < arreglo [ mitad ] )  
            final = mitad - 1 ;  
        else if ( elem > arreglo [ mitad ] )  
            inicio = mitad + 1 ;  
        else  
        {  
            return mitad ;  
        }  
    }  
    return -1 ;  
}
```

Ojo: arreglo debe estar organizado, logn.

Típos de análisis:

- Mejor caso.
- Peor caso.
- Caso promedio

GROWTH RATE FUNCTIONS

Propiedades :

$$\begin{aligned} 23\log(n) &= O(\log n) \\ n^2+n &= O(n^2) \\ (3n+1) * (2n+\log n) &= O(n) * O(n) = O(n \cdot n) = O(n^2). \end{aligned}$$

(2n log n + 10n + 7log n + 40 es mejor que $\sum n \log n + 2n + 10 \log n + 1$.
2n log n + O(n) es mejor que $\sum n \log n + O(n)$.

- Porque $2n < \sum n$.

Notita : Busqueda lineal vs. binaria.
BC AC
Lineal 1 n
Binaria 1 $\log n$

Notita: Selection sort vs Insertion sort

	Selection	sort	vs	Insertion	sort
BC				WC	
Selection	n^2			n^2	
Insertion	n			n^2	

BOLSAS, COLAS Y PILAS

Pila:



Cola:



Implementar Pila: linked list.



Array.



Hacer crecer un arreglo con el doble de espacios

Resizing Pila, implementando arreglo.

BC AC WC

Construcción

Push.

Pop.

Size.

1 2 3 4 5 6 7 8 9 10

¿Qué es mejor? Implementar pilas con linkedlist o con un arreglo de tamaño fijo?

linked list: $O(1)$ peor caso.

Extra espacio y extra memoria.

Arreglo: $O(k)$ caso promedio.

Menos espacio gastado.

Implementar Cola: UnorderedList.



FIFO.

Array.



DJIKSTRA TWO STACKS ALGORITHM

• null

Value

Operator



ITERATOR

HasNext() Boolean

Next() E

