

Notas de Asignatura

Estructuras de Datos y Algoritmos

Claudia Nallely Sánchez Gómez

Elaborado por:

Profesora: Claudia Nallely Sánchez Gómez

Universidad Panamericana campus Aguascalientes
Facultad de Ingeniería
Academia de Cómputo



Contenido

1. Introducción	6
1.1 Algoritmo	6
1.2 Estructura de Datos	6
1.3 Complejidad	6
2. Recursividad	11
3. Backtracking (Vuelta atrás)	14
Ejemplo 1: Problema de la mochila	14
Ejemplo 2: Problema de las ocho reinas	16
Ejemplo 3: Sudoku	17
3. Algoritmos de ordenamiento	18
4.1 Burbuja	19
4.2 Ordenamiento por Selección	19
4.3 Ordenamiento por Inserción	20
4.4 Ordenamiento por Montículos (Heapsort)	22
4.5 Ordenamiento Rápido (Quicksort)	25
4.6 Ordenamiento por Mezclas (MergeSort)	27
4.7 Ordenamiento por Conteo (Counting sort)	28
4.8 Ordenamiento Radix	29
4.9 Comparación de Algoritmos de Ordenamiento	30
5. Búsqueda en arreglos	31
5.1 Búsqueda secuencial	31
5.2 Búsqueda binaria	32
5.3 Búsqueda por transformación de claves (Hashing)	33
6. Algoritmos sobre cadenas de caracteres	35
6.1 Búsqueda secuencial	36
6.2 Búsqueda de cadenas: Knutt – Morris – Pratt	37
6.3 Búsqueda de cadenas: Boyer – Moore - Horspool	39
6.4 Comparación de cadenas: Distancia de Hamming	42
6.5 Comparación de cadenas: Distancia de Levenshtein	42
7. Dynamic Programming (Programación dinámica)	44
Example 1: Fibonacci	44
Example 2: Binomial coefficients	44
Ejemplo 3: Intercambio de monedas	45
Ejemplo 4: Problema de la mochila	47
8. Greedy Algorithms (algoritmos voraces)	49
Ejemplo 1: Intercambio de monedas	49

Ejemplo 2: Problema de la mochila	50
9. Lists	52
9.1 Singly Linked List.....	52
9.2 Doubly Linked List	52
9.3 Circularly Linked List	52
10. Stacks, Queues and Deques.....	53
10.1 Stacks	53
10.2 Queue	54
10.3 Deques.....	55
11. Trees.....	56
11.1 Representation	57
11.2 Traversals: DFS and BFS.....	59
11.3 Binary trees	61
11.3.1 Traversals: Inorder, preorder and postorder	61
11.3.2 Binary search tree	62
11.3.3 Árboles AVL (Adelson-Velsky-Landis).....	66
11.4 Set.....	68
11.5 Map	68
9.4 Árbol de segmentos.....	69
11. Compresión de datos	71
10.1 Código de Huffman.....	71
10.2 Codificación aritmética.....	73
12. Gráficas (Grafos).....	75
11.1 Representaciones de las gráficas	78
11.2 Búsqueda en anchura y en profundidad	80
11.3 Búsqueda Primero el Mejor (Best First Search)	84
11.4 Búsqueda A*	85
13. Algoritmos sobre gráficas.....	87
12.1 Algoritmo Unión – Buscar (Union Find).....	87
12.2 Árbol de mínima expansión	90
12.2.1 Kruskal.....	91
12.2.2 Prim	95
12.3 Distancias mínimas	99
12.3.1 Algoritmo de Dijkstra	99
12.3.2 Algoritmo de Floyd - Warshall	103
12.4 Orden topológico	105
Apéndice A. Processing	106
Interface gráfica.....	106

Funciones: setup() y draw()	107
Función: mouseClicked()	108
Función: keyPressed()	108
Apéndice B. Java	109
Graphics	109
ArrayList	110
Wrapper o envoltorio	110
Stack	111
Queue	111
PriorityQueue	112

1. Introducción

1.1 Algoritmo

Algoritmo: Conjunto finito y ordenado de pasos para resolver un determinado problema.

Existen una gran cantidad de algoritmos diseñados para resolver problemas comunes, por ejemplo:

- Calcular la ruta más corta de una ciudad a otra
- Determinar el número mínimo de monedas para cumplir con cierta cantidad.
- Ordenar un conjunto de nombres en orden alfabético
- Buscar un registro en una Base de Datos
- Etcétera.

Generalmente los algoritmos se expresan en pseudocódigo para que puedan entenderse sin importar el lenguaje de programación en el que van a implementarse.

1.2 Estructura de Datos

Estructura de Datos: Forma de almacenar información en una computadora de forma que sea eficiente su acceso.

1.3 Complejidad

Como programadores, debemos preocuparnos porque la resolución de cierto problema sea eficiente, esto es:

- El **tiempo de ejecución** debe ser el más corto posible, o al menos que se resuelva en un tiempo considerable.
- La **cantidad de memoria** debe ser la menor posible, o al menos no exceder la cantidad de memoria (bytes) destinados para resolver dicho problema.

En la mayoría de los casos, elegir el lenguaje de programación o los recursos de hardware no está en nuestras manos, por lo tanto, debemos preocuparnos porque el algoritmo sea eficiente.

La **complejidad algorítmica** representa la cantidad de recursos que requiere un algoritmo para resolver un problema, esto es, que tan eficiente es cierto algoritmo.

Tiempo de ejecución

Una medida muy útil es el tiempo de ejecución de un programa. Puede medirse físicamente, ejecutando el programa con cronometro en mano, o bien, calcularse sobre el código.

Cálculo del tiempo de ejecución en JAVA

```
long tiempo_inicio, tiempo_fin, tiempo_ejecucion;
tiempo_inicio = System.currentTimeMillis();
// Inicio programa
    for(int i=0; i<9999; i++)
        System.out.println("Dentro del ciclo");
// Fin programa
tiempo_fin = System.currentTimeMillis();
tiempo_ejecucion = tiempo_fin - tiempo_inicio;
System.out.println("Tiempo de ejecución "+ tiempo_ejecucion + " milisegundos");
```

Una estimación del número de operaciones por segundo en una máquina es:

C/C++	Java o C#	Python
1×10^8	$1 \times 10^8 / 3$	$1 \times 10^8 / 6$

Por ejemplo, suponga que un programador codifica dos algoritmos que procesan ciertos registros (los dos algoritmos procesan los registros bien) y prueba cada uno de ellos con un conjunto distinto de registros para medir su eficiencia. El resultado del tiempo de ejecución de las pruebas es:

# registros	10	20	50	100	1000	5000
Algoritmo 1	0.00s	0.01s	0.05s	0.47s	23.92s	47min
Algoritmo 2	0.05s	0.05s	0.06s	0.11s	0.78s	14.22s

¿Qué algoritmo elegirías? ¿Por qué crees que aunque los algoritmos hagan lo mismo sus tiempos son distintos?

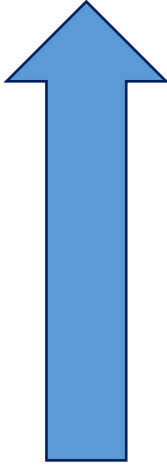
Ejemplo tomado de: <https://www.topcoder.com/community/data-science/data-science-tutorials/computational-complexity-section-1/#SECTION00080000000000000000>

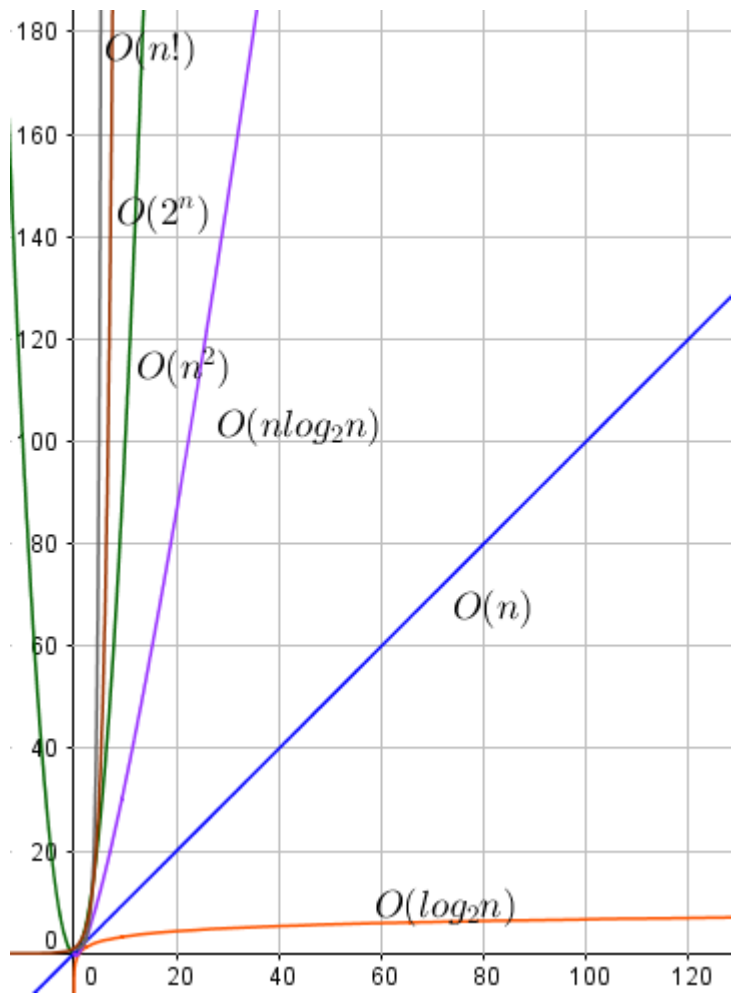
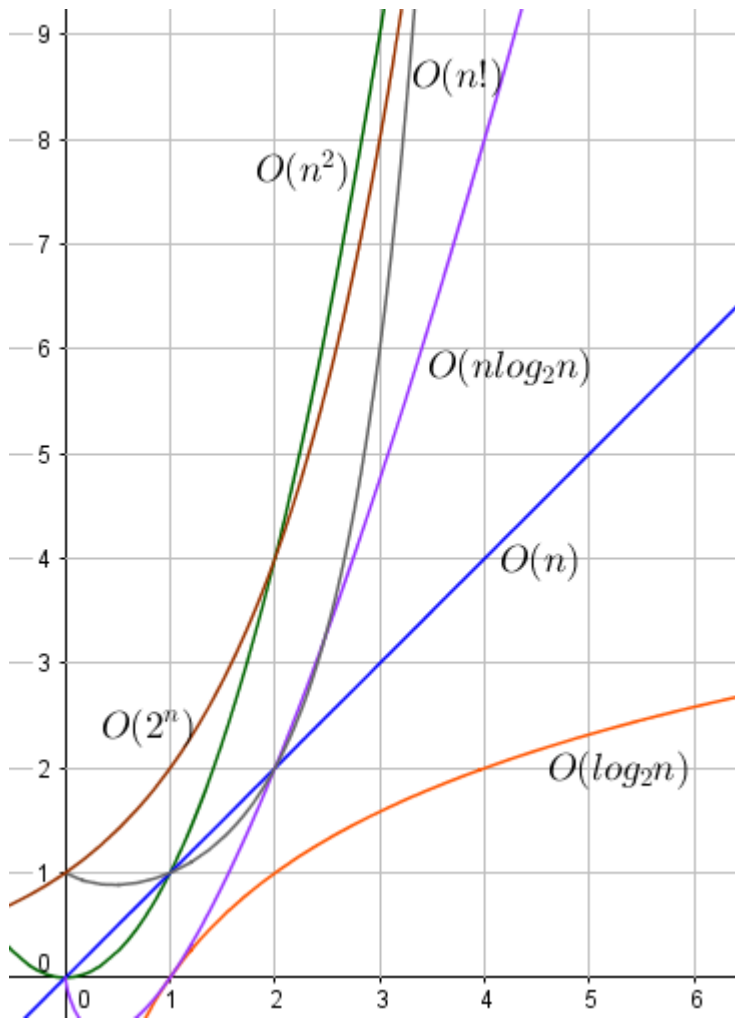
Órdenes de complejidad

El orden de complejidad, denotado como $O(\dots)$, es una medida relativa de la complejidad de dicho algoritmo. Es relativa, porque un algoritmo puede cambiar su eficiencia debido a causas externas a él, por ejemplo:

- La máquina sobre la cual se está ejecutando
- Lenguaje de programación
- Cualquier otro elemento de software o hardware que interfiera en la ejecución

Los órdenes utilizados en el análisis de algoritmos, desde el más óptimo, hasta el menos óptimo son:

Notación	Orden	<div>Mejor</div>  <div>Peor</div>
$O(1)$	Constante	
$O(\log \log n)$	Sublogarítmico	
$O(\log n)$	Logarítmico	
$O(n)$	Lineal	
$O(n \log n)$	Lineal algorítmico	
$O(n^c)$	Potencial	
$O(c^n)$	Exponencial	
$O(n!)$	Factorial	
$O(n^n)$	Potencial exponencial	



Orden constante $O(1)$

Cuando las instrucciones se ejecutan una sola vez, por ejemplo:

Insertar un registro en una Base de Datos, Complejidad $O(2)$

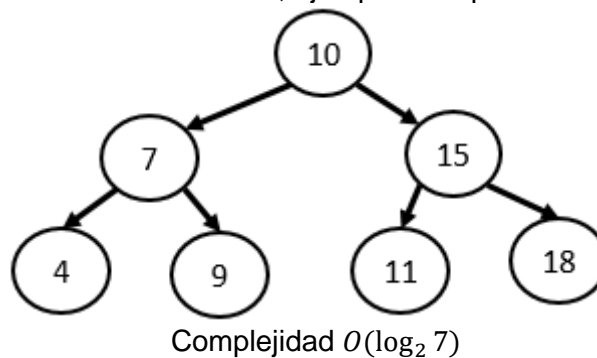
Parámetro: Registro a insertar

Agregar el registro en la última posición

Incrementar en uno el número de registros

Orden logarítmico $O(\log n)$

Cuando existe una iteración o recursión no estructural, ejemplo: búsqueda binaria:

**Orden lineal $O(n)$**

Cuando existen instrucciones que se ejecutan un cierto número de veces. Generalmente ocurren cuando hay ciclos, por ejemplo:

Promedio de N números, $O(\text{Cantidad de números})$

Parámetro: Arreglo de números

Definir e inicializar suma en cero

Desde 1 hasta cantidad de números

Leer el número

$\text{suma} = \text{suma} + \text{numero}$

$\text{promedio} = \text{suma} / \text{cantidad de números}$

Imprimir el promedio

Orden potencial $O(n^c)$

Cuando existen ciclos anidados. Por ejemplo: Suma de matrices cuadradas:

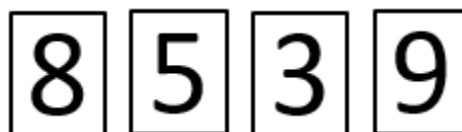
$$\begin{bmatrix} 5 & 3 & 0 \\ -1 & 1 & 4 \\ 6 & 3 & -2 \end{bmatrix} + \begin{bmatrix} 7 & 3 & -8 \\ 0 & 4 & -3 \\ 6 & 0 & 1 \end{bmatrix}$$

Complejidad $O(\text{Tamaño de la matriz}^2)$

Orden factorial $O(n!)$

Cuando existen operaciones que requieren un número factorial de operaciones. Por ejemplo:

Calcular la cantidad de números que se pueden formar con cuatro dígitos distintos:



Complejidad $O(4!)$

Para tratar de medir el impacto de la complejidad de un algoritmo en tiempo, vamos a suponer que una máquina en un lenguaje determinado hace 1×10^8 operaciones por segundo y vamos a tratar de estimar el tiempo que se tardarán diferentes algoritmos con órdenes distintos de complejidad y diferente número de datos:

Algoritmo \ N	2	10	100	1000	1×10^6	1×10^9
$O(\log_2 N)$	0.01us	0.03us	0.06us	0.09us	0.2us	0.3us
$O(N)$	0.02us	0.1us	1.0us	10us	0.01s	1s
$O(N \log_2 N)$	0.02us	0.3us	6.6us	99us	0.20s	4.98min
$O(N^2)$	0.04us	1.0us	0.1ms	0.01s	2.7hrs	317años
$O(2^N)$	0.04us	10us	4×10^{14} años	-	-	-
$O(N!)$	0.02ms	0.03s	-	-	-	-

ACTIVIDAD:

1. Suponga que $n = 1,000,000$. ¿Aproximadamente cuánto es más rápido un algoritmo con complejidad $O(n \log_2 n)$ contra otro $O(n^2)$? Utilizando logaritmo base 2.

- a) 20x
- b) 1,000x
- c) 50,000x
- d) 1,000,000x

Generalmente para el análisis de un algoritmo se deben tener en cuenta las siguientes cosas:

- Complejidad en promedio
- Complejidad en el mejor caso
- Complejidad en el peor caso
- Cantidad de memoria a utilizar

Ejemplo: Análisis de la complejidad de una función

```
int factorial( int n){
    int f = 1;           // 1 operación
    for(int i=0;i<n;i++){ // 1 inicialización, n comparaciones, n incrementos
        f = f*i;          // 1 operación que se realiza n veces
    }
    return f;            // 1 operación
}
```

$$O() = 1 + 1 + n + n + n + 1$$

$$O() = 3 + 3n$$

Se deja el término con mayor grado eliminando el coeficiente

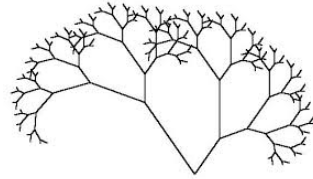
Por lo tanto, la función factorial es de orden $O(n)$

2. Recursividad

Es una técnica ampliamente utilizada en programación para resolver problemas a partir de la solución del mismo problema pero más pequeño.

Por ejemplo, dibujar un árbol:

Se generan dos ramas que
 tienen otras dos ramas más pequeñas que
 tienen otras dos ramas más pequeñas que
 tienen otras dos ramas más pequeñas que
 ...



Para implementar esta técnica en programación se utilizan funciones que se mandan llamar a ellas mismas.

Una función recursiva, es aquella que se manda llamar a sí misma.

Las claves para construir una función recursiva es la siguiente:

- Debe existir al menos un **caso base**, el cual pueda resolverse sin necesidad de llamar nuevamente a la función. Esto con el objetivo de evitar que la función se mande llamar de forma infinita.
- Cada **llamada recurrente** debe ser con el objetivo de resolver un problema de menor complejidad.

Ejemplo: Imprimir los números en orden descendente

Por ejemplo: 6 5 4 3 2 1 0, 3 2 1 0

Caso base: Si el número es 0, ya no se imprimen más números

Proceso recursivo: Imprimir el número y mandar llamar la función con un número más pequeño

Función: imprimir

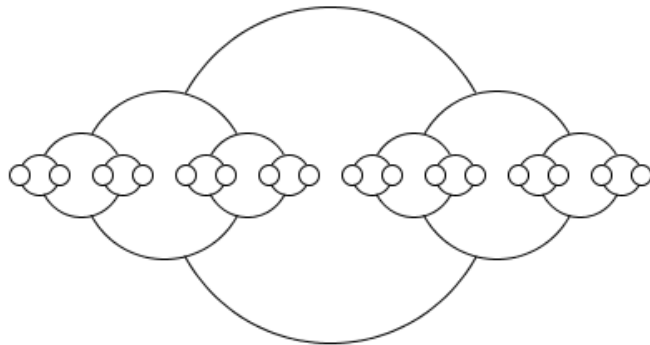
```
void imprimirOrdenDescendente(int n){
    if( n<= 0 ){
        print(" " + 0);
    }else{
        print(" " + n);
        imprimirOrdenDescendente(n-1);
    }
}
```

Ejemplo: Pintar un fractal de círculos

```
void setup(){
    size(500,500);
    background(255);
    fill(255); stroke(0);
    circulo(250,300,100);
}

void circulo(int x,int y, int radio){
    ellipse(x,y,radio*2,radio*2);

    if( radio>=10 ){
        circulo(x+radio,y,radio/2);
        circulo(x-radio,y,radio/2);
    }
}
```



ACTIVIDAD: Realice los siguientes programas utilizando recursividad:

1. Imprimir los números naturales en orden ascendente a partir de un número, por ejemplo:
5 -> 1 2 3 4 5
3 -> 1 2 3
2. Imprimir los números naturales en orden descendente a partir de un número, por ejemplo:
5 -> 5 4 3 2 1
3 -> 3 2 1
3. Imprimir los números naturales pares en orden descendente a partir de un número, por ejemplo:
10 -> 10 8 6 4 2
4. Recorrer e imprimir los valores de un array de forma recursiva en orden ascendente.
Ejemplo de array: [5,2,6,8,3,2]
5. Dibujar un fractal

Ejemplo: Factorial de un número

Ejemplos de cálculo de factorial: $6! = 6*5*4*3*2*1$ $4! = 4*3*2*1$ $2! = 2*1$ $1! = 1$

Caso base: El factorial de 1 es 1

Proceso recursivo: $\text{Factorial}(n) = n * \text{Factorial}(n-1)$

Función: factorial

```
int factorial(int n){
    if( n<=1 )           // Caso base
        return 1;
    else                 // Proceso recursivo
        return n * factorial(n-1);
}
```

¿Qué pasa con la memoria utilizando recursividad?

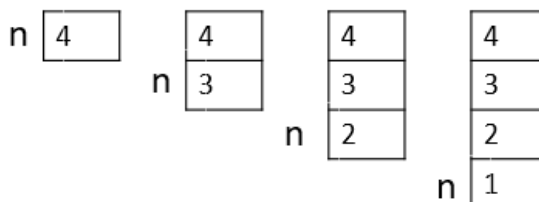
Lo que pasa es que se almacena en la pila de la memoria la información del número de llamada y el valor de la variable, por ejemplo:

Al utilizar la función recursiva para el cálculo del factorial:

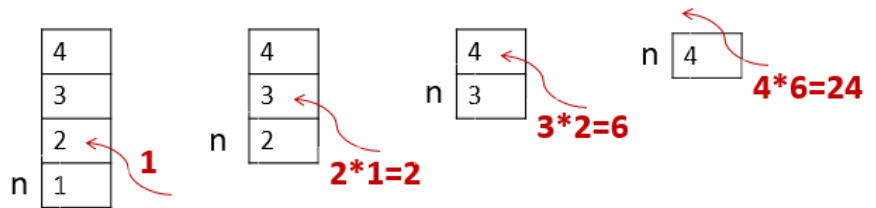
resultado = factorial(4);

Primero se llena la pila con los parámetros

```
int factorial(int n){
    if( n<=1 )           // Caso base
        return 1;
    else                 // Proceso recursivo
        return n * factorial(n-1);
}
```



Luego se regresan los resultados



ACTIVIDAD: Realice los siguientes programas utilizando recursividad:

1. Cálculo del factorial de un número, por ejemplo:

5 -> $5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$

3 -> $3 \cdot 2 \cdot 1 = 6$

2. Calcular el número Fibonacci de un número.

Recordar que $F(n) = F(n-1) + F(n-2)$, y $F(0) = 0$, $F(1) = 1$

3. Sumar los números naturales antecesores de un número, por ejemplo:

4 -> $1 + 2 + 3 + 4 = 10$

4. Sumar los números naturales antecesores y pares de un número, por ejemplo:

10 -> $2 + 4 + 6 + 8 + 10 = 30$

4 -> $2 + 4 = 6$

5. Calcule una división entera utilizando restas sucesivas. Por ejemplo:

7/3 -> $7 - 3 = 4$ $4 - 3 = 1$ $1 - 3 = \text{ERROR}$ Por lo tanto $7/3 = 2$

15/2 -> $15 - 2 = 13$ $13 - 2 = 11$ $11 - 2 = 9$ $9 - 2 = 7$ $7 - 2 = 5$ $5 - 2 = 3$ $3 - 2 = 1$ $1 - 2 = \text{ERROR}$ Por lo tanto $15/2 = 7$

6. Conversión de número decimal a binario

8 -> 1000

6 -> 110

7. Conversión de números binarios a decimal

1000 -> 8

110 -> 6

8. Invertir un número, por ejemplo:

123 -> 321

95827 -> 72859

9. Sumar los dígitos de un número, por ejemplo:

123 -> $1 + 2 + 3 = 6$

91 -> $9 + 1 = 10$

10. Imprimir todas las combinaciones posibles de varios caracteres diferentes, por ejemplo:

abc: abc acb bca bac cab cba

3. Backtracking (Vuelta atrás)

Es una técnica de solución de problemas **discretos** basado en **recursividad**, por ejemplo:

- Calcular la ruta más corta para llegar de un punto a otro
- Problema de la mochila
- Las ocho reinas

Algoritmo Backtracking

Funcion: Backtracking(parámetro: solución)

Si la solución es válida

 Si la solución actual es mejor a la mejor solución encontrada hasta el momento

 Solución mejor es igual a la solución actual

 Fin si

Sino

 Revisar las soluciones derivadas de forma recursiva

Fin si

En general, lo que se hace es construir un árbol de soluciones, donde las modificaciones de las soluciones generan nuevas soluciones hijas. Se parece a una Búsqueda en Profundidad en un árbol.

Ejemplo 1: Problema de la mochila

Se tiene una mochila con una capacidad limitada en peso y el objetivo es guardar objetos en la mochila de tal forma que se maximice la suma de los precios de los objetos y no se exceda la capacidad en peso de la mochila. Para esto se tienen n objetos con peso y precio determinado.

Parámetros:

- Capacidad de la mochila (en peso)
- Peso, precio y cantidad de objetos de tipo 1
- Peso, precio y cantidad de objetos de tipo 2
- ...

Retorno: - Tipo y cantidad de objetos a guardar en la mochila

Cada solución se puede representar con un vector del tamaño de los objetos disponibles que contiene valores booleanos indicando si se tomó o no cada objeto:

$$\text{Solución} = \{O_1, O_2, O_3, \dots, O_n\} \quad \text{donde } O_i = [0,1]$$

Ejemplo:

Capacidad de la mochila 3kg

Objetos:

- 1 objetos de 1 kg con un valor de \$5
- 2 objetos de 1 kg con un valor de \$30
- 1 objetos de 2 kg con un valor de \$20

Paso 1

0kg \$0
{x,x,x,x}

{Objeto 1kg \$5, Objeto 1kg \$30, Objeto 1kg \$30, Objeto 2kg \$20}

x-> No se ha decidido 0-> no se coloca el objeto 1-> si se coloca el objeto

{x,x,x,x} No es una solución válida aún porque no se ha decidido nada

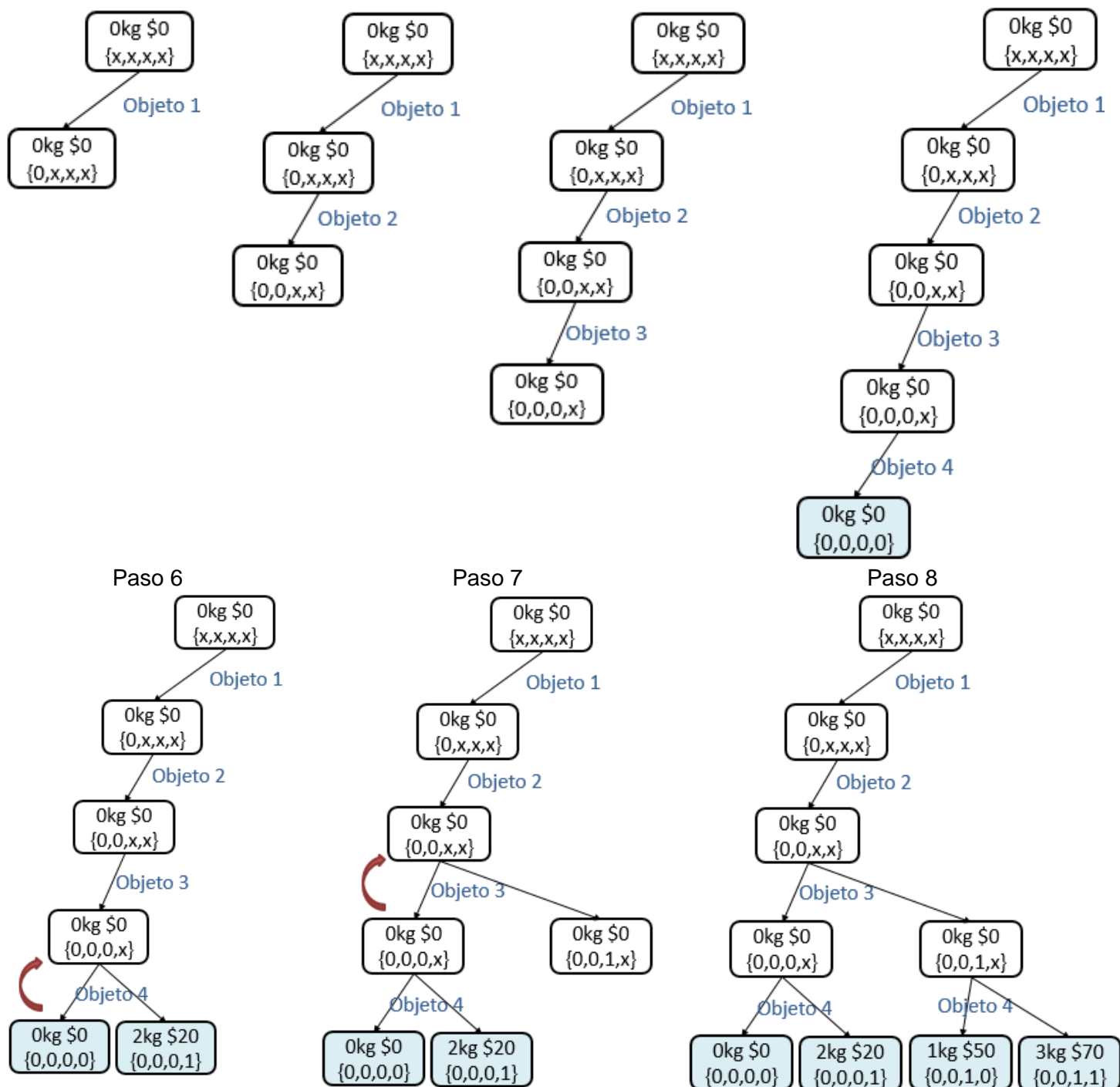
La mochila pesa 0 kg y tiene una utilidad de \$0

Paso 2

Paso 3

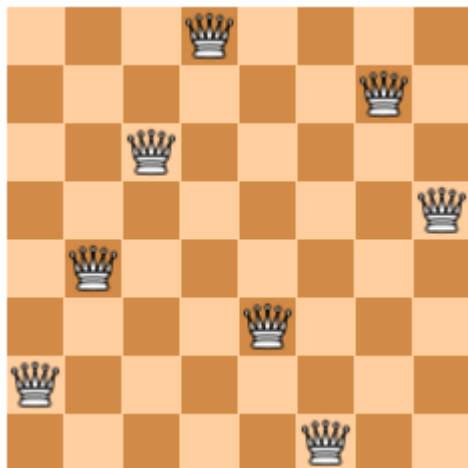
Paso 4

Paso 5



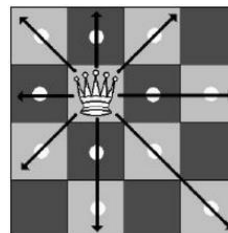
Ejemplo 2: Problema de las ocho reinas

Este problema consiste en colocar ocho reinas en un tablero de ajedrez de 8x8 de tal forma que no se maten entre ellas.



wikipedia.org

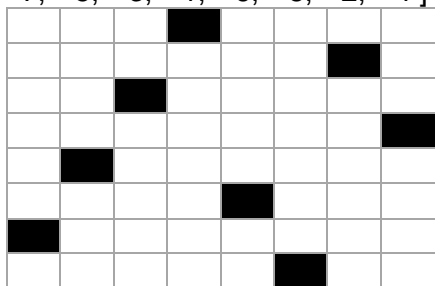
Esto considerando que una reina puede matar a cualquier ficha que este colocada en la misma fila, columna o en diagonal.



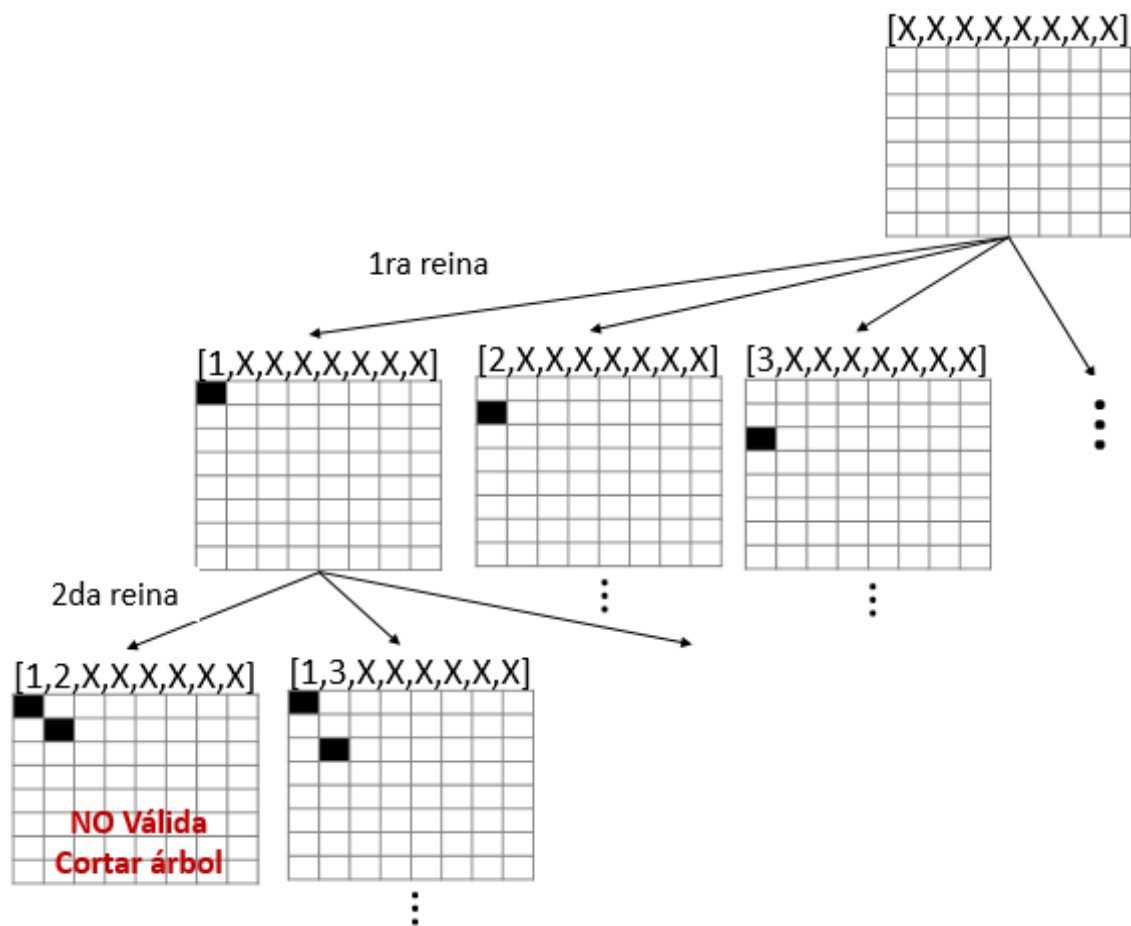
wikipedia.org

Cada reina debe ponerse en una columna distinta, porque si no se matarían entre sí, por lo tanto cada solución se puede representar con un arreglo de tamaño ocho. La posición del arreglo representa la columna y el número en el arreglo representa la fila. Por ejemplo:

Solución = [7, 5, 3, 1, 6, 8, 2, 4]



Ejemplo:



Ejemplo 3: Sudoku

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

3. Algoritmos de ordenamiento

Un algoritmo de ordenamiento está diseñado para cambiar la ubicación de los elementos de una lista de tal manera que queden ordenados de acuerdo a algún criterio como: numérico, alfabético, etc.

Si tenemos el siguiente vector:

$$v = [4, 9, 3, 6, 1]$$

después de ordenarlo numéricamente quedaría de la siguiente forma:

$$v = [1, 3, 4, 6, 9]$$

El ordenar los elementos de una lista sirve para:

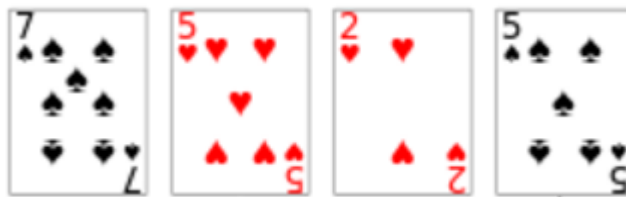
- Optimizar la búsqueda de registros en una base de datos
- Generar reportes con la información ordenada
- Etcétera

Estabilidad

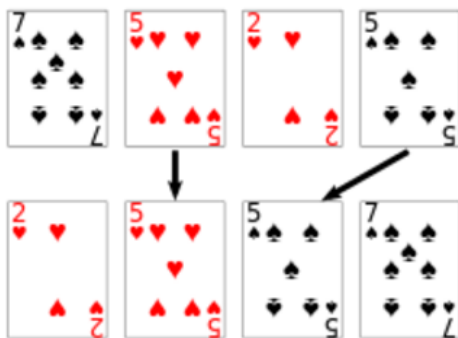
Los algoritmos de ordenamiento estables mantienen el orden relativo de elementos con valores iguales.

Ejemplo: Algoritmo de ordenamiento estable y no estable

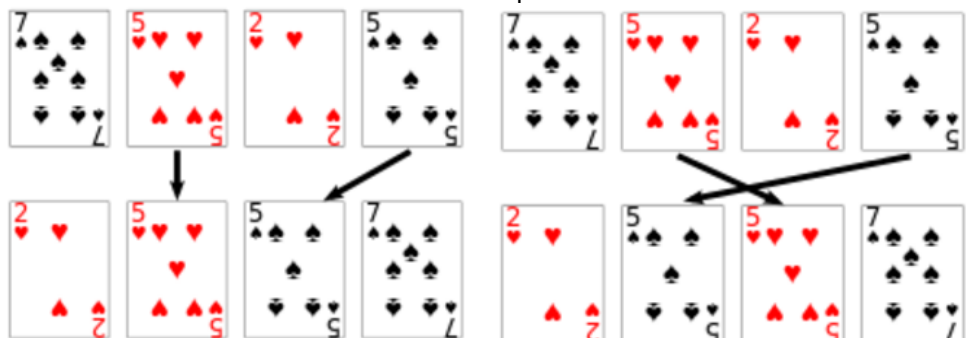
Problema de Ordenamiento



Algoritmo de Ordenamiento Estable
1 solución



Algoritmo de Ordenamiento NO Estable
2 soluciones posibles



4.1 Burbuja

Es un algoritmo sencillo de ordenación, su complejidad es $O(n^2)$ y la memoria que utiliza es $M(n)$. Se llama burbuja porque lo que hace es mover los números grandes hacia los últimos índices.

Ejemplo: Ordenar el arreglo [4, 9, 3, 6, 1] con burbuja

Recorre todas las casillas i desde 1 hasta $n-1$ comparando i con $i+1$, si la casilla i tiene un valor mayor: se intercambian

[4, 9, 3, 6, 1]

[4, 9, 3, 6, 1] → [4, 3, 9, 6, 1]

[4, 3, 9, 6, 1] → [4, 3, 6, 9, 1]

[4, 3, 6, 9, 1] → [4, 3, 6, 1, 9]

Recorre todas las casillas i desde 1 hasta $n-2$, comparando i con $i+1$, si la casilla i tiene un valor mayor: se intercambian

[4, 3, 6, 1, 9] → [3, 4, 6, 1, 9]

[3, 4, 6, 1, 9]

[3, 4, 6, 1, 9] → [3, 4, 1, 6, 9]

Recorre todas las casillas i desde 1 hasta $n-3$, comparando i con $i+1$, si la casilla i tiene un valor mayor: se intercambian

[3, 4, 1, 6, 9]

[3, 4, 1, 6, 9] → [3, 1, 4, 6, 9]

Recorre todas las casillas i desde 1 hasta $n-4$, comparando i con $i+1$, si la casilla i tiene un valor mayor: se intercambian

[3, 1, 4, 6, 9] → [1, 3, 4, 6, 9]

4.2 Ordenamiento por Selección

Es un algoritmo sencillo de ordenación, su complejidad es $O(n^2)$ y la memoria que utiliza es $M(n)$. En forma general, lo que hace es lo siguiente:

- Busca el 1er elemento con menor valor y lo intercambia con el 1er elemento
- Busca el 2do elemento con menor valor y lo intercambia por el 2do elemento
- Busca el 3er elemento con menor valor y lo intercambia por el 3er elemento
- ...

Ejemplo: Ordenar el arreglo [4, 9, 3, 6, 1]

[4, 9, 3, 6, 1] → El menor es 1 → [1, 9, 3, 6, 4]

[1, 9, 3, 6, 4] → El menor es 3 → [1, 3, 9, 6, 4]

[1, 3, 9, 6, 4] → El menor es 4 → [1, 3, 4, 6, 9]

[1, 3, 4, 6, 9] → El menor es 6 → *No se realiza cambio*

4.3 Ordenamiento por Inserción

Es un algoritmo de ordenación basado en el proceso que utiliza un ser humano para ordenar, su complejidad es $O(n^2)$ y la memoria que utiliza es $M(n)$. En forma general, lo que hace es tomar uno a uno los datos e irlos ordenando conforme los va tomando, como cuando tomamos un manojo de cartas y las vamos ordenando una a una.