```
public class IteradorLista < T extends Comparable < T > implements Iterador < T >
{
    IteradorLista (Nodo < T > primero)
    {
        proximo = primero;
        boolean hasNext();
        E next();
    }
}
```

ELEMENTARY SORT ALGORITHMS

Insertion:

7	10	5	3	8	4	2	9	6
j								

Mejor caso: Casi ordenados

7	10	5	3	8	4	2	9	6
j								

1. Casi ordenados
2. Few unique
3. Random
4. Reverse.

7	10	5	3	8	4	2	9	6
j								

Poor caso: Reverse

7	5	10	3	8	4	2	9	6
j	i							

- Estable.
- $O(n^2)$ en el peor caso
- $O(n)$ cuando están casi ordenados.
- Muy lento en reverse.

5	7	10	3	8	4	2	9	6
j	i							

No visto todo aún

Selection Sort:

7	10	5	3	8	4	2	9	6
i								

Mejor caso: Cualquiera, se demora lo mismo.

2	3	5	10	8	4	7	9	6
i								

• No estable.

2	10	5	3	8	4	7	9	6
i								

• $O(n^2)$ comparaciones.

2	10	5	3	8	4	7	9	6
i								

• $O(n)$ swaps.

2	3	5	10	8	4	7	9	6
i								

• No adaptativo.

H-Sort demo.

6	7	5	3	4	10	2	9	8
j	i							

3	7	5	6	4	10	2	9	8
j	i							

3	7	5	6	4	10	2	9	8
j	i							

3	4	5	6	7	10	2	9	8
j	i							

3	4	5	6	7	10	2	9	8
j	i							

3	4	5	6	7	10	2	9	8
j	i							

3	4	5	2	7	10	6	9	8
j	i							

3	4	5	3	7	10	6	9	8
j	i							

Sea $h = 3$.

Organización parcial

Depende del h escogido respecto a la cantidad de datos.

$$h(i) = \begin{cases} 1 & \text{si } i=1, \\ 3h(i-1) + 1 & \text{si } i \geq 2. \end{cases}$$

Mejor caso: Casi ordenados.

1. Casi ordenados.
2. Reversed
3. Few unique
4. Random.

Pior caso: Random

- No estable.
- $O(n^{3/2})$
- $O(n \log n)$ si estan casi ordenados.

Elementary sorting algorithms

Selection

Insertion Sort

h-Sort ($3x+1$)

goal

BC

N^2

N

$N \log N$

N

AC

N^2

N^2

?

N

WC

N^2

N^2

$N^{3/2}$

$N \log N$

$N \log N$

Shuffle Sort

2	3	4	5	6	7	8	9	10
---	---	---	---	---	---	---	---	----

Shuffle

8	6	9	7	2	4	10	5	3
---	---	---	---	---	---	----	---	---

Merge Sort

- Dividir el arreglo en dos
- Ordenar recursivamente cada pedazo
- Mezclar ambas partes

Insertion (N^2)

Thousand

Instant

Billion

317 años

Million

2.8 horas

MergeSort ($N \log N$)

instant

18 min.

1 sec.

a[1]	E	E	G	M	R	A	C	E	R	T
lo	mrd	mid+1	hi							

Copiar a auxiliar.

Mejor caso: Funciona igual en todos.

aux[1]	E	E	G	M	R	A	C	E	R	T
--------	---	---	---	---	---	---	---	---	---	---

a[1]	E	E	G	M	R	A	C	E	R	T
------	---	---	---	---	---	---	---	---	---	---

Compara el mínimo en cada mitad.

E	E	G	M	R	A	C	E	R	T
i		j							

E	E	G	M	R	A	C	E	R	T

Cambia con K

E	E	G	M	R	A	C	E	R	T
i		j							

A	E	G	M	R	A	C	E	R	T
K									

E	E	G	M	R	A	C	E	R	T
i		j							

A	C	G	M	R	A	C	E	R	T
K									

QuickSort

- Shuffle the array.
- Partition tal que, para $a[j]$ está en su lugar.
- No larger entry to the left of j .
- No smaller entry to the right of j .

K	R	A	T	E	L	E	P	U	I	M	Q	C	X	O	S
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

E	C	A	I	E	K	L	P	U	T	M	Q	R	X	O	S
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	C	E	E	I	K	L	P	U	T	M	Q	R	X	O	S
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

A	C	E	E	I	K	L	M	O	P	Q	R	S	T	U	X
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Mejor caso: Random, nearly sorted, reversed.

Pior caso: Few unique.

- No estable
- $O(n^2)$ time.
- $O(n \log n)$ time.
- No adaptative

Quick Sort 3 way.

	No estable.	$O(\log n)$ extra space.	$O(n^2)$ time, tipicamente $O(\log n)$.	Adaptativo
Selección	✓	✓	✓	Adap. In place?
Insertion	✓	✓	✓	Stable?
Shell	✓	✓	✓	Best
Merge				
TimSort				
Quick	✓	✓	✓	
3-way Quick	✓	✓	✓	

Mejor caso: Few unique
Poor caso: Random, nearly sorted, reversed.

	No + espacio	Respeto el orden.
Average	$\frac{1}{2}N^2$	$\frac{1}{2}N^2$
Worst	$\frac{1}{4}N^4$	$\frac{1}{2}N^2$
Notas	$N \log_3 N$	$cN^{3/2}$
	$\frac{1}{2}N \log N$	$N \log N$
	$N \log N$	$N \log N$
	$2N \log N$	$2N \log N$
	N	$\frac{1}{2}N^2$

	Worst case	Average case	Worst case
Selección	Shell, insertion, bubble	Shell, insertion, bubble	Selection
Insertion	Bubble, shell, quick, quick3.	Bubble, shell, quick, quick3.	Selection
Shell	Merge, quick, quick3, insertion	Merge, quick, quick3, insertion	Selection
Quick	Shell, insertion, merge, quick, bubble	Shell, insertion, merge, quick, bubble	Selection
3-way Quick			

$$\sim f(N) \quad \frac{f(N)}{f(N)} \Rightarrow 1 \quad \text{as } N \text{ grows}$$

Adaptativo

RESUMEN SEGUNDO PARCIAL DATOS

Binary Heap

- › Si el último nivel NO esta lleno, lo que tenga debe estar mas a la izquierda. El resto DEBE ESTAR LLENO.
- Si el hijo está en k , su papá en $\lceil k/2 \rceil$
- PARA UN PADRE EN k , SUS HIJOS DEBEN ESTAR EN $2k$ Y $2k+1$**

- › Insertar un elemento

- Arbol: $\log N$
- Arreglo: $O(1)$

- › Swim

- De abajo para arriba

- › Sink

- De arriba para abajo.

LOG₂N

LOG₂N

2LOG₂N

Menos eficiente.

Operaciones:

- Insertion:

- › Cada vez que metemos algo, miramos que satisface la propiedad min heap.
- Comparamos cada nuevo con su papá y cambiamos si necesario.
- La raíz en cada subtree es la mayor de las otras llaves presentes. Y así hacia arriba.

- Borrar:

- La raíz, lo cambio con el ultimo, y reuso como insertion otra vez.

HeapSort

- › Construimos el Heap
- › Lo volvemos max heap.
- › Cambiamos la raíz y el ultimo, eliminamos la ultima (raíz).
- › Volvemos a hacer max heap.

Symbol Tables

- ? Llave valor.
- › Busco por llave.

```
public class ST <Key, value>
```

```
ST().
```

```
void put(Key key, Value val) a [key] = val.
```

```
Value get(Key key) a [key].
```

```
boolean contains(Key key)
```

```
void delete(Key key)
```

Convenção Valores no son nulos, get() devuelve null si no encuen-

tra la llave, método put sobreescribe el valor.

Asuma:

- Keys are comparable.
- Comparamos con compareTo() ó equals().
- $(x == y)$ **X**

Buscar: Busca por todas las llaves.

Insertar: Busca, si no está, la añade al frente.

	Garantizado:		Promedio:		
	Search	Insert	Search	Insert	
Busqueda en lista desordenada.	N	N	$\frac{N}{2}$	N	equals()
Busqueda binaria. Arreglo ordenado.	log N	N	log N	$\frac{N}{2}$	compareTo()
		Delete		Delete. ($N/2$) en ambos.	

HASH

hashCode() retorna un int de 32-bits.

Forma correcta 100% NO FAKE

```
private int hashCode(Key key){  
    return (key.hashCode() & 0x7FFFFFFF) % M;  
}
```

¿Cuánto cuesta buscar? $O(k) = \frac{N}{m} m$ (capacidad).

Factor de carga $\frac{\# \text{ocupados}}{\# \text{totales}} \leq 0,75$ ¿Se pasa? resize().

Separate chaining Cada pos tiene una lista, por si se repite módulo.

Linear probing Cuando se coincide en módulo, se encuentra el siguiente espacio libre y se pone ahí.

Buscar: Busca donde debería estar, si no, tratar en $i+1, i+2\dots$

RESIZE:

$\frac{N}{M} \cdot \frac{1}{2}$ Duplica. $\frac{N}{M} \leq \frac{1}{8}$ Quite la mitad.

GARANTIZADO

Implementación

Buscar Insertar. Borrar.

Binary Search Trees

Sequential search
lista en desorden.

N

N

N

- O está vacío o tiene dos sub-arboles
- Si es menor: Izquierda.
- Si es mayor: Derecha.

Binaria. Arreglo
en orden.

$\log N$

N

N

- Nodo = (Llave, Valor, referencia al sub-árbol izquierdo y derecho).

Binary SearchTrees.

N

N

N

- la forma del árbol depende del orden de inserción.

BinarySearchTrees
Rojo-Negro

$2\log N$

$2\log N$

$2\log N$

2-3 tree

$\log N + 1$

$\log N$

$\log N$

SeparateChaining.

N

N

N

- Datos random:

- Insertar: $2\ln N$.
- Buscar: $2\ln N$.

linear Probing

N

N

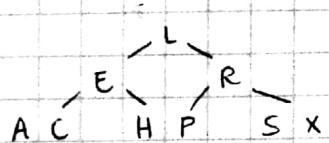
N

Balanced Search Trees

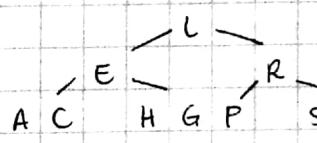
2-3 Tree

2-node One Key → Two children.

3-node Two Key → Three children.



Insertar G



Peor caso cuando todos son tipo 2:

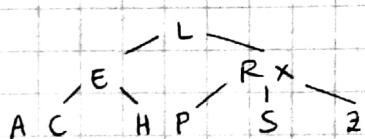
Altura $\log N$.

Mejor caso cuando todos son tipo 3:

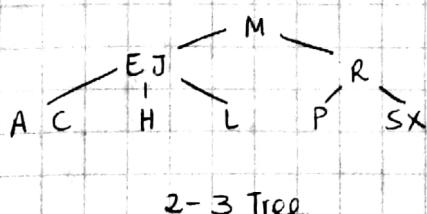
Altura: $\log_3 N$

Insertar Z.

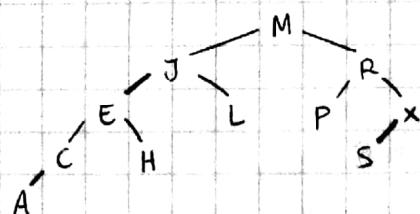
Cada vez que subo, parto en 2.



Red-black Trees



2-3 Tree



Black-red.

Los rojos SIEMPRE a la izquierda.

Rotar izquierda

Rotar derecha.

Siempre se mete como rojo.

Mejor caso: Todos son tipo 2,
no hay rojo.

Altura: $\log N$.

Peor caso: Todos son tipo 3,

Altura: $2 \log N$

REPASO PARCIAL 3

UNDIRECTED GRAPHS

MÉTODOS DE GRAPH