Ejemplo: Ordenar el arreglo [4, 9, 3, 6, 1]

Ordenados → [x, x, x, x, x] *No ordenados* → [4, 9, 3, 6, 1]

Guardar → 4 *Ordenados* → [4, x, x, x, x] *No ordenados* → [x, 9, 3, 6, 1]

Guardar → 9 *Ordenados* → [4, x, x, x, x] *No ordenados* → [x, x, 3, 6, 1]

9 > 4, *Ordenados* → [4, 9, x, x, x]

Guardar → 3 *Ordenados* → [4, 9, x, x, x] *No ordenados* → [x, x, x, 6, 1]

3 < 9, se recorre el 9 [4, x, 9, x, x]

3 < 4, se recorre el 4 [x, 4, 9, x, x]

1er posición *Ordenados* → [3, 4, 9, x, x]

Guardar → 6 *Ordenados* → [3, 4, 9, x, x] *No ordenados* → [x, x, x, x, 1]

6 < 9, se recorre el 9 [3, 4, x, 9, x]

6 > 4, *Ordenados* → [3, 4, 6, 9, x]

Guardar → 1 *Ordenados* → [3, 4, 6, 9, x] *No ordenados* → [x, x, x, x, 1]

1 < 9, se recorre el 9 [3, 4, 6, x, 9]

1 < 6, se recorre el 6 [3, 4, x, 6, 9]

1 < 4, se recorre el 4 [3, x, 4, 6, 9]

1 < 3, se recorre el 3 [x, 3, 4, 6, 9]

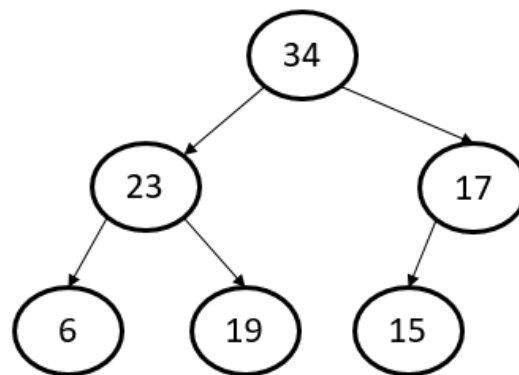
1er posición *Ordenados* → [1, 3, 4, 6, 9]

4.4 Ordenamiento por Montículos (Heapsort)

MONTÍCULO (HEAP)

Es una estructura de datos que guarda la información en forma de árbol de tal forma que:

- Un nodo sólo puede tener dos nodos hijos, esto es, es un árbol binario
- El valor de un elemento “padre” siempre es mayor que los valores de sus “hijos”
- Sólo se puede acceder al elemento raíz
- Está equilibrado, es decir, el número de niveles en las ramas izquierda y derecha sólo difieren a lo más en un nivel.



El montículo puede ser representado fácilmente en un arreglo, por ejemplo:

Índices:	1	2	3	4	5	6	7	...			
	34	23	17	6	19	15					

Índices de los hijos:

Índice del padre:

$$\begin{aligned}\text{Índice hijo 1} &= 2 * (\text{índice nodo padre}) \\ \text{Índice hijo 2} &= 2 * (\text{índice nodo padre}) + 1\end{aligned}$$

$$\text{Índice del padre} = \frac{\text{Índice del nodo hijo}}{2}$$

Las operaciones básicas que se pueden hacer sobre un montículo son:

- Inserción
- Eliminación

Inserción: Se debe agregar el elemento en la última posición. Una vez agregado se debe verificar recursivamente que el valor del elemento sea menor que el valor del elemento padre, en caso contrario se deben intercambiar las posiciones.

Ejemplo: inserción

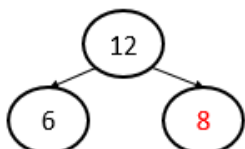
Insertar 6



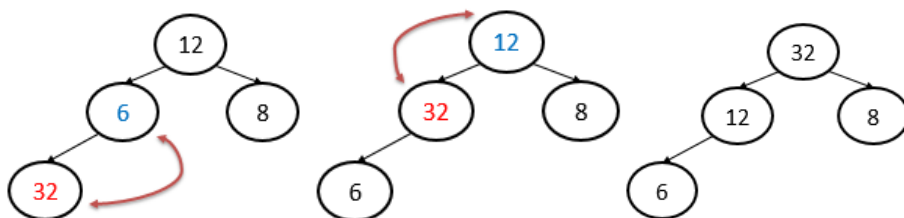
Insertar 12



Insertar 8



Insertar 32



0	1	2	3	4	5	6	7	8
		6						

0	1	2	3	4	5	6	7	8
	6	12						

0	1	2	3	4	5	6	7	8
12	6							

0	1	2	3	4	5	6	7	8
	12	6	8					

0	1	2	3	4	5	6	7	8
	12	6	8	32				

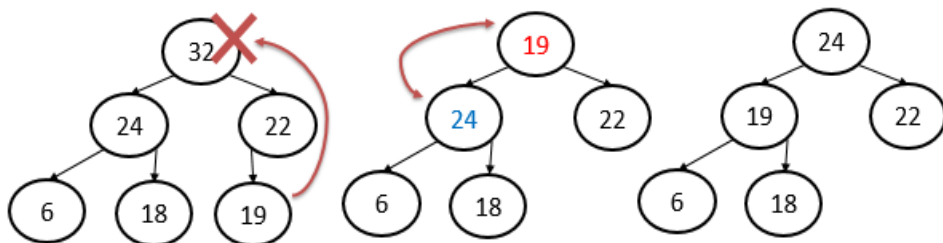
0	1	2	3	4	5	6	7	8
	12	32	8	6				

0	1	2	3	4	5	6	7	8
32	12	8	6					

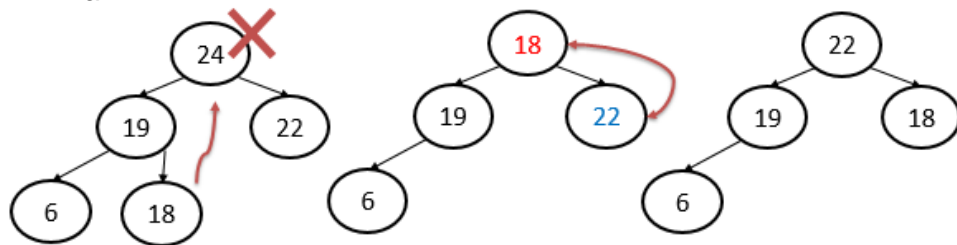
Eliminación: Sólo se puede eliminar el elemento raíz. Para cubrir la posición vacía lo que se hace es colocar el último elemento en la posición raíz y después intercambiar recursivamente posición padre - hijo en caso de que algún nodo hijo tenga un valor más grande que el nodo padre. En caso de que los dos hijos tengan un valor más grande que el padre se debe intercambiar la posición por el hijo con mayor valor.

Ejemplo: eliminación

Eliminar



Eliminar



0	1	2	3	4	5	6
	32	24	22	6	18	19

0	1	2	3	4	5	6
	19	24	22	6	18	

0	1	2	3	4	5	6
	24	19	22	6	18	

0	1	2	3	4	5	6
	24	19	22	6	18	

0	1	2	3	4	5	6
	18	19	22	6		

0	1	2	3	4	5	6
	22	19	18	6		

Ordenamiento con montículos

Es un algoritmo de ordenamiento de complejidad $O(n \log n)$ y utiliza la estructura de datos: montículo.

En general, heapsort consiste en:

1. Insertar todos los elementos en un montículo
2. Eliminar uno a uno los elementos, el orden de la eliminación será del mayor al menor.

ACTIVIDAD:

1. Realice un programa que genere al azar 1000 números y después los ordene utilizando: Heapsort

4.5 Ordenamiento Rápido (Quicksort)

Es un algoritmo creado por C. A. R. Hoare basado en la técnica divide y vencerás. Su complejidad es $O(n \log n)$. El algoritmo Quicksort consiste en:

- Tomar un elemento como pivote, nosotros vamos a tomar el primer elemento
- Se colocan los elementos menores al valor del pivote a la izquierda y los elementos mayores al valor del pivote a la derecha
- Se ordenan recursivamente las dos sublistas (la que queda a la izquierda del pivote y la que queda a la derecha del pivote)
- Se unen la lista ordenada de la izquierda + el pivote + la lista ordenada de la derecha

Ejemplo (idea general):

Se toma el primer elemento de la lista como elemento pivote

[4,9,2,3,6,1,7,0]

Al lado izquierdo se colocan los elementos menores al pivote y al derecho los mayores

[2,3,1,0,4,9,6,7]

Se ordenan las dos sublistas, la del lado izquierdo al pivote, y del lado derecho del pivote

[0,1,2,3] = Quicksort([2,3,1,0]);

[6,7,9] = Quicksort([9,6,7]);

Se unen las dos sublistas ordenadas y el pivote

[0,1,2,3,4,6,7,9]

¿Cómo colocar los elementos pequeños a la izquierda y los grandes a la derecha?

Se utilizan tres índices: i (índice izquierdo), d (índice derecho), p (índice pivote).

- Se inicializan los índices:
 - p en la primer posición, i en la segunda posición y d en la última posición
- Repetir mientras que $i < d$
 - i avanza a la derecha mientras que $i < d$ y $\text{arreglo}[i] \leq \text{arreglo}[p]$
 - d avanza a la izquierda mientras que $i < d$ y $\text{arreglo}[d] > \text{arreglo}[p]$
 - Si $i < d$, entonces se intercambian los valores en las posiciones i,d
- Finalmente se coloca al elemento pivote al centro
 - Si $\text{arreglo}[i] \leq \text{arreglo}[p]$ se intercambian los valores en las posiciones i,p, y el pivote se mueve a la posición i
 - Si $\text{arreglo}[i] > \text{arreglo}[p]$, se intercambian los valores en las posiciones i-1,p y el pivote se mueve a la posición i-1

Ejemplo:

Inicializar los índices

4 9 2 3 6 1 7 0
p i d

Mover los índices i, d

Mover i 4 9 2 3 6 1 7 0
(no se pudo) p i d
Mover d 4 9 2 3 6 1 7 0
(no se pudo) p i d
Intercambio 4 0 2 3 6 1 7 9
(i,d) p i d
Mover i 4 0 2 3 6 1 7 9
(3 posiciones) p i d
Mover d 4 0 2 3 6 1 7 9
(2 posiciones) p i d
Intercambio 4 0 2 3 1 6 7 9
(i,d) p i d
Mover i 4 0 2 3 1 6 7 9
(1 posición) p i,d

Acomodar el pivote

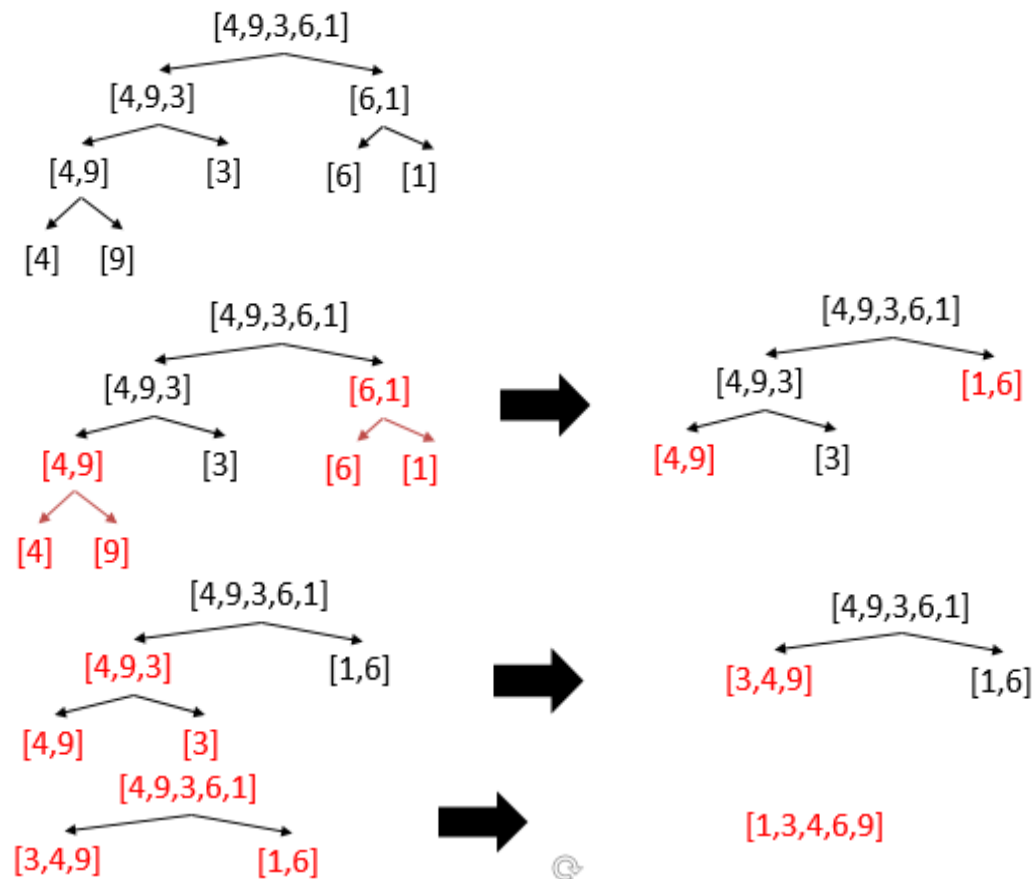
1 0 2 3 4 6 7 9
p i,d

4.6 Ordenamiento por Mezclas (MergeSort)

Fue desarrollado en 1945 por John Von Neumann y también está basado en la técnica divide y vencerás. Consiste en lo siguiente:

- Divide una lista en dos, para ordenarlas de forma recursiva
- Ordena la lista en base a las dos sub listas ordenadas

Ejemplo:



¿Cómo ordenar una lista en base a dos sub listas ordenadas?

- Inicializar las posiciones en las dos sub-listas en el 1er elemento
- Mientras no se hayan recorrido todos los elementos de las dos sub-listas:
 Guardar en la lista ordenada el elemento con menor valor
 Aumentar en uno la posición de la lista de donde se obtuvo el valor

Ejemplo: Mezclar las sub listas [3, 4, 9] y [1, 6]

```

[3, 4, 9] [1, 6] → [ 1 ]
[3, 4, 9] [1, 6] → [ 1, 3 ]
[3, 4, 9] [1, 6] → [ 1, 3, 4 ]
[3, 4, 9] [1, 6] → [ 1, 3, 4, 6 ]
[3, 4, 9] [1, 6] → [ 1, 3, 4, 6, 9 ]
  
```

Nota: Si la lista a ordenar es de tamaño uno o cero, YA ESTA ORDENADA. No es necesaria la recursividad.

4.7 Ordenamiento por Conteo (Counting sort)

Es un algoritmo simple de ordenamiento que se basa en el conteo del número de elementos de cada clase para luego ordenarlos. Sólo se puede utilizar para ordenar elementos contables.

El algoritmo consiste en:

- Pre – calcular el número de valores diferentes
- Crear un vector de conteo para llevar la cuenta de las apariciones de cada valor
- Recorrer la lista y contar los elementos
- Recorrer el valor el vector de conteo para generar la lista ordenada

Ejemplo: Lista a ordenar = {3, 5, 9, 1, 8, 7, 2, 9, 4, 3}

Valores diferentes o clases

1, 2, 3, 4, 5, 6, 7, 8, 9

Vector de conteo

1	2	3	4	5	6	7	8	9
0	0	0	0	0	0	0	0	0

Recorrer la lista y contar los elementos

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
1	0	1	0	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
1	1	1	0	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
0	0	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
1	1	1	0	1	0	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
1	0	1	0	1	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
1	1	1	1	1	1	0	1	1	2	2	2	2	2	2	2	2	2	2	2	2

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
1	0	1	0	1	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1

{	3	,	5	,	9	,	1	,	8	,	7	,	2	,	9	,	4	,	3	}
1	2	3	4	5	6	7	8	9												
1	1	2	1	1	0	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2

Recorrer el vector de conteo para obtener la lista ordenada

1	2	3	4	5	6	7	8	9
1	1	2	1	1	0	1	1	2

1, 2, 3, 3, 4, 5, 7, 8, 9, 9

4.8 Ordenamiento Radix

Es un algoritmo de ordenamiento que ordena datos por partes, es decir, divide cada elemento en otros elementos más pequeños, por ejemplo: los números los divide en dígitos, palabras en letras, etc.

Por ejemplo, si ordena:

- Números enteros: primero ordena por unidades, luego por decenas, luego por centenas, ...
- Fechas: primero ordena por días, luego por meses, y finalmente por años.

Algoritmo:

- Dividir los datos a ordenar en elementos simples
- Ordenar del elemento menos significativo al elemento menos significativo
 - Crear un queue por cada valor que pueda tomar el elemento a ordenar
 - Agregar al queue correspondiente cada dato
 - Obtener nuevamente la lista sacando los datos de las queue en orden

Ejemplo: Ordenar { 345, 872, 123, 654, 298, 008, 734 }

Lista = { 345, 872, 123, 654, 298, 008, 734, 126 }

Se agregan los datos en la queue correspondiente a la **unidad**:

0	1	2	3	4	5	6	7	8	9
		872	123	654, 734	345	126		298, 008	

Se obtienen los datos de las queue en orden

Lista = { 872, 123, 654, 734, 345, 126, 298, 008 }

Se agregan los datos en la queue correspondiente a la **decena**:

0	1	2	3	4	5	6	7	8	9
008		123, 126	734	345	654		872		298

Se obtienen los datos de las queue en orden

Lista = { 008, 123, 126, 734, 345, 654, 872, 298 }

Se agregan los datos en la queue correspondiente a la **centena**:

0	1	2	3	4	5	6	7	8	9
008	123, 126	298	345			654	734	842	

Se obtienen los datos de las queue en orden

Lista = { 008, 123, 126, 298, 345, 654, 734, 872 }

4.9 Comparación de Algoritmos de Ordenamiento

	Complejidad promedio	Comparación	Técnica	Recursivo	Estable
Bubble Sort	$O(n^2)$	Si		No	Si
Selection Sort	$O(n^2)$	Si		No	No
Insertion Sort	$O(n^2)$	Si		No	Si
Heap Sort	$O(n \log n)$	Si	Heap (montículo)	No	No
Quick Sort	$O(n \log n)$	Si	Divide y vencerás	Si	No
Merge Sort	$O(n \log n)$	Si	Divide y vencerás	Si	Si
Radix Sort	$O(wn)$	No	Queues (colas)	No	Si
Counting Sort	$O(n)$	No	Conteo	No	No

Complejidad	Promedio	Mejor caso	Peor caso	Memoria adicional
Bubble Sort	$O(n^2)$	—	—	
Selection Sort	$O(n^2)$	—	—	
Insertion Sort	$O(n^2)$	$O(n)$ Ordenados	$O(n^2)$ Ordenados de forma inversa	
Heap Sort	$O(n \log n)$	—	—	
Quick Sort	$O(n \log n)$	—	$O(n^2)$ Árbol totalmente desequilibrado	
Merge Sort	$O(n \log n)$	—	—	$M(n)$
Radix Sort	$O(wn)$	—	—	$M(n) - \text{dinámica}$
Counting Sort	$O(n)$	—	—	$M(r)$

Bubble Sort	Propósito general	
Selection Sort		
Insertion Sort		
Heap Sort		
Quick Sort		
Merge Sort	Datos <u>divisibles</u> en partes que tienen <u>pocos</u> valores diferentes:	<ul style="list-style-type: none"> - Números con pocos dígitos Digito de la unidad, decena, centena, ... - Fechas en: Dígitos correspondientes a la unidad y decena del día Dígitos correspondientes a la unidad y decena del mes Dígitos correspondientes a la unidad, decena, centena, millar del año - Palabras, se dividen en letras
Radix Sort		
Counting Sort		

5. Búsqueda en arreglos

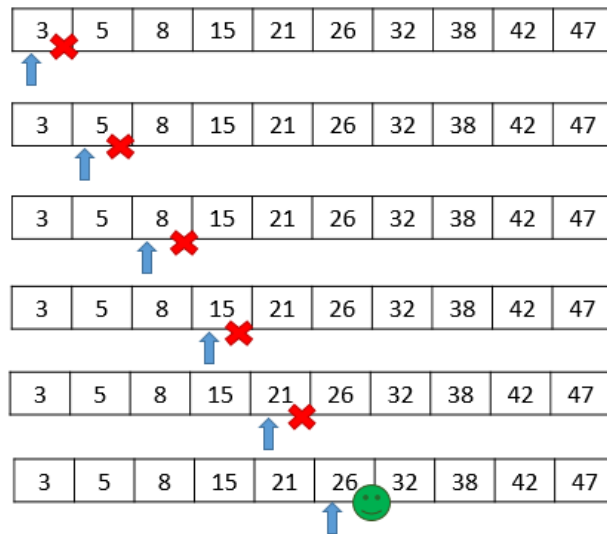
5.1 Búsqueda secuencial

Consiste en buscar un elemento, verificando elemento por elemento de forma secuencial hasta lograr el objetivo.