```
Graph (int V)
void addEdge (int v, int w)
Iterable <Int> add (int v)
int V()
int E()
```

New graph with V vertices.
add edge v-w.
vertices adjacent to v.
de vertices
de arcos.

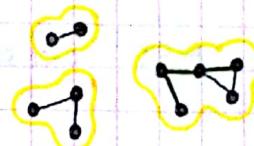
DEFINICIONES

Grafo: Grupo de vértices conectados por arcos.

Camino: Secuencia de vértices conectados por arcos.

Ciclo: Camino "redondo".

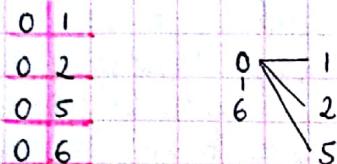
Grado:  > Vértice de grado 3.



■ Componentes conectadas.
■ Camino de longitud 4.

GRAPH REPRESENTATION

- Tener una lista de arcos (ó array).



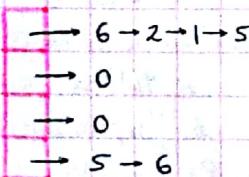
- Tener una matriz de adyacencia.

1 = están conectados.

0 = NO están conectados

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

- Tener un arreglo de listas.



In real life: Se tiene un arreglo de listas.

COMPONENTES CONECTADAS

- Iniciar vértices como no marcados
- Para cada vértice no marcado, hacer DFS para identificar a todos los vértices que hacen parte de una misma componente.

DFS: DEPTH-FIRST SEARCH

- Unroll a ball of string behind you.
- Mark each visited intersection and each visited passage.
- Retrace steps when no unvisited options.
- Mark v as visited
- Recursively visit all unmarked vertices adjacent to v.

marked []

edgeTo []

BFS: BREADTH-FIRST SEARCH

- Repeat until queue is empty.
 - Remove vertex v from queue.
 - Add to queue all unmarked vertices adjacent to v and mark them.

- Examina en orden increasing distance from s.

BFS

DFS

TIME

Camino más corto entre s y t.
Camino entre s y t.
Componentes conectadas

✓

✓

✓

✓

✓

✓

E + V

E + V

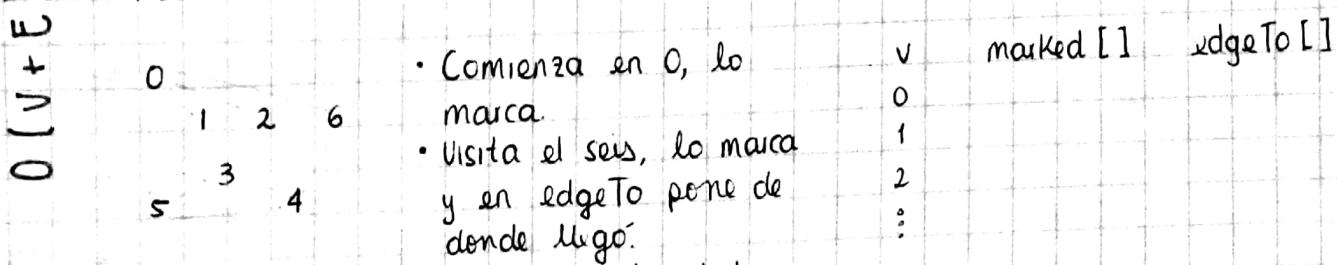
E + V

Paula Velandia 201715353

ALGORITMOS

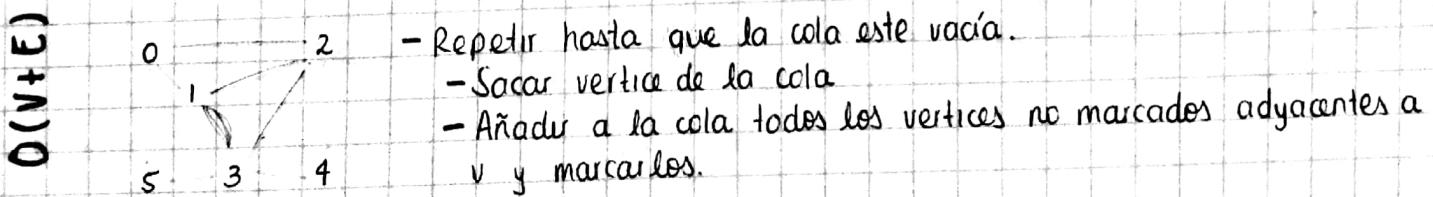
DFS: (RECURSIVO) REVISA CONECTIVIDAD.

- Marca vértice v como visitado
- Recursivamente visita los vértices no marcados adyacentes a v .



- Visita el cuatro, visita el cinco, visita al tres.
- El tres propone al 4, pero ahí ya se llegó desde el seis.
- Se devuelve al cinco.
- El cinco propone al cero, pero el cero ya está marcado.
- Se devuelve al cinco, se devuelve al cuatro, el cuatro propone al tres pero ya lo visitaron.
- Se devuelve al seis, se devuelve al cero.
- El cero todavía puede proponer al dos, lo propone y lo marca.
- Igual al uno.

BFS: (COLA) CAMINOS MÁS CORTOS (MENOR NÚMERO DE ARCOS)



- Añade 0 a la cola
- Visita al dos y saca al 0 de la cola.
- Mete al dos en la cola y vuelve al 0.
- Mete al uno en la cola, mete al cinco en la cola.
- Saca el dos y mira a sus adyacentes. El dos propone al uno pero ya lo marcó el cero
- Propone el tres y lo mete en la cola
- Propone el cuatro y lo mete.
- Desencola el que tocó y mira si propone algo nuevo.

v $edgeTo[]$ $distTo[]$

Cola

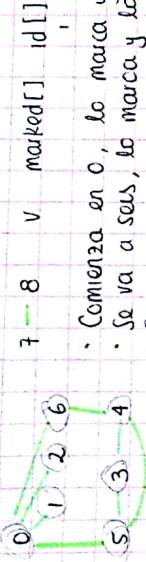
0 -> 1 2 3
0

COMPONENTES CONECTADAS

- Para visitar un vértice v

- Marcar vértice v como visitado.

- Recursivamente visitar todos los vértices adyacentes a v .



$O(|V| + E)$

MÉJICR Y PECP CASO.

- Comienza en 0, lo marca y lo colorea.
- Se va a seis, lo marca y lo colorea del mismo color.
- Se va a 4, a 5, a 3; 3 propone a 4 pero ya está, se devuelve a 5, 5 propone a 0
- Se devuelve a 4 a 6 y a 0. Colorea.
- De ahi a 2 y de 0 a 1. Colorea.
- Llega a 7 y hace lo mismo en un color diferente.

GREEDY MST

PUEDE DAR VARIAS SOLUCIONES
Comienza con todos los vértices coloreados de gris.



Se queda con los que van de un color a otro, ordenadas por peso.

- Meje 0-2 (primero en el corte por peso) al MST.
- Meje 5-7
- Meje 6-2

Meje 7-0.

Meje 3-2.

Meje 1-7

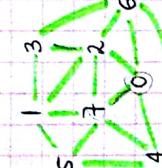
Meje 5-4

Solución: $\langle 0,2 \rangle, \langle 5,7 \rangle, \langle 6,2 \rangle, \langle 7,0 \rangle, \langle 3,2 \rangle, \langle 1,7 \rangle, \langle 5,4 \rangle$

KRUSKAL

Arces en orden ascendente de peso.

Añada al siguiente arco al arbol T amenes que hacerlo crea un ciclo.



Comienza en 0-7 porque es el de menor peso.

No crea ciclo, lo agrega.

Sigue en orden de pesos, con 2-3. No crea ciclo, lo agrega.

Sigue 1-7, no crea ciclo. Sigue 0-2, no crea ciclo.

Sigue 5-3, no crea ciclo.

Sigue 1-3, peso crea ciclo, no lo pone mas. y seguimos evaluando.
4-5 no crea. 1-2, crea. 6-2 no crea y completa el MST.

PRIM LAZY

Comienza en 0 y meje sus vecinos ordenados por peso.

Se va por el vecino menor, lo borra de la "lista" y lo añade al MST.

Hace lo mismo con el vecino (reordena sus vecinos).

Se va por el menor y agrega.

Repite otra vez.

Agrega, reordena, atienda.

All que saque, le muco necesita.

ESTRUCTURAS DE DATOS

Profesor: Mario Fernando de la Rosa.

Porcentajes: 3 exámenes 15% clu.

3 proyectos 15% clu.

Talleres 10%

GENERICIDAD.

- Definir el comportamiento de una entidad independientemente de los datos que maneja.
- Favorece reutilización de clases e interfaces.

Ejemplo: ArrayList es genérico

< Definición de la clase genérica >

public class ArrayList <T>

< Buen uso >

ArrayList <String> nombres = new ArrayList <String>

< Mal uso >

nombres.add(new Integer(2)); Error compilación.

Genericidad con clase Object.

OBJETO

```
+ String toString()  
+ hashCode()  
+ boolean equals(Object obj)  
+ Object clone()
```

ARREGLO FLEXIBLE

```
- T [] elementos  
- int tamañoMax  
- int tamañoAct.  
...
```

T

```
+ ArregloFlexible (int mux)  
+ void agregarElemento (T dato)  
+ T [] darElementos()  
+ T buscarDato (T dato)  
...
```

- El tipo genérico T representa una clase **NO** puede ser un tipo primitivo, debe ser: Integer, float, Double, Boolean o char...

```
public class ArregloFlexible <T>  
{  
    private int tamañoMax;  
    private int tamañoAct;  
    private T elementos[];  
  
    public ArregloFlexible()  
    {
```

Elementos = (T[]) new Object[max];
tamañoMax = max;
tamañoAct = 0;

Este arreglo flexible solo se le pueden aplicar los métodos de la clase Object.

3

T buscar (T dato)

Solución: Se tienen que usar alguno de los métodos de la clase genérica T:

equals(), toString() o hashCode()

Si T no tiene el método se usa el de

La clase Object.

4

Genericidad con Comparable

public int compareTo (T obj)

Retorna 0 si son iguales

Retorna -1 si actual < parámetro

Retorna 1 si parámetro < actual.

• public class Clase <T extends Comparable>

La clase Clase es de tipo T que implementa Comparable donde T define el método int compareTo (Object obj).

• public class Clase <T extends Comparable <T>>

La clase Clase es de tipo T que implementa Comparable <T> donde T define el método int compareTo (T obj).

Digamos que T extiende Comparable <T>.

↳ Se define la clase genérica: Clase listaOrdenada <T extends Comparable <T>>.
↳ Entonces, existirán allí métodos tipo:

public boolean agregarElem (T elem);

Genericidad con Comparator

Más de una comparación

public int compare (T objL, T objD). Permite comparar dos objetos de tipos

objL y objD.

• Si crea un comparador para cada tipo diferente de consulta

Clase

Al artículo se puso

comparador. Lo

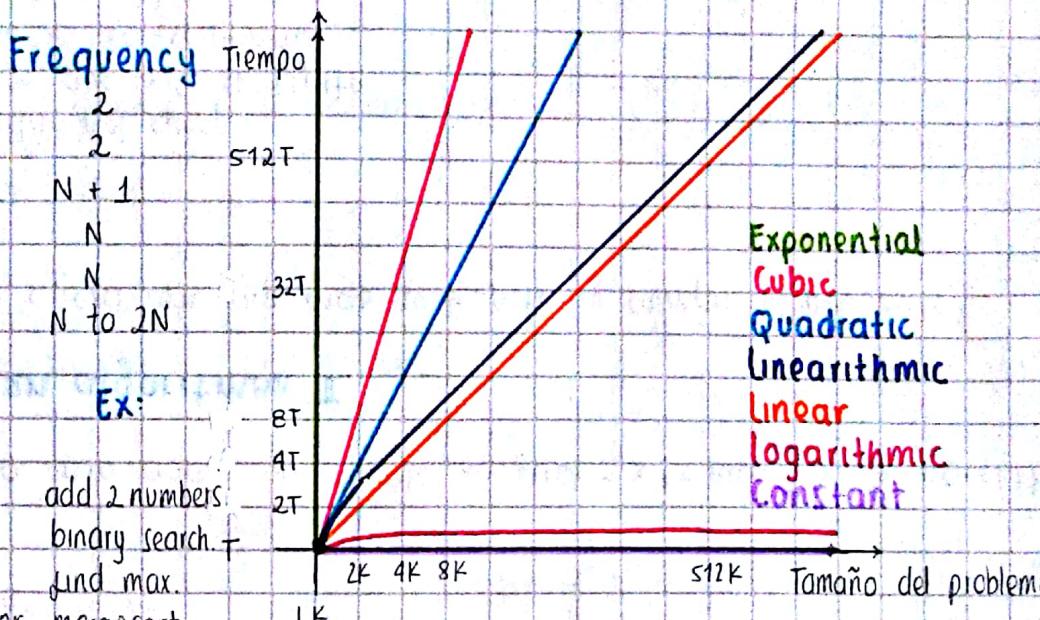
se puso

para

Analysis of Algorithms.

- How many instructions as a function of input size N?

Operation	Frequency
variable declaration	2
assignment statement	2
less than compare	$N + 1$
equal to compare	N
array access	N
increment	$N \text{ to } 2N$



TIP: Try to determine best case, worst case and average case.

~Lectura Semana 1.

- Bag ~ public class Bag <Item>

Bag()

creates a new bag.

void add (Item item)

add an item.

boolean isEmpty()

is the bag empty?

int size()

of items in the bag.

- Queue ~ public class Queue <Item>

Queue()

creates a new bag

void enqueue (Item item)

add an item.

Item dequeue ()

is the bag empty?

boolean isEmpty()

is the queue empty?

int size()

of items in the bag.

- Stack ~ public class Stack <Item>

Stack()

creates a new stack

void push (Item item)

add an item

Item pop()

remove the most recently added item.

boolean isEmpty()

is the stack empty?

int size()

number of items in the stack

Java's wrappers ~ Boolean
Byte

Boolean
byte

Character	char
Double	double
Float	float
Integer	int
long	long
Short	short

Ex: Stack <Integer> stack = new Stack <Integer>();
 stack.push(17);
 int i = stack.pop();

auto boxing (int → Integer).
 auto un-boxing (Integer → int).

- API ~ public class Stopwatch()
 - Stopwatch() creates a new Stopwatch.
 - double elapsedTime() run elapsed time since creation.

Genericidad con Comparable II.

- Suponga ArrayList T extends Comparable <T>.

ArrayList <Carro> carros = new ArrayList <Carro>();

La clase Carro debe ser Comparable y no lo es, error de compilador.
 Por eso hay que declararla como Comparable.