Ventaja (+) Los elementos pueden o no estar ordenados

Desventaja (-) Muy ineficiente

Ej. Búsqueda del número 26:



5.2 Búsqueda binaria

Permite buscar un elemento específico en una lista ordenada. Utiliza la técnica divide y vencerás.

Ventaja (+) El algoritmo búsqueda binaria es más eficiente que búsqueda secuencial
B. secuencial $O(n)$ – Búsqueda lineal $O(\log n)$

Desventaja (-) Los elementos DEBEN estar ordenados

Desventaja (-) No debe haber claves repetidas

Algoritmo de búsqueda binaria:

Elemento a buscar: buscado

Arreglo en el que se busca: arreglo

inicio = primer índice

fin = último índice

Mientras (inicio ≤ fin)

$$\text{centro} = \frac{\text{fin} + \text{inicio}}{2}$$

Si $e == \text{arreglo}[\text{centro}]$

Regresar el elemento

Si no, Si $e < \text{arreglo}[\text{centro}]$

fin = centro - 1

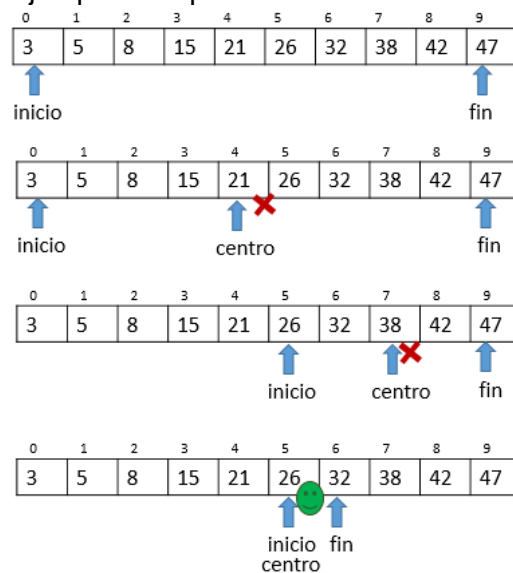
Si no, Si $e > \text{arreglo}[\text{centro}]$

inicio = centro + 1

Fin mientras

Regresar un dato nulo o mensaje de elemento no encontrado

Ejemplo: búsqueda del número 26



5.3 Búsqueda por transformación de claves (Hashing)

Consiste en asignar a cada elemento un índice que se obtiene mediante la transformación del mismo elemento utilizando una función de conversión llamada función **hash**. Lo ideal es que la función hash tuviera la propiedad de que a cada elemento le correspondiera un índice y a cada índice le correspondiera un elemento. De esta forma, no es necesaria una búsqueda, si no que se accede al índice del elemento de forma directa.

Como se puede ver, lo importante es el diseño de la función hash, existen algunas técnicas como:

Clave compuesta: Son claves numéricas para los elementos de conjuntos donde existen bloques de tamaño consecutivo. Por ejemplo, una forma de generar los id para los universitarios suponiendo que el ingreso a la universidad en cada ciclo escolar no excede los 700 alumnos es la siguiente:

Año de ingreso + Número del 0 al 700 (El año inicial podemos tomarlo como 2000)

Ejemplos:

Id	Índice
2000001	1
2004356	3156
2015492	10992

Entonces la función hash sería

$$\text{índice} = (\text{primeros cuatro dígitos del año} - 2000) * 700 + (\text{tres últimos dígitos})$$

Desventaja (-) Podrían quedar algunos índices vacíos, por lo se reserva memoria que no se utiliza.

Truncamiento: Se crea una clave donde una parte de ella es el índice. Por ejemplo, una forma para crear el id de los universitarios podría ser:

Año ingreso + Carrera + índice

Donde el índice corresponde al índice verdadero donde se guarda el elemento.

Ejemplos:

Id	Índice
2004IIA003912	3912
2014IID097521	97521
2015IM099823	99823

Operaciones matemáticas: Existen diversas formas de crear claves utilizando funciones matemáticas como:

- **Módulo:** El índice es el módulo de la clave entre un número fijo (preferentemente primo). Por ejemplo si el número primo es 13. En este caso sólo se podrán tener N índices distintos

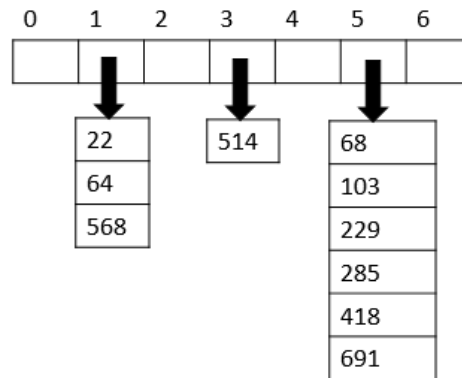
Clave	Índice
89725	12
37630	8
34969	12

- **Plegamiento:** Se divide la clave en diferentes partes para después sumarlas y obtener la clave. Por ejemplo:

Clave: 129 234 987
 Índice: $129 + 234 + 987 = 1350$

Desventaja (-) Puede haber **colisiones**, esto es, que dos o más elementos tengan el mismo índice. Una forma de manejar esto sería de la siguiente forma:

Los elementos que tengan el mismo índice se almacenan en una lista, entonces, la búsqueda de un elemento de un índice en específico se podría hacer de forma secuencial o binaria.



ACTIVIDAD:

Realice un programa que almacene la información de empleados. Por cada empleado se debe almacenar: clave, nombre(s), apellidos y fecha de ingreso. Se puede suponer que la empresa comenzó sus operaciones en el año de 2000 y cada año contrata a lo más a 100 empleados.

- Genere una función hash utilizando la técnica de clave compuesta o truncamiento.
- Genere aleatoriamente una lista inicial con 50 empleados, con todo y su clave.
- El programa debe tener un menú con las siguientes opciones:
 1. Mostrar la lista de empleados.
Incluyendo: clave, nombre, apellido paterno, apellido materno y fecha de ingreso.
 2. Agregar un nuevo empleado.
Solicitando: nombre, apellidos y fecha de ingreso, generando la clave automáticamente.
 3. Buscar un empleado en base a la clave.
 0. Salir

6. Algoritmos sobre cadenas de caracteres

Es un arreglo ordenado de caracteres (char).

Ejemplo: String cadena = "Universidad";

En memoria se representa:

	0	1	2	3	4	5	6	7	8	9	10	11
Cadena =	U	n	i	v	e	r	s	i	d	a	d	\0

Algoritmos de búsqueda de cadenas

Un algoritmo de búsqueda de cadenas tiene como objetivo encontrar una subcadena o patrón dentro de otra cadena de caracteres o texto.

Ejemplo:

String texto = "Hola como estas, yo soy una cadena de muestra";

String patron = "estas"

```
int posicion = Busqueda( texto, patron );
```

```
print(posicion);
```

```
10
```

Estos algoritmos comúnmente regresan la posición del carácter inicial del patrón dentro del texto y en caso de no encontrar el patrón retornan un valor que indique que no se encontró, como -1.

Algoritmos de comparación de cadenas

Los algoritmos de comparación de cadenas tienen como objetivo calcular la distancia entre dos cadenas de caracteres.

Ejemplo: Comparación de cadenas con el mismo número de caracteres:

Distancia("casa", "casa") = 0

Distancia("casa", "cAsA") = 2

Distancia("casa", "capa") = 1

Ejemplo: Comparación de cadenas con cadenas de diferentes longitudes

Distancia("casa", "casita") = 4

La distancia entre las dos cadenas de caracteres depende del algoritmo utilizado.

6.1 Búsqueda secuencial

Es un algoritmo para búsqueda de cadenas de caracteres. Lo que hace es buscar posición por posición el patrón dentro del texto. Su complejidad es $O(n * m)$, n es el tamaño del texto y m es el tamaño del patrón.

Ejemplo:

Texto: "Hola como estas"

Patrón: "como"

Para $i = 0$

H	o	l	a		c	o	m	o	
	c	o	m	o					
	x	v	x	x					

¿Coincide? No

Para $i = 1$

H	o	l	a		c	o	m	o	
		c	o	m	o				
		x	x	x	x				

¿Coincide? No

...

Para $i = 5$

H	o	l	a		c	o	m	o	
					c	o	m	o	
					v	v	v	v	

6.2 Búsqueda de cadenas: Knutt – Morris – Pratt

Es un algoritmo para búsqueda de caracteres que fue elaborado por Donald Knuth, Vaughan Pratt y James H. Morris en 1977.

Lo que hace es buscar un patrón dentro de un texto en base a “estados” para realizar la búsqueda de forma eficiente. Su complejidad es $O(n + m)$, n es el tamaño del texto y m es el tamaño del patrón.

¿Cómo lo hace?

Paso 1. Analizar el patrón calculando:

- El número de estados posibles
- El carácter en cada estado
- El estado al que se regresa en caso de un error

a) Los estados se inicializan con números desde el cero hasta el número de caracteres menos uno (con salto de uno), en cada estado se pone un carácter del patrón en orden y se inicializa el primer regreso al estado -1.

Ejemplo:

Patrón: xoxoxxo

estado	0	1	2	3	4	5	6
	x	o	x	o	x	x	o
regreso	-1						

b) Para calcular los regresos de los estados siguientes (1 en adelante):

Iniciar la variable k en -1

Analizar uno a uno los estados para calcular su regreso:

Mientras $k > -1$ y el carácter en $k+1$ no es igual al carácter del estado_actual

$k =$ regreso que marca el estado k

Si carácter en $k+1$ es igual al carácter del estado actual

$k = k + 1$

regreso en el estado actual es igual a k

Si no

Regreso en el estado actual es igual a -1

Ejemplo:

Patrón: xoxoxxo

estado	0	1	2	3	4	5	6
	x	o	x	o	x	x	o
regreso	-1	0					
		-1	0	1	2	3	
					0	1	
					-1	0	1

En negro se marcan los regresos que se van asignando, los **rojos** los errores al probar $k+1$, y los **azul** los estados que regresaron.

estado	0	1	2	3	4	5	6
	x	o	x	o	x	x	o
regreso	-1	-1	0	1	2	0	1

Paso 2. Buscar el patrón dentro del texto.

Se inicia en el estado -1 y en el primer carácter del texto

Analizar uno a uno los caracteres del texto:

Mientras el estado > -1 y carácter del estado+1 no es igual al carácter del texto

 estado = regreso del estado

Si carácter del estado+1 es igual al carácter del texto

 estado = estado + 1

 avanzar un carácter

 Si el estado == tamaño del patrón a buscar, entonces ya se ha encontrado 😊

Si no

 avanzar un carácter

Ejemplo:

Patrón: xoxoxxo

Texto: xoxxoxxoxxoxxoxxoxxoxxo

Paso 1:

estado	0	1	2	3	4	5	6
	x	o	x	o	x	x	o
regreso	-1	-1	0	1	2	0	1

Paso 2:

	x	o	x	x	o	o	x	x	o	x	x	x	o	x	o	o	x	o	x	o	x	o	x	x	o
-1	0	1	2	3																					
		0	1																						
		-1	0	1	2																				
			-1	0																					
				-1	0	1																			
					-1	0	1	2	3																
						0	1																		
						-1	0	1																	
							-1	0	1	2	3	4													
											1	2													
											-1	0													
												-1	0	1	2	3	4	5							
																2	3	4	5	6					

6.3 Búsqueda de cadenas: Boyer – Moore - Horspool

Es un algoritmo para búsqueda de caracteres. Inicialmente, Bob Boyer y J. Strother Moore publicaron su algoritmo de búsqueda de cadenas Boyer-Moore en 1977, y después Nigel Horspool mejoró dicho algoritmo en 1980.

Al igual que el algoritmo anterior, busca un patrón dentro de un texto en base a “saltos” para realizar la búsqueda de forma eficiente. Su complejidad en promedio es $O(n + m)$, n es el tamaño del texto y m es el tamaño del patrón, pero en el peor de los casos se tiene una complejidad de $O(nm)$.

Ejemplo de búsqueda:

Texto: ababbaccabcaabb

Patrón: cabcaab

Inicia buscando en la posición 0, comparando de derecha a izquierda

Texto:	a	b	a	b	b	a	c	c	a	b	c	a	a	b	b
Patrón:	c	a	b	c	a	a	b								

Busca el carácter del texto en el patrón y mueve el patrón hasta que coinciden

Texto:	a	b	a	b	b	a	c	c	a	b	c	a	a	b	b
Patrón:				c	a	b	c	a	a	b					

Paso 1. Analizar los caracteres del patrón.

Calcula cuantos espacios se deben recorrer, en caso de que no coincida el patrón con el texto, según el carácter del texto encontrado.

a) Se crea una tabla con los caracteres del patrón más un espacio para cualquier otro carácter, y se inicializan todos los valores con el tamaño del patrón

Ejemplo:

Patrón: cabcaab

a	b	c	Otro
7	7	7	7

b) Se verifican uno a uno los caracteres del patrón con $i=1$ hasta el tamaño del patrón $- 1$, y se actualiza la casilla del carácter con el tamaño del patrón $- i$.

Ejemplo:

Patrón: cabcaab

Valores iniciales:

a	b	c	Otro
7	7	7	7

1	2	3	4	5	6	7	a	b	c	Otro
c	a	b	c	a	a	b	7	7	6	7
1	2	3	4	5	6	7	a	b	c	Otro
c	a	b	c	a	a	b	5	7	6	7
1	2	3	4	5	6	7	a	b	c	Otro
c	a	b	c	a	a	b	5	4	6	7
1	2	3	4	5	6	7	a	b	c	Otro
c	a	b	c	a	a	b	5	4	3	7
1	2	3	4	5	6	7	a	b	c	Otro
c	a	b	c	a	a	b	2	4	3	7
1	2	3	4	5	6	7	a	b	c	Otro
c	a	b	c	a	a	b	1	4	3	7

a	b	c	Otro
1	4	3	7

Paso 2: Buscar el patrón en el texto

Alinear el texto con el patrón, comenzando con las posiciones en 0

Repetir

Guardar el carácter del texto que se encuentre más a la derecha al alinearlo con el patrón

Comparar de derecha a izquierda los caracteres del texto y del patrón

En caso de que no coincidan

Se debe mover el patrón el número de espacios que marque el carácter del texto guardado

Mientras no coincidan

Ejemplo:

Texto: ababbaccabcaabb

Patrón: cabcaab

Paso 1:

a	b	c	Otro
1	4	3	7

Paso 2:

Texto:

a	b	a	b	b	a	c	c	a	b	c	a	a	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Patrón:

c	a	b	c	a	a	b								
---	---	---	---	---	---	---	--	--	--	--	--	--	--	--

↑

Falló, mover 3 espacios

Texto:

a	b	a	b	b	a	c	c	a	b	c	a	a	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Patrón:

				c	a	b	c	a	a	b				
--	--	--	--	---	---	---	---	---	---	---	--	--	--	--

↑

↑

↑

Falló, mover cuatro espacios

Texto:

a	b	a	b	b	a	c	c	a	b	c	a	a	b	b
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Patrón:

							c	a	b	c	a	a	b	
--	--	--	--	--	--	--	---	---	---	---	---	---	---	--

↑

↑

↑

↑

↑

↑

↑

6.4 Comparación de cadenas: Distancia de Hamming

Se denomina así por su inventor Richard Hamming, profesor de la Universidad de Nebraska. Es un algoritmo sencillo para **calcular la distancia entre dos cadenas de caracteres de la misma longitud**. Su complejidad es $O(n)$, donde n es el tamaño de las cadenas.

Lo que hace es calcular el número de errores, o caracteres que no corresponden.

Ejemplo:

Distancia_Hamming ("casa", "casa") -> 0

Distancia_Hamming ("capa", "casa") -> 1

Distancia_Hamming ("casa", "casita") -> No válido

6.5 Comparación de cadenas: Distancia de Levenshtein

La distancia de Levenshtein debe su nombre en honor al científico ruso Vladimir Levenshtein quién se ocupó de esta distancia en 1965. Su complejidad es $O(m*n)$, donde m es el tamaño de la primer cadena y n es el tamaño de la segunda cadena.

Consiste en calcular el mínimo número de operaciones para transformar una cadena de caracteres en otra cadena de caracteres. Las operaciones por carácter son:

- Sustitución
- Inserción
- Eliminación

Ejemplo:

Distancia_Levenshtein ("casa", "casa") -> 0

Distancia_Levenshtein("capa", "casa") -> 1 (sustitución p por s)

Distancia_Levenshtein("casa", "casita") -> 2 (inserción i, inserción t)

Algoritmo

Paso 1:

Construir una tabla D, con

#filas = longitud de la primer cadena + 1 y

#columnas = longitud de la segunda cadena + 1.

Inicializar en cero todas las celdas

La primera columna se inicializa con los números de 0 hasta el tamaño de la primera cadena

La primera fila se inicializa con los números desde 0 hasta el tamaño de la segunda cadena

Ejemplo: Comparación de casa y casita

		c	a	s	i	t	a	
		0	1	2	3	4	5	6
c		1						
a		2						
s		3						
a		4						

Paso 2:

Recorrer i desde 1 hasta la longitud de la primera cadena

Recorrer j desde 1 hasta la longitud de la segunda cadena

Si $\text{primercadena}[i] == \text{segundacadena}[j]$, entonces $\text{costo} = 0$

Si no $\text{costo} = 1$

$d[i,j] = \text{mínimo de:}$

$D[i-1, j] + 1$, // eliminación

$D[i, j-1] + 1$, // inserción

$D[i-1, j-1] + \text{costo}$ // sustitución

Finalmente el costo está en la última fila y la última columna de la tabla.

Ejemplo: Comparación de casa y casita

		c	a	s	i	t	a	
		0	1	2	3	4	5	6
c	1	2	0					
a	2							
s	3							
a	4							

		c	a	s	i	t	a	
		0	1	2	3	4	5	6
c	1	0	1					
a	2							
s	3							
a	4							

		c	a	s	i	t	a	
		0	1	2	3	4	5	6
c	1	0	1	2				
a	2							
s	3							
a	4							

El algoritmo anterior funciona cuando la inserción, eliminación y sustitución tienen el mismo peso, además de que no es importante que letras se eliminan, sustituyen e insertan. Sin embargo, en algunas ocasiones es necesario cambiar estas ponderaciones, en general el algoritmo es:

D_{FxC} , $F = \# \text{Caracteres en la palabra 1}$, $C = \# \text{Caracteres en la palabra 2}$

$$D[i][0] = \sum_{k=1}^i \text{Costo de eliminar a la letra de la posición } k \text{ de la palabra 1}$$

$$D[0][j] = \sum_{k=1}^j \text{Costo de insertar a la letra de la posición } k \text{ de la palabra 2}$$

$$D[i][j] = \min \begin{cases} D[i][j] + \text{Costo de sustituir la letra } i \text{ de la palabra 1 por la letra } j \text{ de la palabra 2} \\ D[i-1][j] + \text{Costo de eliminar la letra } i \text{ de la palabra 1} \\ D[i][j-1] + \text{Costo de insertar la letra } j \text{ de la palabra 2} \end{cases}$$

7. Dynamic Programming (Programación dinámica)

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the subproblem is encountered.

Example 1: Fibonacci

The Fibonacci is defined as:

$$Fibonacci(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ Fibonacci(n-1) + Fibonacci(n-2) & \text{si } n > 1 \end{cases}$$

DP: Fibonacci(4)

n	Fibonacci(n)
0	0

n	Fibonacci(n)
0	0
1	1

n	Fibonacci(n)
0	0
1	1
2	1

n	Fibonacci(n)
0	0
1	1
2	1
3	2

n	Fibonacci(n)
0	0
1	1
2	1
3	2
4	3

n	Fibonacci(n)
0	0
1	1
2	1
3	2
4	3
5	5

Example 2: Binomial coefficients

$$(a+b)^1 = 1a + 1b$$

$$(a+b)^2 = 1a^2 + 2ab + 1b^2$$

$$(a+b)^3 = 1a^3 + 3a^2b + 3ab^2 + 1b^3$$

$$(a+b)^4 = 1a^4 + 4a^3b + 6a^2b^2 + 4ab^3 + 1b^4$$

	k=1	k=2	k=3	k=4	k=5
n=1	1	1			
n=2	1	2	1		
n=3	1	3	3	1	
n=4	1	4	6	4	1

The binomial coefficients can be calculated using DP as follows:

$$C(n, k) = \begin{cases} 1 & \text{si } k = 1 \\ 1 & \text{si } k = n + 1 \\ C(n-1, k-1) + C(n-1, k) & \text{si } 1 < k < n + 1 \end{cases}$$

where: $1 \leq k \leq n + 1$

Ejemplo 3: Intercambio de monedas

Dada una cantidad solicitada, se debe calcular el tipo y la cantidad de monedas a entregar para cumplir con dicha cantidad, utilizando el menor número de monedas posibles.