```
class Carro implements Comparable <Carro>
{
  método que comparaPorPlaca();
}
```

Como implementa Comparable debería implementarse el método compareTo.

Lectura Semana 2

¿Cómo recorrer una lista enlazada? for (Node x = first ; x != null ; x = x.next())

Estimación de tiempo en algoritmos I.

- Siempre se considera la operación que más se repite y siempre el

Todo el ciclo:

Cuerpo ciclo:

Estimar la complejidad del cuerpo.
 Escoger la operación que más se
 cuenta ± operaciones seleccionada.

O (N (#Número operaciones cuerpo))

BINARY HEAP

Árbol binario: Cada nodo tiene una prioridad mayor o igual a la de sus hijos.

HEAP Todas las raíces están llenas, al último puede o no estarlo. Si el último tiene elementos, deben estar más a la izquierda. La persona que nunca se va a usar, ¿por qué?

Para que se cumpla.

Heap Binario:

- Para un padre en la posición k , sus hijos están en $2k$ y $2k+1$.



HEAP : SORT

1. Construir un heap binario.

#comparaciones

$O(2N)$

#intercambios

$O(N)$

- $\log_2 N$ Insertar un elemento (Insert(τ elem))
 - Donde primero se puebla
 - Árbol: $\log N$
 - Arreglo: $O(1)$

2. Orden ascendente.

TABLA SIMBOLOS

$2 \log_2 N$

MENOS EFICIENTE

- Se agrupan los elementos por tuplas.

- Llave

- Valor

- La llave corresponde al identificador ÚNICO para acceder a ese valor

$C = \{(U_1, V_1), \dots, (U_n, V_n)\}$ **No puede haber una llave con dos valores.**

CÓDIGO

```
public class ST < key, value >
```

```
{
```

```
    void put(key k, Value val) { Si no la contiene, la añade. Si la contiene, le
```