Parámetros: - La cantidad que se debe completar
- Valores de los diferentes tipos de monedas.

Retorno: - Tipo y cantidad de monedas

Ejemplos:

IntercambioMonedas(\$12, [1,6,10]) → [0,2,0] // 2 monedas de \$6

IntercambioMonedas(\$20, [1,5,10,20]) → [0,0,0,1] //1 moneda de \$20

Para resolver este problema se debe generar una tabla con las siguientes características:

Filas: Indican los tipos de monedas que se tienen

Columnas: Desde el valor mínimo monetario que se puede entregar hasta la cantidad que se requiere

Cada celda debe indicar la mínima cantidad de monedas necesarias (de igual o menor numeración de la moneda de la fila) para completar la cantidad de la columna.

Ejemplo: IntercambioMonedas(\$12, [1,2,5,10]) – Tabla dinámica

		Cantidad solicitada											
Monedas		\$1	\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9	\$10	\$11	\$12
	\$1												
	\$2												
	\$5												
	\$10												

Luego se debe llenar la primera fila con la cantidad de monedas que se necesitan para cumplir con la cantidad de cada columna, en caso de que no se pueda cubrir la cantidad se llena con infinito.

Ejemplo: IntercambioMonedas(\$12, [1,2,5,10]) – Llenado de la primera fila

		Cantidad solicitada											
Monedas		\$1	\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9	\$10	\$11	\$12
	\$1	1	2	3	4	5	6	7	8	9	10	11	12
	\$2												
	\$5												
	\$10												

Ejemplo: IntercambioMonedas(\$10, [2,5,10]) – Llenado de la primera fila

		Cantidad solicitada								
Monedas		\$2	\$3	\$4	\$5	\$6	\$7	\$8	\$9	\$10
	\$2	1	Inf	2	Inf	3	Inf	4	Inf	5
	\$5									
	\$10									

Luego para las siguientes filas se decide el valor de cada celda de acuerdo al siguiente algoritmo

Si (valor de la moneda(fila) > cantidad(columna))

Se toma el resultado de la casilla superior

Si no, (valor de la moneda(fila) <= cantidad(columna))

Se toma el mínimo entre:

- Pagar con monedas de numeración más baja, # Monedas = # Monedas casilla superior

- Pagar con **una** moneda del tipo actual y el resto con:

el resultado calculado con la moneda actual (fila) y la cantidad restante, es decir:

Cantidad restante = Cantidad solicitada - ValorMoneda

#Monedas = 1 + # monedas cantidad restante

Ejemplo: IntercambioMonedas(\$6, [1,2]) – Llenado de las celdas

Cantidad solicitada

Monedas		\$1	\$2		\$3		\$4		\$5		\$6		...
	\$1	1	2		3		4		5		6		
	\$2	1	2	1	3	2	4	2	5	3	6	3	
			1		2		2		3		3		

Para calcular las denominaciones de las monedas, además de la cantidad de monedas, basta con agregar con arreglo con el número de monedas de cada denominación.

Ejemplo: IntercambioMonedas(\$6, [1,2]) – Llenado de las celdas

Cantidad solicitada

Monedas		\$1	\$2	\$3	\$4	\$5	\$6	...					
	\$1	1 {1,0}	2 {2,0}	3 {3,0}	4 {4,0}	5 {5,0}	6 {6,0}						
	\$2	1 {1,0}	2 1	1 2	3 2	2	4 2	3	5 3	3	6 3	3	
			{0,1}		{1,1}		{0,2}		{1,2}		{0,3}		

En esta caso el valor óptimo para entregar la cantidad de \$6 con monedas de denominación \$1 y \$2. Es 3{0,3}, es decir: 3 monedas, 0 de \$1 y 3 de \$2.

Ejemplo 4: Problema de la mochila

Se tiene una mochila con una capacidad limitada en peso y el objetivo es guardar objetos en la mochila de tal forma que:

- Se maximice la suma de los precios de los objetos
- No se exceda la capacidad en peso de la mochila.

Para esto se tienen n objetos con peso y precio determinado.

Parámetros:

- Capacidad de la mochila (en peso)
- Peso, precio y cantidad de objetos de tipo 1
- Peso, precio y cantidad de objetos de tipo 2
- ...

Retorno: - Tipo y cantidad de objetos a guardar en la mochila

El primer paso consiste en crear la tabla dinámica con las siguientes características:

Filas: Cada fila representa un objeto

Columnas: Cada columna representa el peso, va desde el peso mínimo hasta la capacidad de la mochila

Celda: Indica la cantidad de objetos que deben estar en la mochila de acuerdo al tipo de objetos disponibles y la capacidad.

Ejemplo: Tabla dinámica

Capacidad de la mochila 15kg

Objetos:

- 1 objetos de 1 kg con un valor de \$5
- 2 objetos de 1 kg con un valor de \$30
- 1 objetos de 2 kg con un valor de \$20

	1	2	3	4	5	6	7	8	9
1kg, \$5									
1kg, \$30									
1kg, \$30									
2kg, \$30									

Luego se debe llenar la primera fila con la ganancia de poner un objeto de ese tipo.

Ejemplo:

	1	2	3	4	5	6	7	8	9
1kg, \$5	5	5	5	5	5	5	5	5	5
	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}
1kg, \$30									
1kg, \$30									
2kg, \$30									

Luego para las siguientes filas se decide el valor de cada celda de acuerdo al siguiente algoritmo

Si (peso del objeto(fila) > capacidad de la mochila(columna))
Se toma el resultado de la casilla superior

Si no, (peso del objeto(fila) <= capacidad de la mochila(columna))
Se toma la ganancia máxima de:
- No utilizar ese objeto, es decir, tomar el resultado de la casilla superior
- Si utilizar el objeto, tomando **un** objeto del tipo actual y la capacidad restante de la mochila con:
el resultado calculado en la fila superior, es decir, fila: anterior, columna: capacidad restante
Capacidad restante = Capacidad mochila(columna) – Peso del objeto(fila)

Ejemplo:

	1	2	3	4	5	6	7	8	9
1kg, \$5	5	5	5	5	5	5	5	5	5
	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}	{1,0,0,0}
1kg, \$30	30	35	35	35	35	35	35	35	35
	{0,1,0,0}	{1,1,0,0}	{1,1,0,0}	{1,1,0,0}	{1,1,0,0}	{1,1,0,0}	{1,1,0,0}	{1,1,0,0}	{1,1,0,0}
1kg, \$30	30	60	65	65	65	65	65	65	65
	{0,0,1,0}	{0,1,1,0}	{1,1,1,0}	{1,1,1,0}	{1,1,1,0}	{1,1,1,0}	{1,1,1,0}	{1,1,1,0}	{1,1,1,0}
2kg, \$30	30	60	65	90					
	{0,0,1,0}	{0,1,1,0}	{1,1,1,0}	{0,1,1,1}					

8. Greedy Algorithms (algoritmos voraces)

Son algoritmos que resuelven problemas de optimización (minimización o maximización) de una forma sencilla, pero no siempre garantizan la mejor solución. También son conocidos como algoritmos ávidos, devoradores o golosos. De forma general lo que hace es tomar la mejor decisión en cada paso basado en una heurística.



¡Cómete siempre todo lo que tengas a mano!

Algoritmos Greedy. Departamento de Ciencias de la Computación e I.A. Universidad de Granada. (<http://elvex.ugr.es/decsai/algorithms>)

Ejemplo 1: Intercambio de monedas

Dada una cantidad solicitada, se debe calcular el tipo y la cantidad de monedas a entregar para cumplir con dicha cantidad, utilizando el menor número de monedas posibles.

Parámetros: - La cantidad que se debe completar
- Valores de los diferentes tipos de monedas.

Retorno: - Tipo y cantidad de monedas

Ejemplos:

IntercambioMonedas(\$12, [1,6,10]) \rightarrow [0,2,0] // 2 monedas de \$6

IntercambioMonedas(\$20, [1,5,10,20]) \rightarrow [0,0,0,1] //1 moneda de \$20

Para resolver este problema utilizando un algoritmo voraz lo que se hace es: utilizar el máximo número posible de monedas de mayor denominación, luego el máximo número posible de monedas de la siguiente denominación y así sucesivamente.

Ejemplo: IntercambioMonedas(\$87, [1,2,5,10,50])

Cantidad inicial: \$87

Paso	Denominación	Número monedas	Cantidad
1	50	1	37
2	10	3	7
3	5	1	2
4	2	1	0
5	1	0	0

Entonces para cubrir la cantidad de \$87 se devuelve 1(\$50), 3(\$10), 1(\$5), 1(\$2)

Ejemplo 2: Problema de la mochila

Se tiene una mochila con una capacidad limitada en peso y el objetivo es guardar objetos en la mochila de tal forma que:

- Se maximice la suma de los precios de los objetos
- No se exceda la capacidad en peso de la mochila.

Para esto se tienen n objetos con peso y precio determinado.

Parámetros:

- Capacidad de la mochila (en peso)
- Peso, precio y cantidad de objetos de tipo 1
- Peso, precio y cantidad de objetos de tipo 2
- ...

Retorno:

- Tipo y cantidad de objetos a guardar en la mochila

Para resolver el problema de la mochila utilizando un algoritmo voraz lo que se hace es:

1. Calcular la ganancia por kilo de cada objeto
2. Agregar a la mochila objeto por objeto, comenzando con los objetos con la mayor ganancia por kilo

Ejemplo:

Capacidad de la mochila 5kg

Objetos:

- A - 2 objetos de 1 kg con un valor de \$5
- B - 2 objetos de 1 kg con un valor de \$30
- C - 2 objetos de 2 kg con un valor de \$20

Ganancia por kilo

Objeto	Ganancia en pesos	Peso en kg	Ganancia/peso
A	5	1	5
B	30	1	30
C	20	2	10

Agregar objetos a la mochila

Capacidad de la mochila: 5kg

Paso	Objeto – Ganancia/peso (Ordenados de mayor a menor)	# objetos disponibles	Peso del objeto	Solución Número de objetos	Capacidad en la mochila
1	B – 30	2	1	2	3 kg
2	C – 10	2	2	1	1 kg
3	A – 5	2	1	0	0

Resultado: 2 objetos tipo B y 1 objetos tipo C

EJERCICIOS

1. Suponemos que se tienen las siguientes actividades:

Actividad	Horario	Prioridad
Celebrar el aniversario con tu novio(a)	17 – 19 hrs	3
Asistir a clase de Algoritmos	9 – 11 hrs	1
Ver el partido de México	16 – 18 hrs	5
Comer con tu familia	13 – 15 hrs	2
Asistir a asesorías de Cálculo	12 – 14 hrs	4

Realiza una propuesta del algoritmo que resuelve el problema de selección de actividades basándote en la idea de los algoritmos Greedy.

2. Realiza tres programas utilizando algoritmos Greedy:

- Problema de la Mochila
- Intercambio de monedas
- Selección de actividades

9. Lists

When we want to work with unknown number of data values, we use a linked list data structure to organize that data. Linked list is a linear data structure that contains sequence of elements such that each element links to its next element in the sequence. Each element in a linked list is called as "Node".

A node is a collection of two sub-elements or parts. A **data** part that stores the element and a **link** part that stores the link to the next or previous node.

9.1 Singly Linked List

Single linked list is a sequence of elements in which **every element has link to its next element** in the sequence.



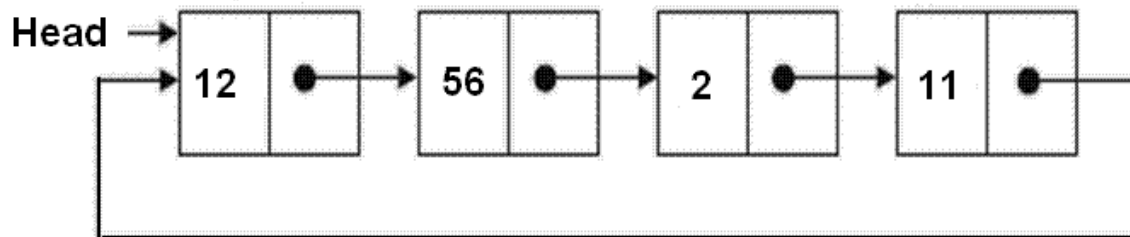
9.2 Doubly Linked List

Circularly linked list is a sequence of elements in which **every element has link to its next element** in the sequence, and the last element also has link to the first one.



9.3 Circularly Linked List

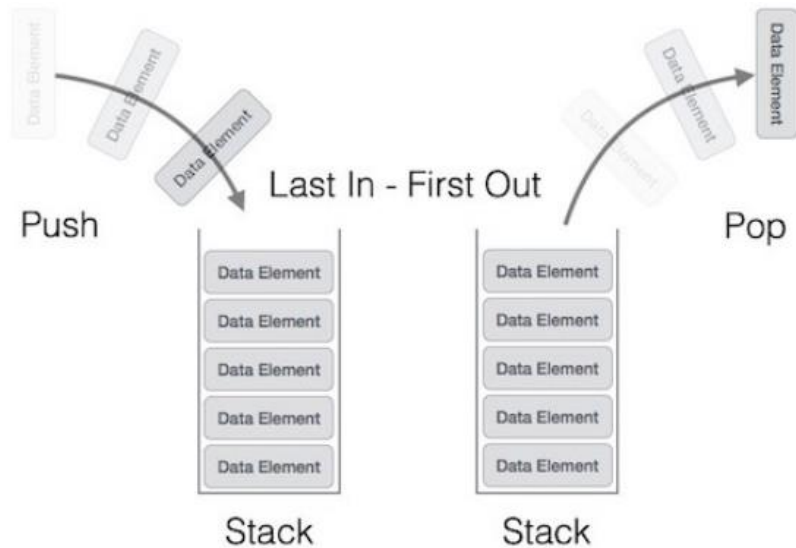
Circularly linked list is a sequence of elements in which **every element has link to its next element** in the sequence, and the last element also has link to the first one.



10. Stacks, Queues and Deques

10.1 Stacks

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (Last In First Out) or FILO (First In Last Out).



Mainly the following three basic operations are performed in the stack:

- **Push:** Adds an item.
- **Pop:** Removes an item.
- **Peek or Top:** Returns top element.
- **isEmpty:** Returns true if stack is empty, else false.

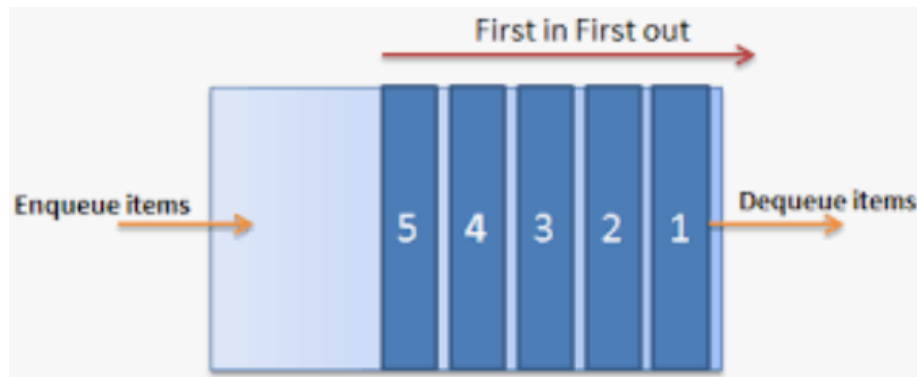
Template implementation:

```
Node<E>
Node next
E e
Node(E e)
```

```
Stack<E>
Node first;
int size;
Stack()
int getSize()
bool isEmpty()
void push(E e)
E top()
void pop()
```

10.2 Queue

Queue is a linear data structure which follows a particular order in which the operations are performed. The order may be FIFO(First In First Out).



Mainly the following three basic operations are performed in the queue:

- **Push:** Adds an item at the back.
- **Pop:** Removes an item from the front.
- **Front:** Returns the first element.
- **isEmpty:** Returns true if queue is empty, else false.

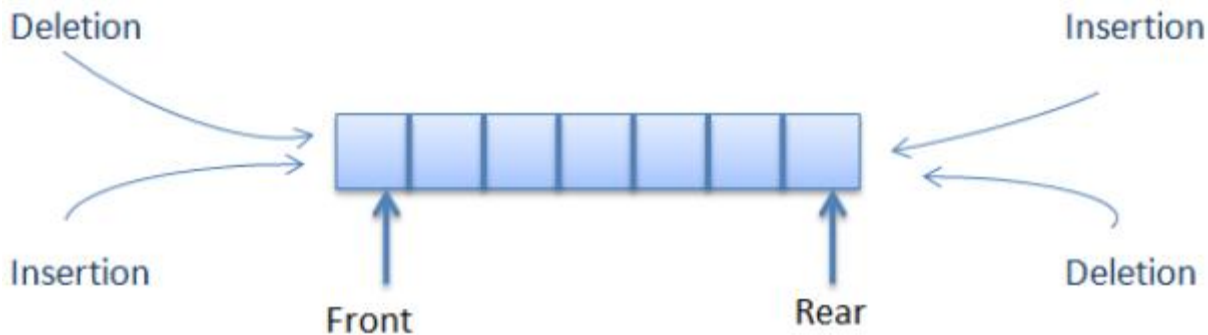
Template implementation:

```
Node<E>
Node next
E e
Node(E e)
```

```
Queue<E>
Node first;
Node last;
int size;
Stack()
int getSize()
bool isEmpty()
void push(E e)
E front()
void pop()
```

10.3 Deques

Double-ended queue or “**deque**” is a data structure that supports insertion and deletion at both the front and the back of the queue.



Mainly the following three basic operations are performed in the queue:

- **addFirst:** Adds an item at the first.
- **addLast:** Adds an item at the back.
- **removeFirst:** Removes and returns the first element.
- **removeLast:** Removes and returns the last element.
- **first:** Returns the first element.
- **last:** Returns the last element.
- **isEmpty:** Returns true if queue is empty, else false.

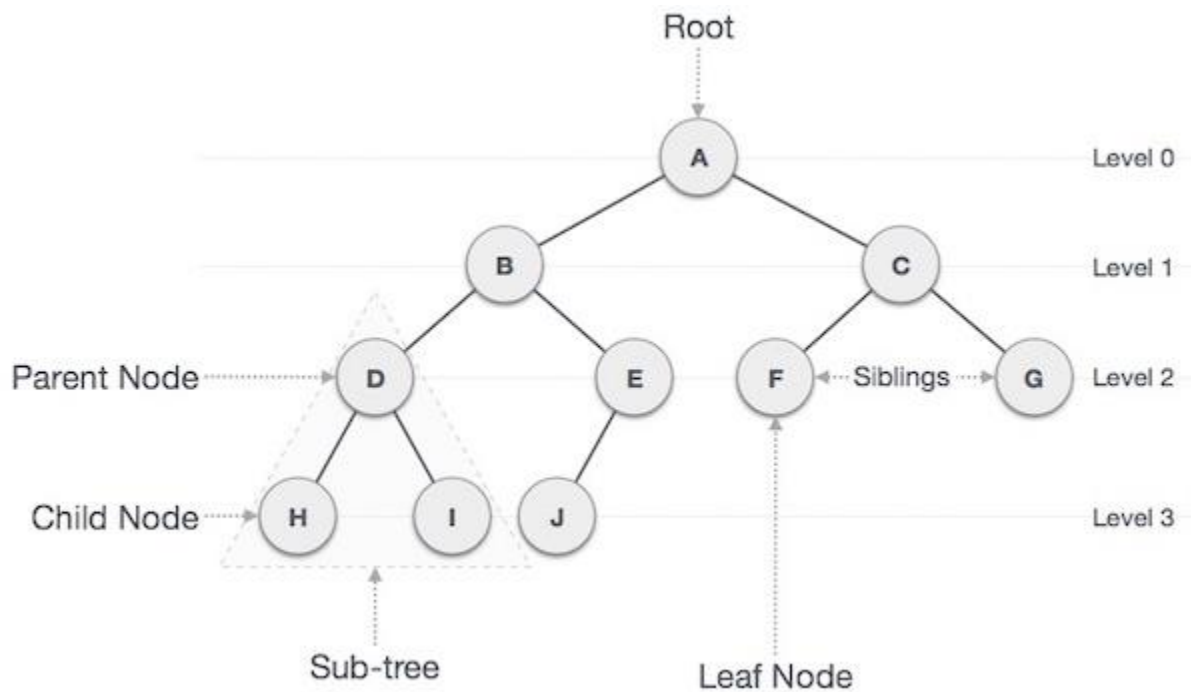
Template implementation:

```
Node<E>
Node next
E e
Node(E e)
```

```
Deque<E>
Node first;
Node last;
int size;
Deque()
int getSize()
bool isEmpty()
void addFirst(E e)
void addLast(E e)
E removeFirst()
E removeLast()
E first()
E last()
```

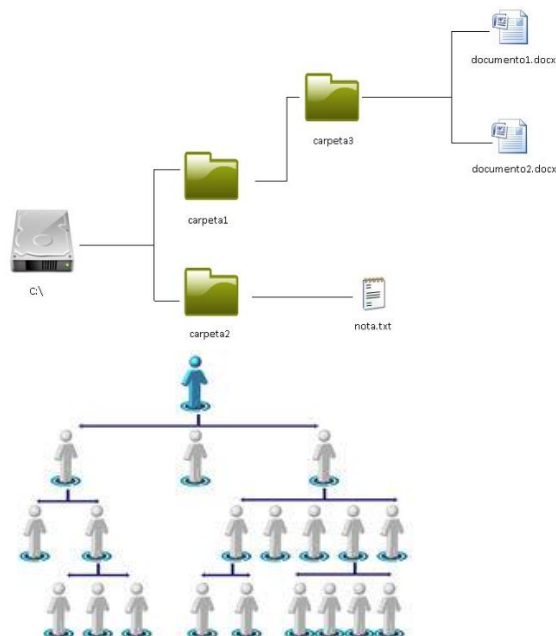
11. Trees

A **tree** is an abstract data type that stores elements hierarchically. With the exception of the top element, each element in the tree has a **parent** element and zero or more **children** elements. We typically call the top element the **root** of the tree.



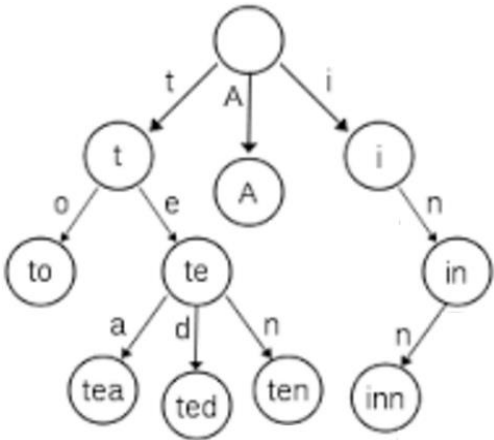
They can be used to represent:

- File system structure.
Nodes represent folders or files.

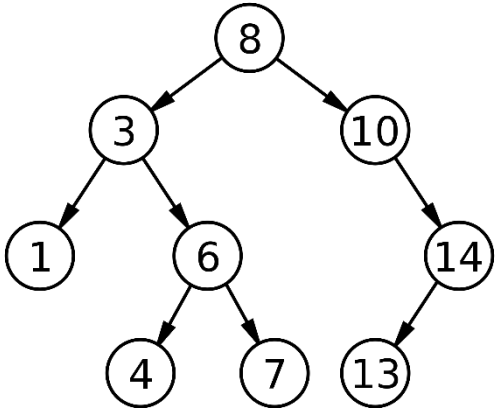


- Hierarchical data

- Trie tree



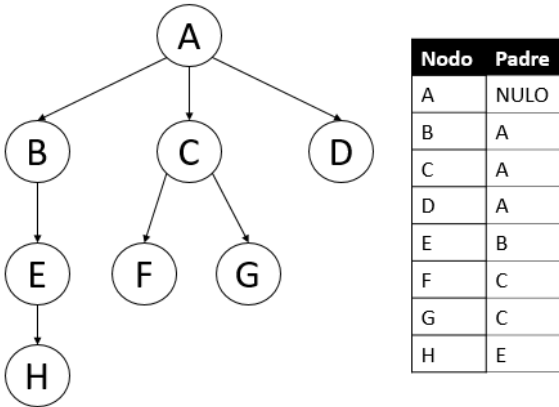
- Binary search tree



11.1 Representation

9.1.1 Arrays

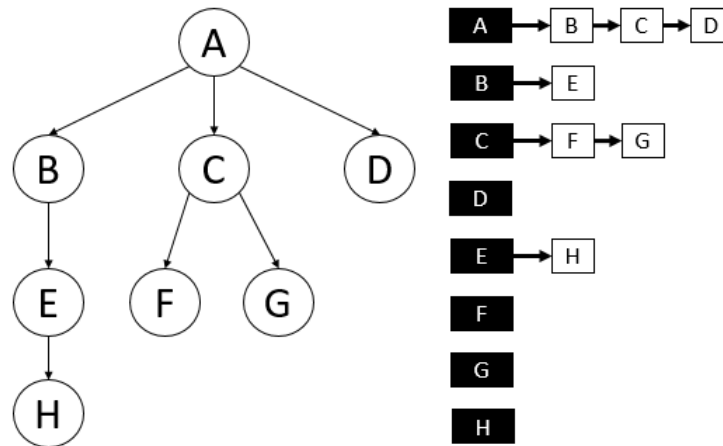
In an array, the parent of each node is stored.



- (+) Resources are optimized.
- (-) The queries and searches are complicated.

9.1.2 Links

Each node has the links to its children.



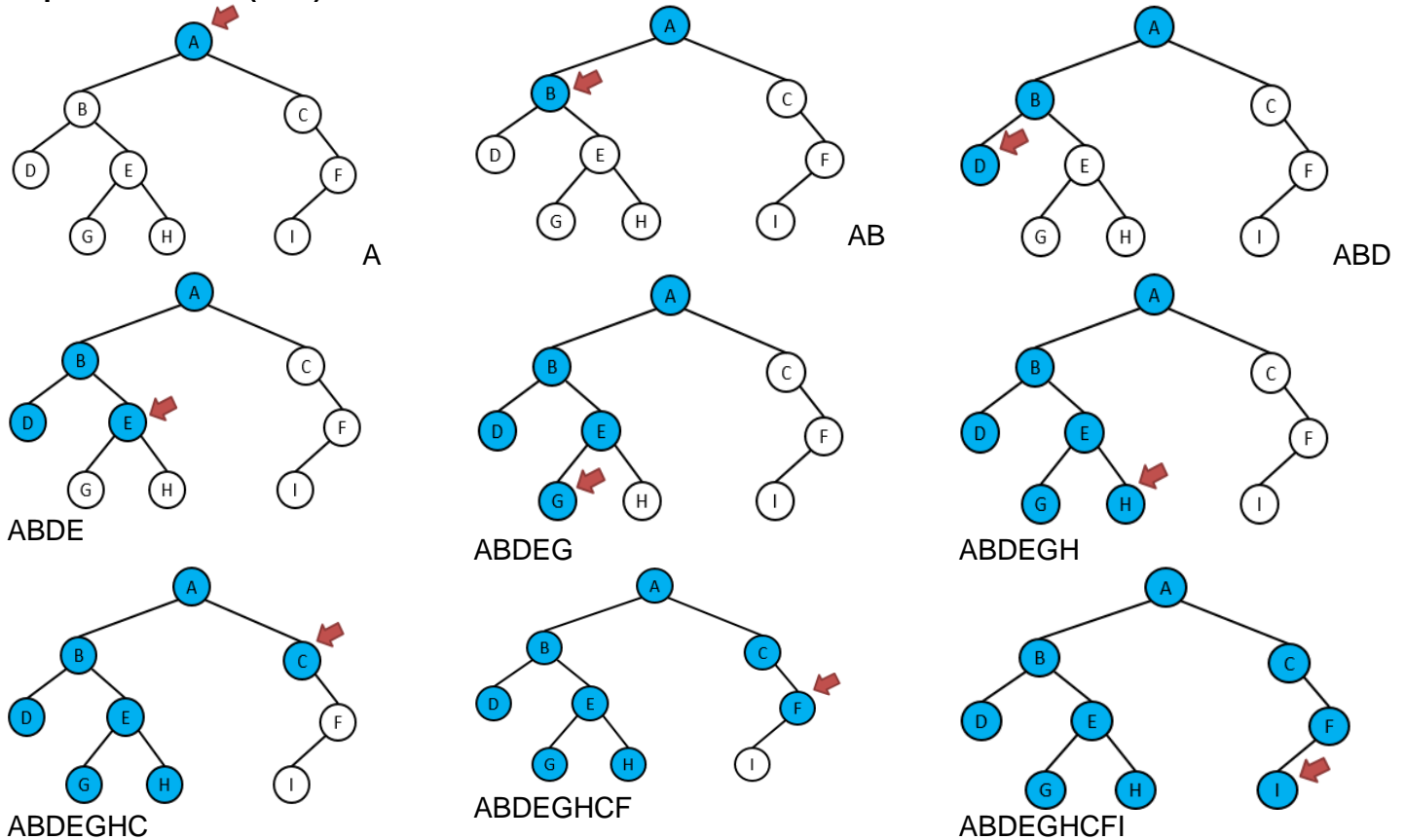
In some implementations, the nodes also have a link to their parents.

- (+) Easy searches and queries.
- (-) Difficult implementation.

11.2 Traversals: DFS and BFS

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

Depth first search (DFS)



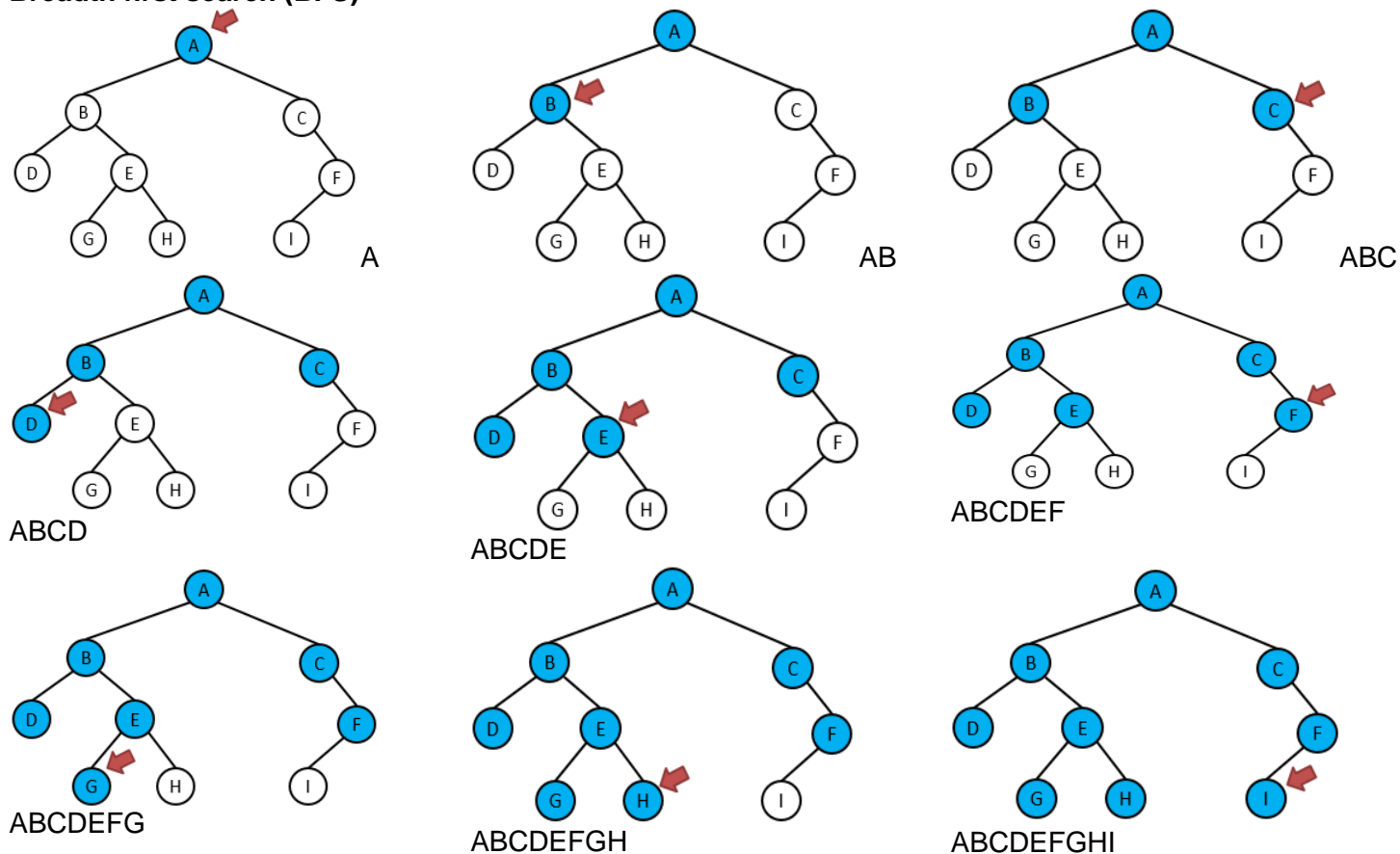
Implementation 1: RECURSION

```
Class Tree :: DFS()
root.DFS()
```

```
Class Node :: DFS()
Print value
For each child
  child.DFS()
```

Implementation 2: While + Stack

```
Class Tree :: DFS()
Create a stack of nodes
Add the root node to the stack
While stack is not empty:
  Get a node from the stack
  Print the node
  For each child of the node
    Add the child to the stack
```

Breadth first search (BFS)**Implementation 1: While + Queue**

```

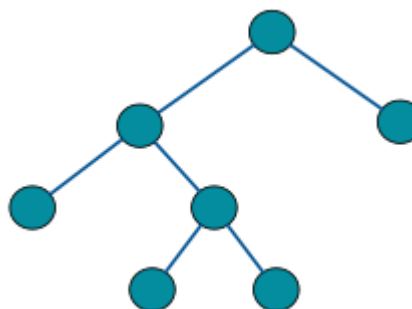
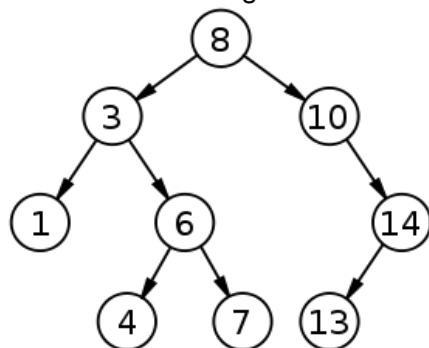
Class Tree :: BFS()
Create a queue of nodes
Add the root node to the queue
While queue is not empty:
    Get a node from the queue
    Print the node
    For each child of the node
        Add the child to the queue
  
```

Cl
Cr
Ag
Mi

11.3 Binary trees

A binary tree is an ordered tree with the following properties:

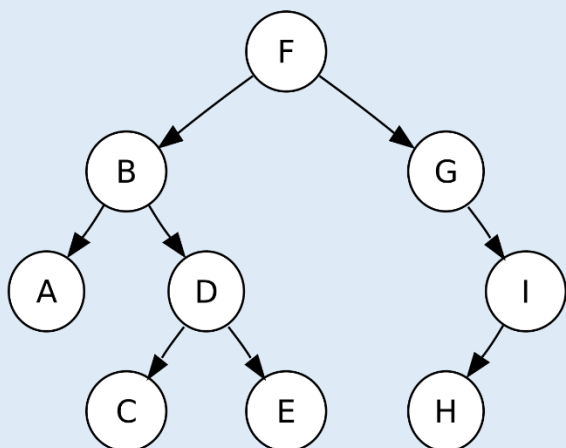
- Every node has at most two children
- Each child node is labeled as being either a left child or a right child.



11.3.1 Traversals: Inorder, preorder and postorder

For a binary tree, there are three depth first traversals:

- **Inorder:** (left, root, right)
- **Preorder:** (root, left, right)
- **Postorder:** (left, right, root)



Inorder:
Preorder:
Postorder:

11.3.2 Binary search tree

Las operaciones importantes en un árbol binario de búsqueda son:

- Inserción
- Eliminación
- Consulta

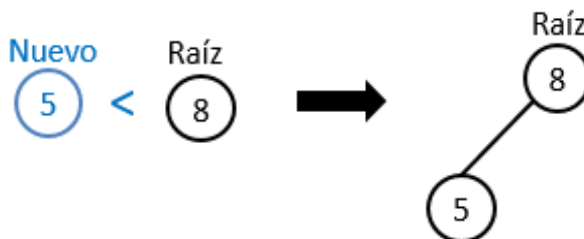
Inserción

La inserción se hace de elemento por elemento, cada nodo va buscando su lugar cuidando que se cumpla la condición de que las claves menores van a la izquierda y las mayores a la derecha.

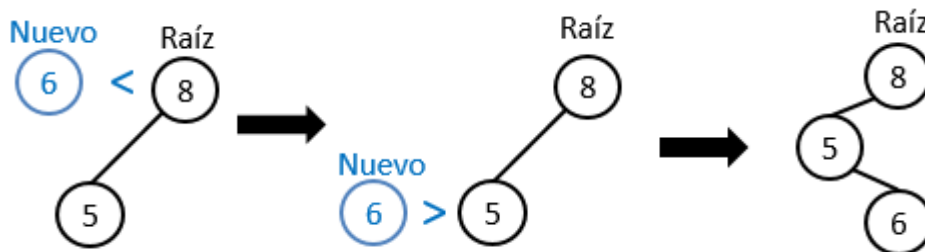
Ejemplo:
Insertar 8



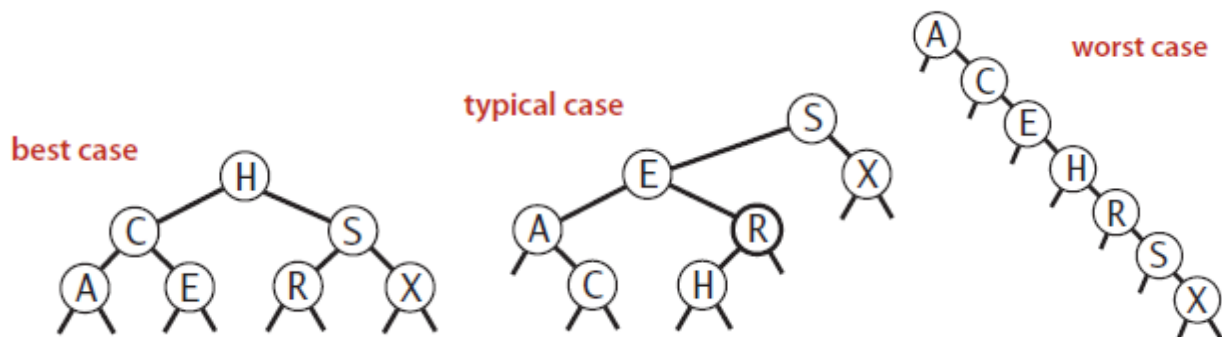
Insertar 5



Insertar 6



Su complejidad es en el caso promedio y el mejor caso: $O(\log n)$, pero en el peor caso se vuelve $O(n)$.



Implementación recomendada:

Árbol

Nodo
int clave

Nodo raíz

insertar(int valor)
imprimir()

Método insertar

Forma 1: Ciclo mientras

Clase Árbol – Método insertar

Parámetro: valor

El nodo n se crea con el valor recibido

Si la raíz == nulo

Raíz = n

Si no

Nodo actual = raíz;

Repetir hasta insertar el nodo

Si(n.clave > actual.clave) //Derecha

Si(actual.nodoDerecho ==null)

actual.nodoDerecho = n;

Si no

actual = actual.nodoDerecho;

Sino //Izquierda

Si(actual.nodoIzquierdo ==null)

actual.nodoIzquierdo = n;

Si no

actual = actual.nodoIzquierdo;

Método imprimir

Clase Arbol – Método imprimir

raiz.imprimir();

Nodo izq

Nodo der

imprimir()

Forma 2: Recursividad

Clase Árbol – Método insertar

Parámetro: valor

El nodo n se crea con el valor recibido

raíz.insertar(n)

Clase Nodo – Método insertar

Parámetro: nodo n

Si(n.clave > clave) //Derecha

Si(nodoDerecho ==null)

nodoDerecho = n;

Si no

nodoDerecho.insertar(n)

Sino //Izquierda

Si(nodolIzquierdo ==null)

nodolIzquierdo = n;

Si no

nodolIzquierdo.insertar(n)

Clase Nodo – Método imprimir

Si izq!=nulo entonces izq.imprimir()

Imprimir clave

Si der!=nulo entonces der.imprimir()

ACTIVIDAD: Árboles binario de búsquedas: inserción

1. Dibuje el resultado de crear árboles binarios de búsqueda con las siguientes secuencias de números:

a) 10, 45, 83, 93, 23, 12, 72, 7, 36, 58, 62

b) 98, 83, 76, 65, 52, 43, 32, 20, 11, 9

c) 21, 94, 45, 68, 32

2. Realice un programa que implemente un árbol binario de búsqueda, debe tener la funcionalidad de inserción e impresión.