```
    value.get(key k) cambia al valor.
```

```
    boolean contains(key k)
```

```
    void delete(key k)
```

Por conveniencia

- Los valores no son nulos
- Método `get()`: Devuelve null si no hay llave.
- Método `put()`: Sobrescribe sobre el valor.

X $K = L_1$
✓ $K \in L_1$

SEQUENTIAL SEARCH IN A LINKED LIST

- Mantener una lista encadenada (desordenada) de pares llave - valor.
- Pasar por todas las llaves hasta encontrarla.
- Si no la encuentra, añade al principio.

Put: Peor caso $O(N)$.

Get: Peor caso $O(N)$.

Remove: Peor caso $O(N)$.

Find: ¿Dy si está ordenada?

Put: Peor caso: $O(N)$

Get: Peor caso: $O(\log_2 N)$.

Remove: Peor caso: $O(N)$.

Find: Caso particular: Tabla Hash.

- Siempre tiene que implementarse con un arreglo (Tupla: llave valor).
- Entra la llave y devuelve un valor.

TABLA HASH

0	1	2	3	4	5	6	7	

Arreglo / área Primaria.
Siendo M la capacidad

$M = 8$

$(K, V) \rightarrow \text{Hash}(K) \rightarrow [0, \dots, M-1]$

Siempre se busca por llave. Si el mundo fuera ideal, las búsquedas siempre serán $O(1)$.

c) Que hace Hash-Code? Retorna un entero de 32-bit (int).
Identificador numérico.

No es tan buena porque podría estarme devolviendo negativos o saliéndose.

$(k, v) \rightarrow \lfloor \text{HashCode}(k) \rfloor \% [0, \dots, M-1]$ ¿Cuál es el problema? Varios números pueden tener el mismo módulo.

¿Cuánto costaría buscar? $O(k) = \frac{N}{m} \rightarrow$ Si son 5'000.000 y 20.000 de capacidad, guardo 250 por casilla.

Fórmula correcta de hacer Hash

private int Hash (Key key).

{

 return (key.hashCode() * 0xFFFFFFFF) % M;

}

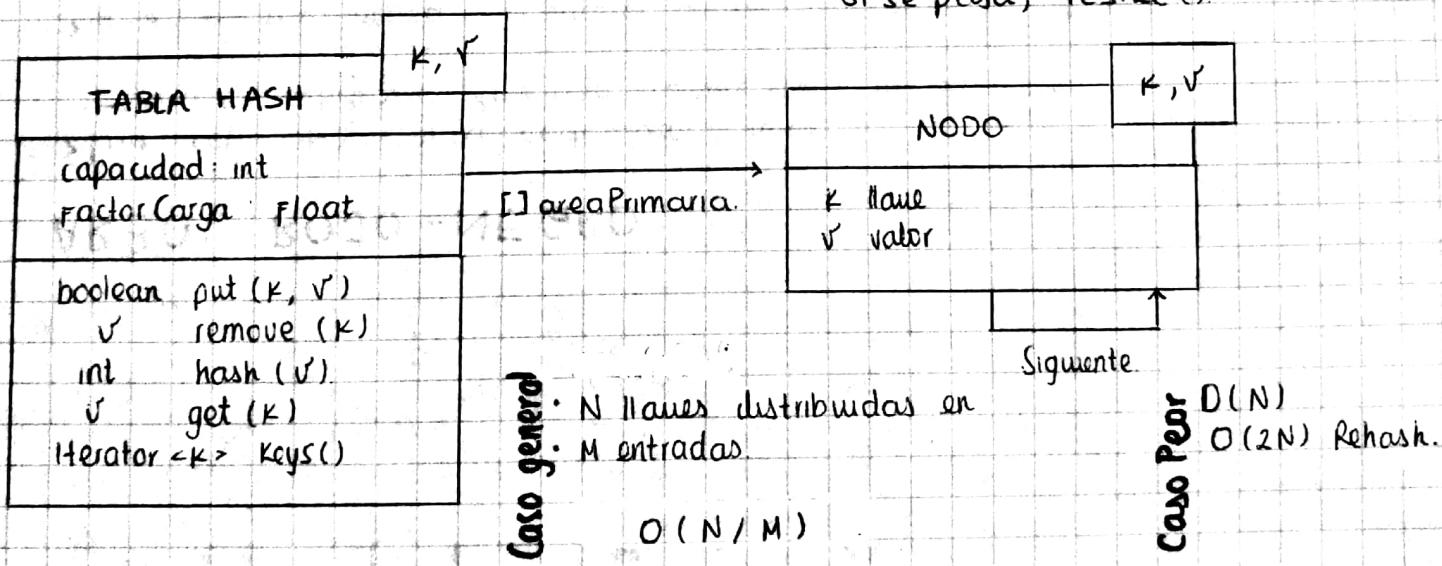
Colisiones Por eso pueden haber varias llaves, misma posición. ¿Cómo lo arregla?
En cada pos, hay una lista.

Factor de carga $\frac{\# \text{ entradas ocupadas}}{\# \text{ entradas totales}}$

¿Cuál es la idea?

Factor Carga: 0,75.

Si se pasa, resize().

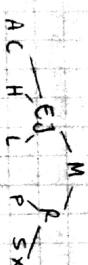


Revisar in order, ...

TREE SHAPE Estructura del árbol depende del cómo se insertan los elementos

2-node: One key, two children.

3-node: Two keys, three children



Balanciado por altura va altura de todas las ramas es la misma.

Implementación

Complejidad:	$O(\log_2 N)$
Altura:	Todos los nodos tipo 3.
Busqueda:	Todos los nodos tipo 2.

Algoritmo:

Post-order:

ARBOL ROJO - NEGRO

- > Link rojo: Representa nodos con 3 hijos.
- Descripción arbitraria: Link rojo siempre van a la izquierda.
- > 2-3 o rojo-negro: El mayor desbalance será de 1 nudo.
- > Representación nodo rojo: atributo de tipo booleano color;
Metodo isRed(Node x)
- > Insertar en un árbol rojo-negro.
- Durante operaciones, mantener
 - Orden simétrico
 - Balance
 - No recorriendo.

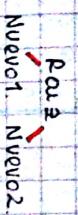
Operaciones elementales árboles rojo-negro.

1. Rotación a la izquierda.
Cambiar un link rojo de derecha a izquierda.
2. Rotación a la derecha
Cambiar un link rojo de izquierda temporalmente a la derecha.
3. Cambio de color
Hacer un split de un nodo.

$$B' \setminus A \setminus C \rightarrow B' \setminus A \setminus C$$

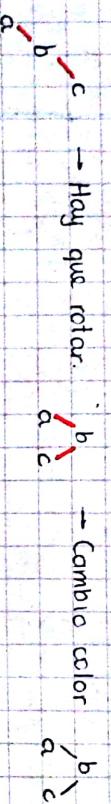
INSERCIÓN EN ARBOL ROJO NEGRO

► Caso 1: Insertar en la raiz.

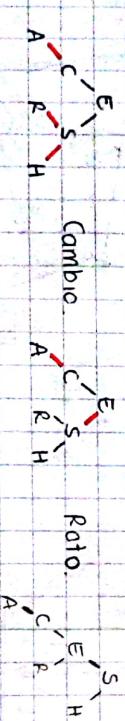


- Generación caso 1
- Insertar a árbol con nodo "tipo 2"