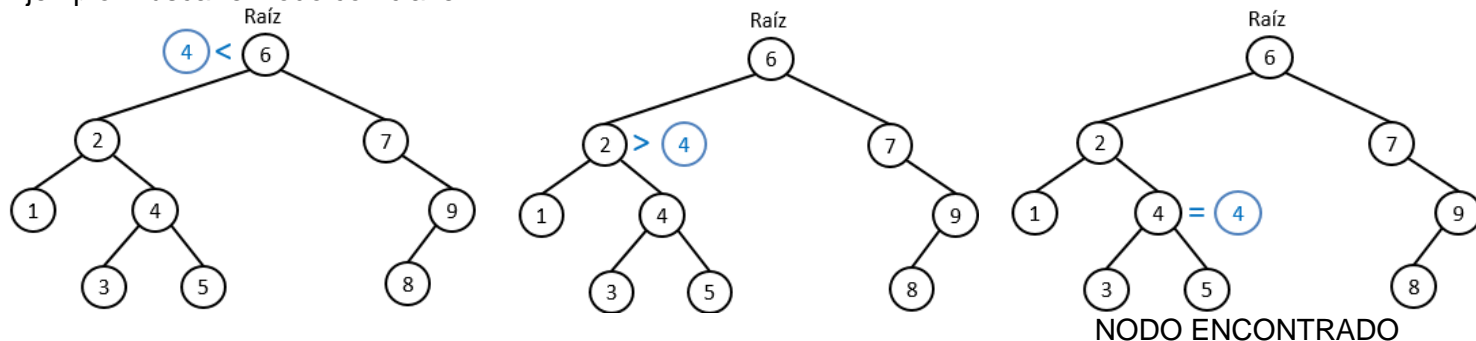
Búsqueda

Consiste en buscar el nodo con una clave específica, puede realizarse por dos motivos:

- Obtener el valor del nodo, en caso de que se almacene información extra a la clave. Por ejemplo, si es la información de una base de datos de alumnos la clave podría ser el ID y el valor podría estar conformado por: nombre, año de ingreso, promedio de calificaciones, etc.
- Verificar si una clave se encuentra o no en el árbol

La búsqueda se realiza recorriendo el árbol de acuerdo a la clave buscada. Primero se verifica si la clave es igual a la clave del nodo raíz, si es así se encontró el nodo, sino se analiza si es menor o mayor para ir a buscar en el nodo a la izquierda o a la derecha. Después se verifica si la clave es igual a la clave del nodo actual, si es así ya se encontró, sino nuevamente se tiene que mover a la izquierda o la derecha, y así sucesivamente. En caso de que se siga el camino correcto, no se encuentre el nodo y ya no existan más nodos por recorrer se indica que no se encontró la clave buscada.

Ejemplo: Buscar el nodo con clave 4



Su complejidad, al igual que la inserción, es en el caso promedio y el mejor caso: $O(\log n)$, pero en el peor caso se vuelve $O(n)$.

ACTIVIDAD: Árboles binario de búsquedas: búsqueda

1. Al programa de la actividad anterior agregue la funcionalidad de búsqueda.

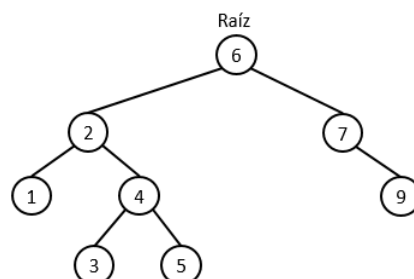
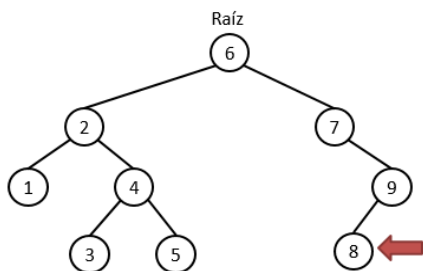
Eliminación

La eliminación en un nodo se hace de tres formas distintas, cada una para un caso específico:

Caso 1: El nodo a eliminar no tiene nodos hijos.

Se busca el nodo a partir de la clave, una vez que se encuentra y si no tiene hijos: se elimina el nodo.

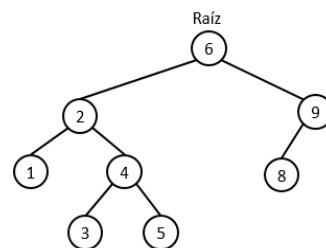
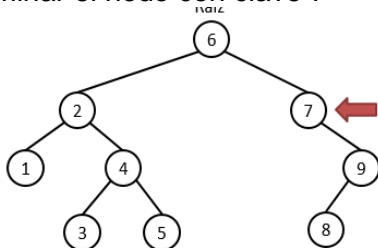
Ejemplo: Eliminar el nodo con clave 8



Caso 2: El nodo a eliminar tiene sólo un hijo.

Se busca el nodo a partir de la clave, una vez que se encuentra y si tiene sólo un hijo: se elimina el nodo y se reemplaza por su nodo hijo.

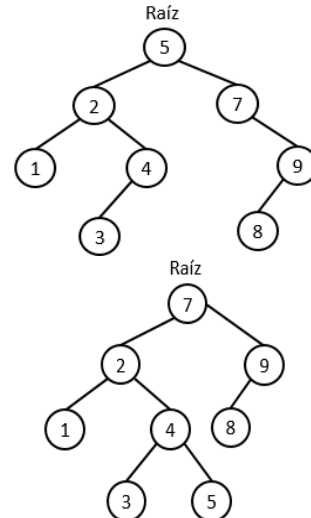
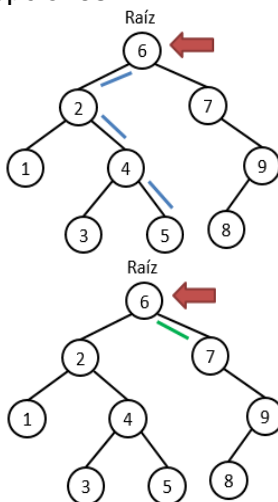
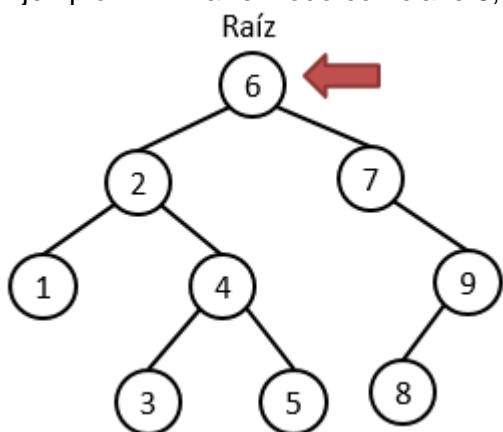
Ejemplo: Eliminar el nodo con clave 7



Caso 3: El nodo a eliminar tiene dos hijos.

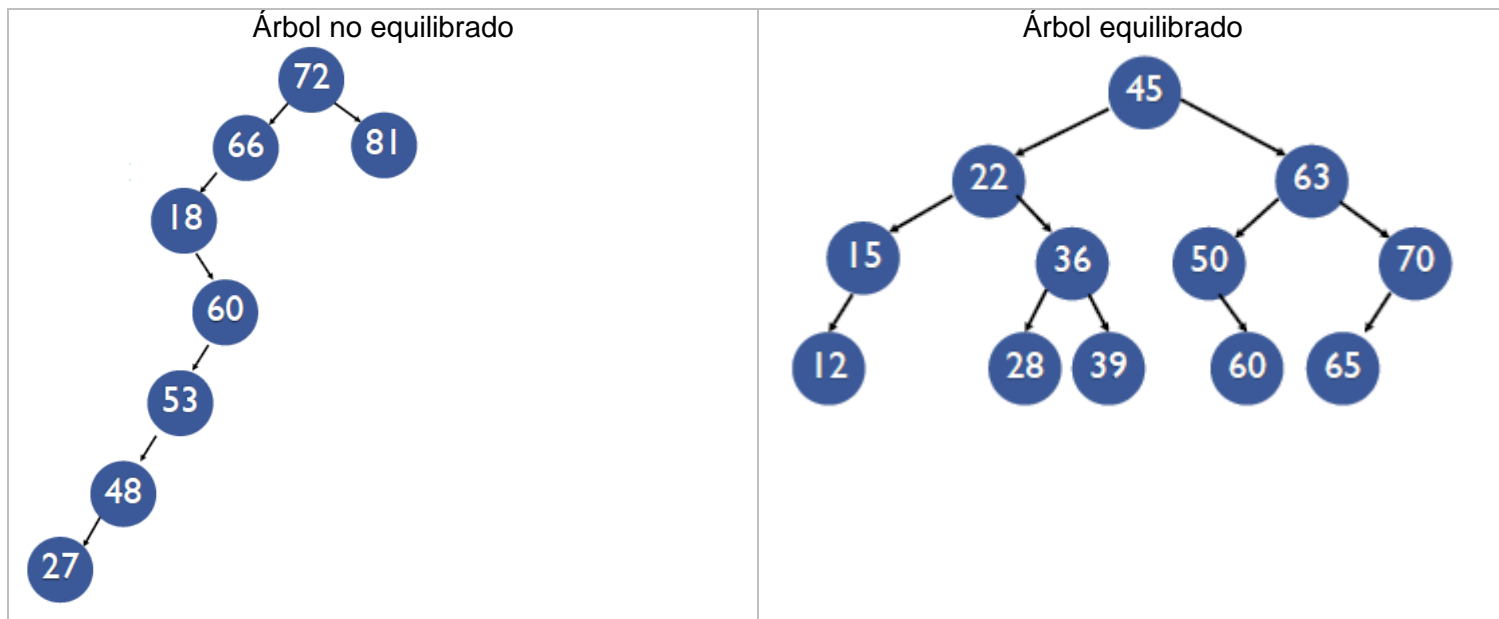
Se busca el nodo a partir de la clave, una vez que se encuentra y si tiene dos hijos: se busca el nodo más a la derecha del lado izquierdo, o el nodo más a la izquierda del lado derecha para que reemplace al nodo a eliminar.

Ejemplo: Eliminar el nodo con clave 6, dos opciones



11.3.3 Árboles AVL (Adelson-Velsky-Landis)

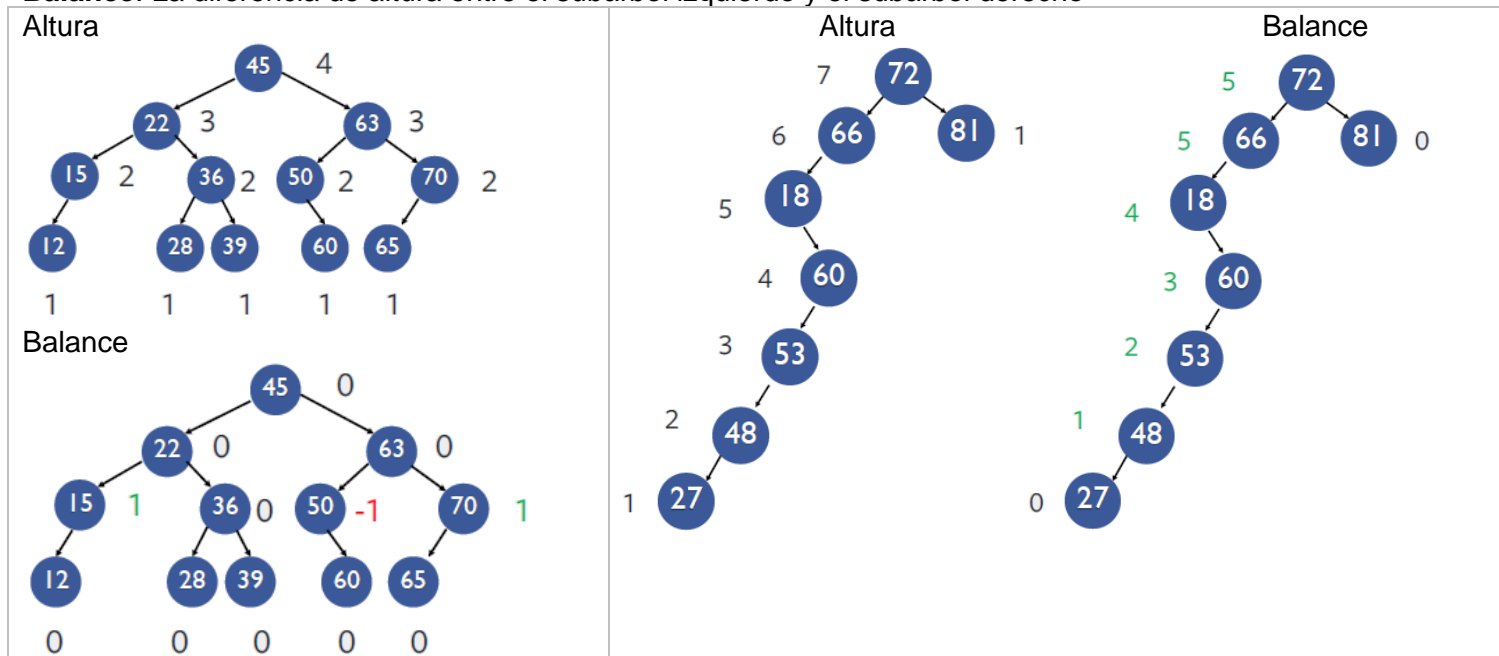
Los árboles AVL son árboles binarios de búsqueda equilibrados, esto con el objetivo de hacer más eficientes las operaciones de inserción, consulta y eliminación.



Para generar árboles equilibrados, los árboles AVL realizan una serie de ajustes por medio de “rotaciones” verificando que el árbol este equilibrado después de cada inserción o eliminación. Para esto es importante incluir algunos conceptos.

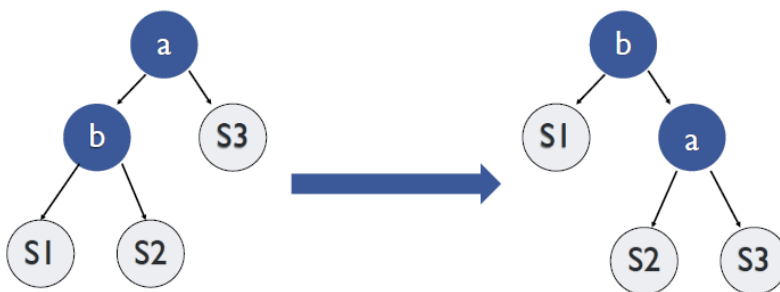
Altura: Es el nivel en el que se encuentra un nodo de abajo hacia arriba.

Balance: La diferencia de altura entre el subárbol izquierdo y el subárbol derecho

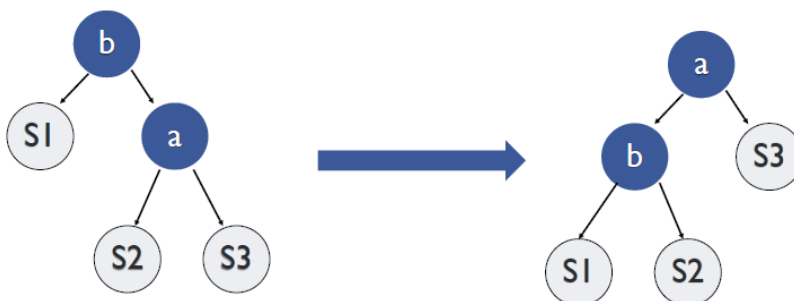


En un árbol AVL sólo se permiten balances iguales a 0, 1 y -1. En caso de que se tenga un balance distinto en algún nodo se debe realizar alguna de las siguientes rotaciones:

Rotación derecha

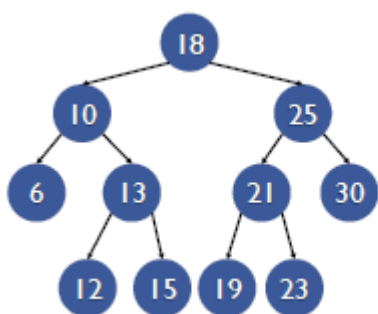


Rotación izquierda

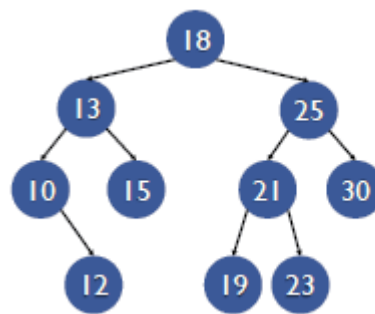


Root Balance	Left Child Balance	Right Child Balance	Fix
-2	Balanced	≤ 0 (0 or -1)	Rotate Root Left
-2	Balanced	1	Rotate Right Child Right; Rotate Root Left
2	-1	Balanced	Rotate Left Child Left; Rotate Root Right
2	≥ 0 (0 or 1)	Balanced	Rotate Root Right

Ejemplos:



Delete 6



Delete 15

11.4 Set

Sets are containers that store **unique** elements following a specific order.

In a set, the value of an element also identifies it (the value is itself the key, of type T), and each value must be unique. The value of the elements in a set cannot be modified once in the container (the elements are always const), but they can be inserted or removed from the container.

Sets are typically implemented as binary search trees.

Mainly the following three basic operations are performed in the set:

- **void insert(key):** Adds an element.
- **void remove(key):** Removes an element.
- **bool contains(key):** Returns true if the structure contains the key, otherwise false.
- **int size():** Returns the number of elements.
- **isEmpty:** Returns true if queue is empty, else false.

11.5 Map

Maps are associative containers that store elements formed by a combination of a **key value** and a **mapped value**, following a specific order.

In a map, the key values are generally used to sort and uniquely identify the elements, while the mapped values store the content associated to this key. The types of key and mapped value may differ, and are grouped together in member type `value_type`, which is a pair type combining both.

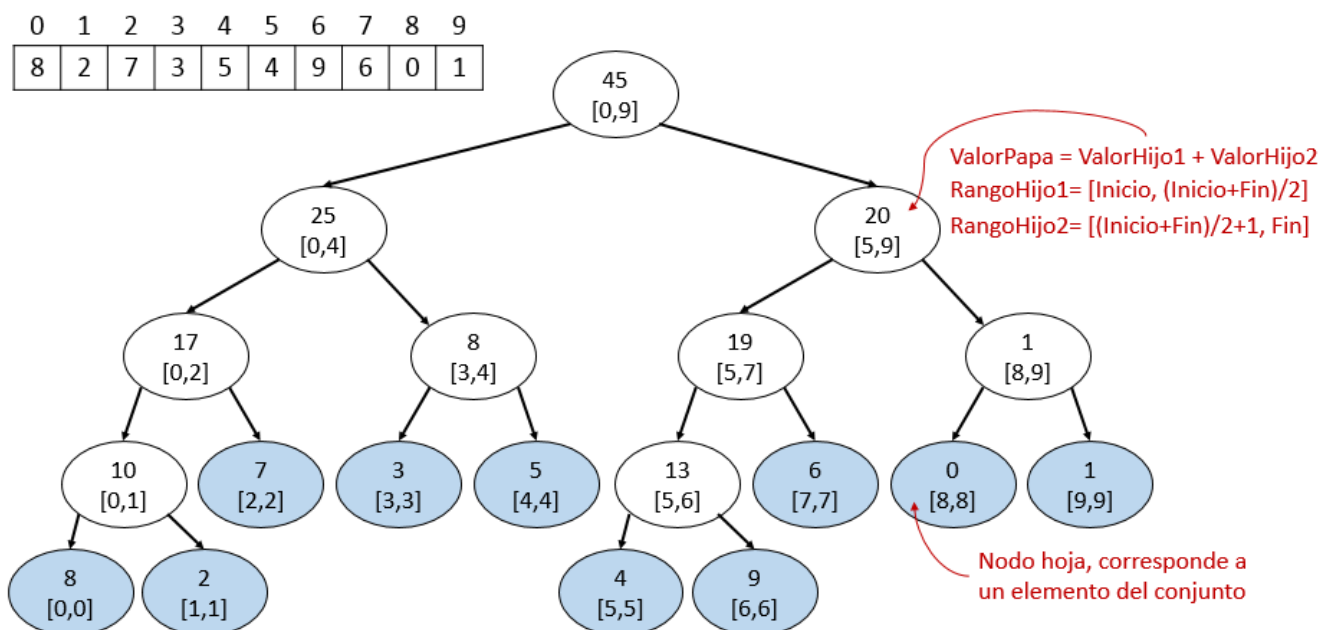
Maps are typically implemented as binary search trees.

Mainly the following three basic operations are performed in the set:

- **void insert(key,value):** Adds an element.
- **void modify(key,value):** Modifies an existent element.
- **value get(key):** Returns the value of a specific key.
- **void remove(key):** Removes an element.
- **bool contains(key):** Returns true if the structure contains the key, otherwise false.
- **int size():** Returns the number of elements.
- **isEmpty:** Returns true if queue is empty, else false.