- Hacer inserción típica de BST.
- Verificar invariantes de color. (Si toca: Rotar, volver a que rotar...)

► Caso 2: Insertar a árbol con nodo tipo 3.

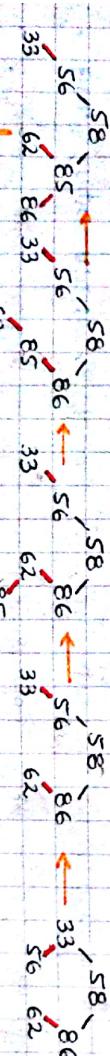
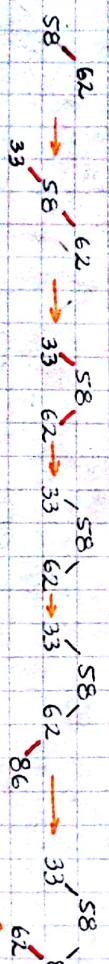


- Generación caso 2:
- Insertar en BST, colorear de rojo.
- Rotar para balancear el 4 nodo (si se genera)
- Cambiar colores para pasar Nuevo arriba (con rojo)
- Rotar a la izquierda si queda a la derecha.
- Repetir si es necesario.

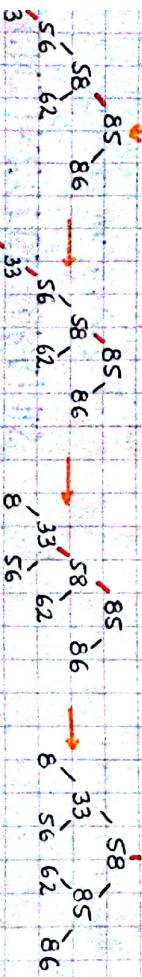


Ejercicio

62, 58, 33, 86, 56, 85, 8



DEBE QUEDAR BALANCEADO



UNDIRECTED GRAPHS

- » **¿Qué es un grafo?** Set of vertices connected pairwise by edges.
Un grafo no dirigido garantiza la bidireccionalidad.
Si hay de A a B, hay de B a A.

- » **Graph terminology**
 - Path, sequence of vertices connected by edges.
 - Grado de un vértice: Cantidad de conexiones.
 - Vértice: Un vértice inicial y final en el mismo.

- » **Graph representation:** Provides intuition about the structure of the graph.

`END`

`GND();`
`addVertex(k, v)`

`addEdge (k, k)`

Matriz adyacente: Simétrica.

	0	1	2	3	4	5	6	7	8
0	0	1	1	0	0	1	1	0	0
1	1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	0	1	1	0	0	0
4	0	0	0	0	1	0	0	0	0
5	0	0	0	0	0	1	0	0	0
6	0	0	0	0	0	0	1	0	0
7	0	0	0	0	0	0	0	1	0
8	0	0	0	0	0	0	0	0	1

1. Hay conexión
2. No hay conexión.

¿Qué mas puedo hacer?

4 → 5
4 → 6
4 → 7
4 → 8
4 → 9
4 → 10
4 → 11
4 → 12
Se ponen en una matriz.

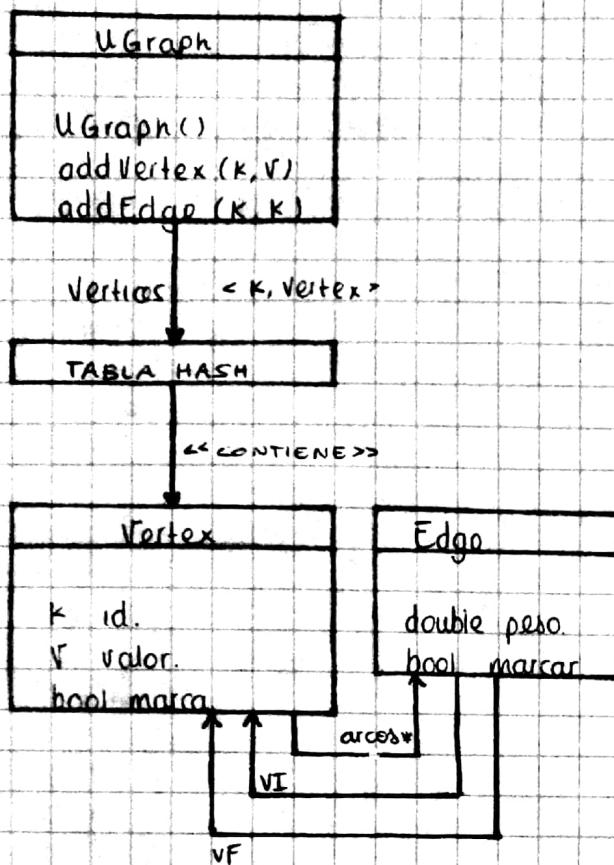
0									
1		■							
2			■						
3				■	■				
4				■	■				
5					■	■			

HAY QUE MARCAR POR DONDE PASO

11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

Adyacentes: Donde hay 1 en la tabla.

$$G = \langle V, E \rangle$$



Complejidad de grafo con 13 vértices más conectado.

$$O \left[13 + \frac{(V-1)(V)}{2} \right]$$

DFS

- To visit a vertex v:
 - Mark vertex v as visited
 - Recursively visit all unmarked vertices adjacent to v.

$$O(DFS(V_0)) = V_c + E_c$$

BFS

- Repeat until queue is empty
 Remove vertex v from queue
 Add queue all unmarked vertices adjacent to v and mark them.

$$O(BFS(V_0)) = V_c + E_c$$

Vértice inicial.

CONNECTED COMPONENTS

- Relación "está conectada" \equiv Relación de equivalencia.

- > Comienza por cualquiera y aplica DFS - marcándolo.
- > Cada que cambio de "grupo", contador Componentes++.

$$O(CC) = V_G + E_G$$

GRAFOS DIRIGIDOS

Digraph: Set of vertices connected pairwise by directed edges.

arcos \equiv # de vecinos

Dirigidos : Doble enlace. (Mentira)

Topological Sort.

- > No dependencia
- > No pueden haber rutas cílicas.

Vertice fuente: Que no depende.
 Que depende de muchos: Sumidero.
 ~ Nadie depende del el

MINIMUM SPANNING TREE : MST

Gráfico no dirigido con pesos positivos.

Output: A min weight spanning tree

DETERMINISMO Siempre misma solución, siempre mismo orden.

KRUSKAL'S ALGORITHM

- Consider edges in ascending order of weight
- Add next edge to tree Γ unless doing so would create a cycle

Pior caso: $O(2E \log_2 E)$

Mejor caso: $O(2(v-1) \log_2 E)$

PRIM'S ALGORITHM

- Start with vertex 0 and greedily grow tree Γ
- Add to Γ the min weight edge with exactly one endpoint in Γ .
- Repeat until $v-1$ edges

Pior caso: $O(E^2 \log_2 V)$ — Mejor

Prm deificado: Pior caso: $O(2E \log_2 E)$

Algoritmo dijkstra