9.4 Árbol de segmentos

Un árbol de segmentos permite almacenar la información de un conjunto de datos en forma de segmentos con el objetivo de realizar una consulta eficiente en un subconjunto contiguo. Fue propuesto por J.L. Bentley en 1977.



Aplicaciones de los árboles de segmentos

- Obtener de forma eficiente la suma de un rango de valores
- Obtener de forma eficiente el valor mínimo o máximo de un rango de valores

Algoritmo de construcción – complejidad $O(n \log n)$

Crear un nodo enviando como parámetro: el arreglo y el rango del arreglo (inicio, fin)

Si $\text{inicio} == \text{fin}$, el valor del nodo = $\text{arreglo}[\text{inicio}]$

Si $\text{inicio} \neq \text{fin}$, se crean dos hijos

Para crear el primer hijo se envía el arreglo y el rango $(\text{inicio}, (\text{inicio} + \text{fin})/2)$

Para crear el segundo hijo se envía el arreglo y el rango $((\text{inicio} + \text{fin})/2 + 1, \text{fin})$

Calcular los valores de los nodos de abajo hacia arriba

Si $\text{inicio} == \text{fin}$

valor del nodo = $\text{arreglo}[\text{inicio}]$

Si no

valor del nodo = valor hijo 1 + valor hijo 2

Algoritmo de consulta – complejidad $O(\log n)$

Consultar el valor en cada nodo, suponiendo que el rango de búsqueda es $(b\text{Inicio}, b\text{Fin})$

Si el rango de búsqueda no coincide con el rango del nodo ($\text{nodo.Fin} < b\text{Inicio}$ o $\text{nodo.Inicio} > b\text{Fin}$)

Se regresa un 0

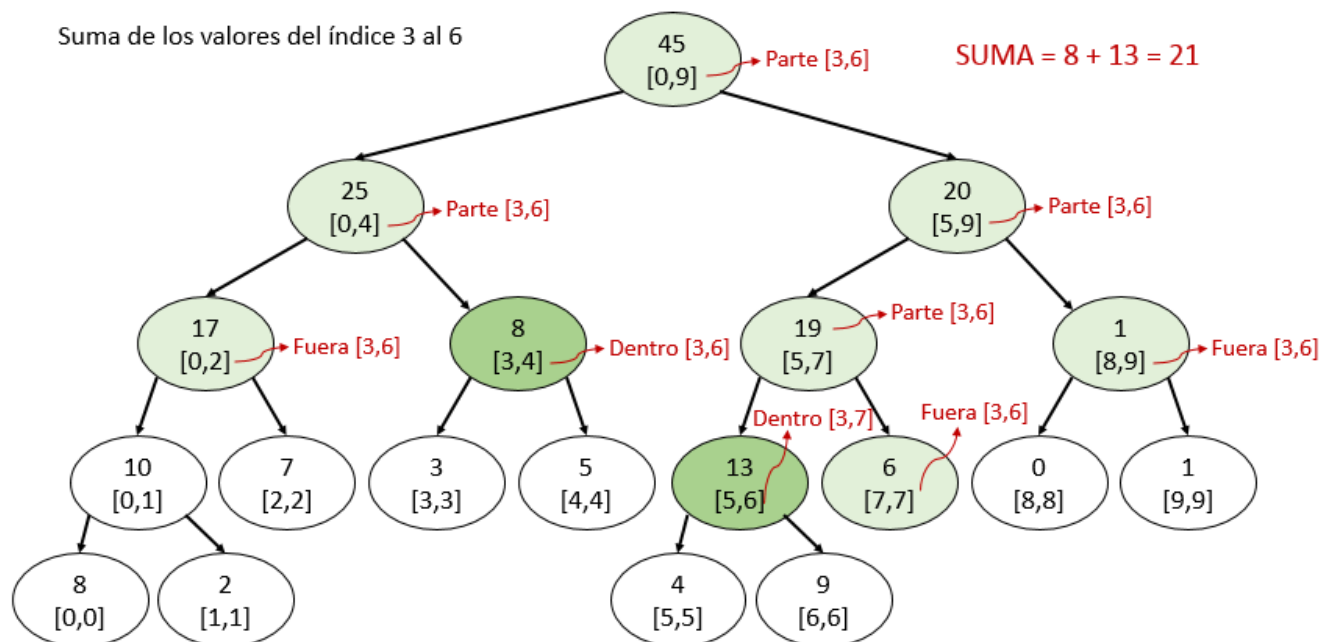
Si el rango del nodo está *dentro* del rango buscado ($\text{nodo.Inicio} \geq b\text{Inicio}$ && $\text{nodo.Fin} \leq b\text{Fin}$)

Se regresa el valor del nodo

Si no se presentan ninguna de las dos anteriores

Seguir buscando en los nodos hijo

Suma de los valores del índice 3 al 6

**Ventajas y desventajas**

- (+) Es un árbol equilibrado, por lo tanto la obtención del valor es eficiente
- (-) Si se agregan datos al conjunto hay que volver a crear el árbol

ACTIVIDAD:

1. Generar en papel los árboles de segmentos para los siguientes arreglos y calcular las sumas de los rangos establecidos

a) [0 1 2 3 4 5 6], sumas de los valores en los rangos:

- (3,5)
- (3,6)
- (0,3)

b) [24 12 96 09 81 32 67], sumas de los valores en los rangos:

- (4,6)
- (0,2)
- (5,6)

2. Generar en papel el árbol de segmentos que permita obtener de forma eficiente el valor mínimo en el siguiente conjunto de datos: [24 12 96 09 81 32 67]. Obtenga el valor mínimo en los rangos:

- a) (4,6)
- b) (0,2)
- c) (5,6)

3. Escribir un programa que a partir de un conjunto de datos sea capaz de crear un árbol de segmentos que permita obtener de forma eficiente: La suma o el valor mínimo de los elementos en un rango específico. El programa debe:

- a) Dar la opción de capturar el conjunto de datos o utilizar uno predefinido
- b) Construir el árbol de segmentos e imprimirlo
- c) Obtener la suma de un rango de índices
- d) Obtener el valor mínimo de un rango de índices

11. Compresión de datos

10.1 Código de Huffman

Es un algoritmo usado para compresión de datos desarrollado por David A. Huffman.

Calcula el código en binario para representar a cada carácter, el cual debe cumplir las siguientes características:

- La longitud del código varía según la probabilidad de aparición del carácter, es decir, los caracteres con mayor probabilidad de aparición tienen un código muy corto y los caracteres con menor probabilidad tienen un código más extenso.
- Los códigos son distinguibles entre sí

Ejemplo 1: Compresión del texto:

“AAABACABACAACBAABACA”

Información de los caracteres

Carácter	Frecuencia	Probabilidad	Código de Huffman
A	10	0.5	A -> 0
B	6	0.3	B -> 10
C	4	0.2	C -> 11

Ejemplo de compresión:

Original: AAABACABACAACBAABACA - 20 char (8 bits cada char) = 160bits

Huffman: 0001001101001100111000100110 - 28 bits + la tabla de conversión

Ejercicio: Calcule el código de Huffman o la cadena original:

1. Cadena: A,A,C,B,A,A,B
Código Huffman: ¿?
2. Código Huffman: 00010111000
Cadena: ¿?

Algoritmo para crear el código de Huffman

Recibe como parámetro: Una cadena de caracteres de muestra (para calcular las probabilidades) o una lista de caracteres con su probabilidad de aparición.

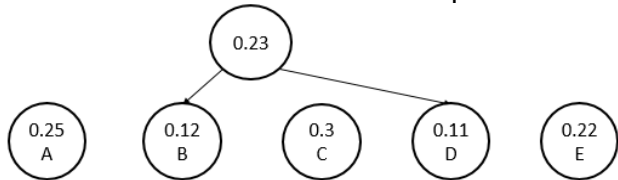
1. En caso de que se haya recibido la cadena de caracteres de muestra
 - a) Calcular la probabilidad de aparición de cada carácter
2. Crear un árbol binario de la siguiente forma:
 - a) Crear un nodo hoja por cada carácter, guardando en el nodo: el carácter y su probabilidad.
 - b) Mientras todos los nodos no formen un solo árbol
 - i. Obtener los dos nodos con menor peso que no tengan un padre
 - ii. Crear un nuevo nodo, cuyos hijos sean los dos nodos de menor peso, y asignar como probabilidad la suma de las probabilidades de sus hijos
3. Recorrer el árbol binario de arriba hacia abajo para calcular el código de cada carácter
 - a) Comenzar el código en la raíz y difundirlo hasta las hojas (comienza vacío)
 - b) Cada nodo envía su código a sus hijos más un 0 al hijo izquierdo y un 1 al hijo derecho.
4. Finalmente se regresan los códigos de los nodos hoja

Ejemplo:

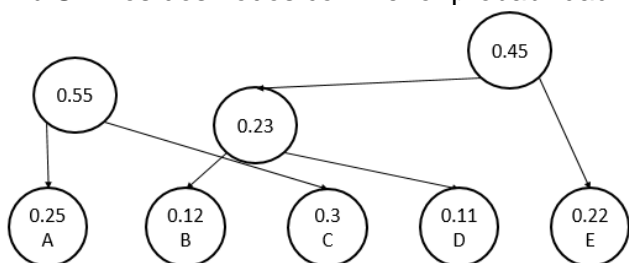
1. Probabilidades de aparición de cada caracter

Carácter	A	B	C	D	E
Probabilidad	0.25	0.12	0.3	0.11	0.22

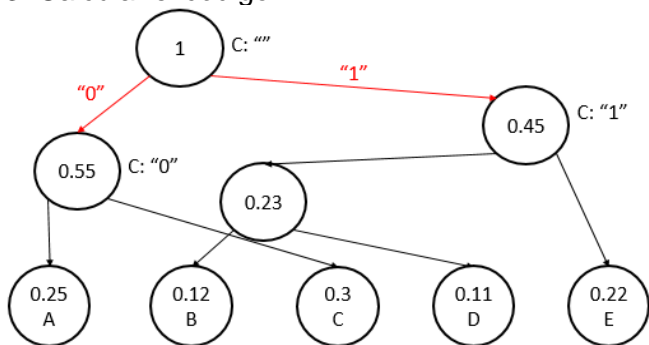
2.b Unir los dos nodos con menor probabilidad



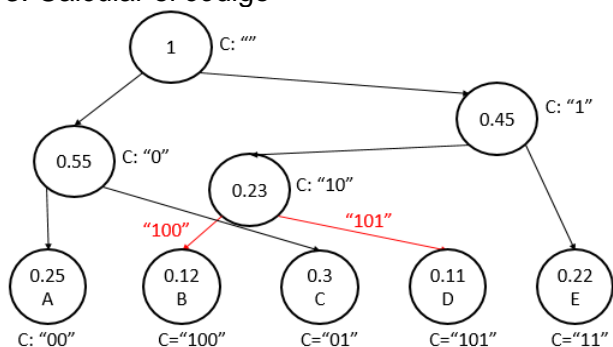
2.b Unir los dos nodos con menor probabilidad



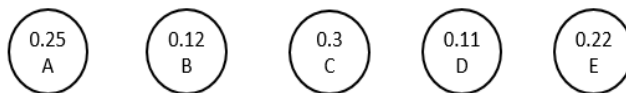
3. Calcular el código



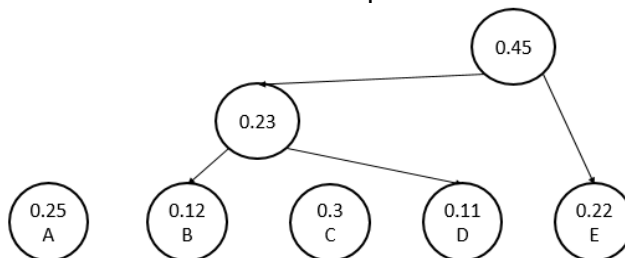
3. Calcular el código



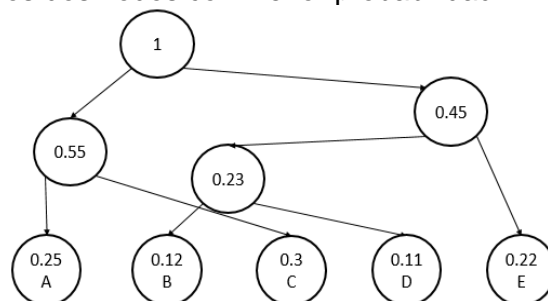
2.a Crear los nodos hoja



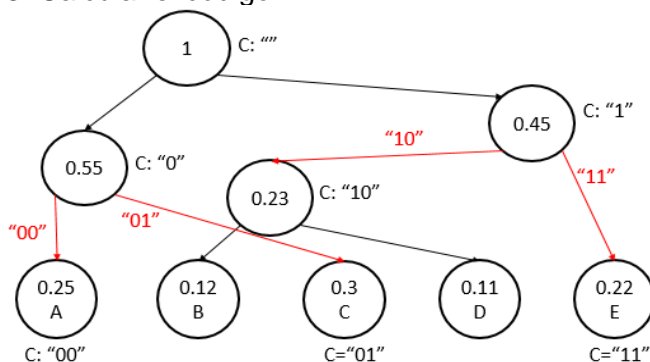
2.b Unir los dos nodos con menor probabilidad



2.b Unir los dos nodos con menor probabilidad



3. Calcular el código



Códigos Huffman:

Carácter	A	B	C	D	E
Probabilidad	0.25	0.12	0.3	0.11	0.22
Código Huffman	00	100	01	101	11

10.2 Codificación aritmética

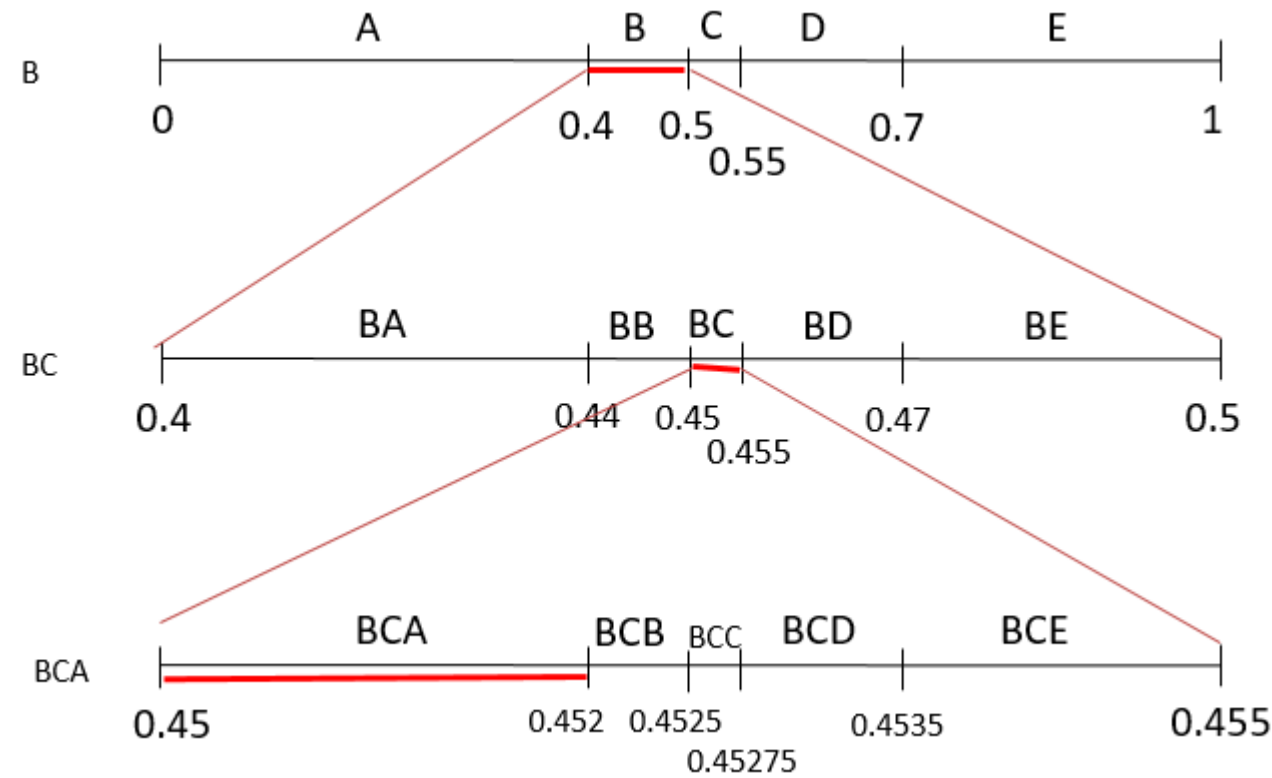
Es un algoritmo usado para compresión de datos donde un grupo de símbolos se codifica utilizando un número real entre 0 y 1. A diferencia de otros algoritmos de compresión, como el código Huffman, comprime la información por grupos de valores sin crear un código por cada símbolo.

Lo primero que se debe hacer es calcular o pedir la probabilidad de aparición de cada carácter para con esto crear un rango entre 0 y 1 para cada carácter, como se muestra en la siguiente tabla:

Símbolo	Probabilidad o Amplitud	Rango	Límite izquierdo	Límite derecho
A	0.40	0.00 – 0.40	0.00	0.40
B	0.10	0.40 – 0.50	0.40	0.50
C	0.05	0.50 – 0.55	0.50	0.55
D	0.15	0.55 – 0.70	0.55	0.70
E	0.30	0.70 – 1.00	0.70	1.00

Luego se va dividiendo de forma recursiva un segmento de valores reales entre 0 y 1 según los rangos de los caracteres, las divisiones se hacen de acuerdo a los símbolos que se codifican.

Ejemplo: Si se quiere codificar el símbolo BCA



BCA puede ser sustituido por cualquier número entre 0.45 y 0.452

Codificación con fórmulas

Ejemplo: Codificar CBA con fórmulas

$$izq = 0, der = 1, amp = 1$$

- C
 - $izq = izq + (izq_C * amp) \rightarrow izq = 0 + (0.5 * 1) = 0.5$
 - $der = izq + (der_C * amp) \rightarrow der = 0 + (0.55 * 1) = 0.55$
 - $amp = der - izq \rightarrow amp = .55 - .5 = 0.05$
- B
 - $izq = izq + (izq_B * amp) \rightarrow izq = 0.5 + (0.4 * 0.05) = 0.52$
 - $der = izq + (der_B * amp) \rightarrow der = 0.5 + (0.5 * 0.05) = 0.525$
 - $amp = der - izq \rightarrow amp = .525 - .52 = 0.005$
- A
 - $izq = izq + (izq_A * amp) \rightarrow izq = 0.52 + (0.00 * 0.005) = 0.52$
 - $der = izq + (der_A * amp) \rightarrow der = 0.52 + (0.40 * 0.005) = 0.522$
 - $amp = der - izq \rightarrow amp = .522 - .52 = 0.002$

Entonces cualquier número entre 0.52 y 0.522 representará a CBA

Decodificación con fórmulas

Para decodificar, se debe tener el número real y el número de niveles codificados

Por ejemplo: Decodificar 0.521 a tres niveles

- Nivel 1
 - 0.521 pertenece al rango de la C $\rightarrow C$
 - $num = \frac{num - izq_C}{amp_C} \rightarrow num = \frac{0.521 - 0.5}{0.05} = 0.42$
- Nivel 2
 - 0.42 pertenece al rango de la B $\rightarrow B$
 - $num = \frac{num - izq_B}{amp_B} \rightarrow num = \frac{0.42 - 0.4}{0.1} = 0.2$
- Nivel 3
 - 0.2 pertenece al rango de la A $\rightarrow A$

Decodificación de 0.521 es CBA

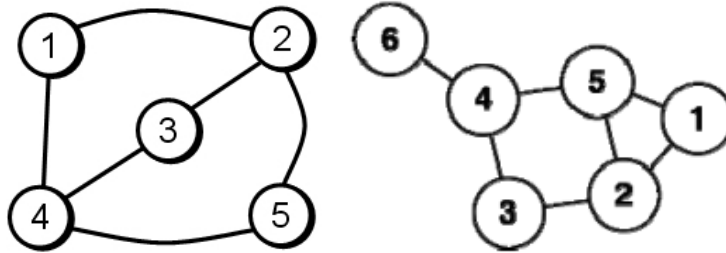
ACTIVIDAD: Codificación con código Huffman y código aritmético

En base a la cadena: "123451231112345123412"

- e) Codifique con Huffman la cadena "2134"
- f) Codifique con código aritmético la cadena "2134"
- g) Decodifique con Huffman: 10001101
- h) Decodifique con código aritmético: 0.5748 a 4 niveles

12. Gráficas (Grafos)

Una gráfica es una estructura de datos que representa a un conjunto de objetos llamados **vértices o nodos** unidos por enlaces llamados **aristas o arcos**, que permiten representar las relaciones entre los elementos de un conjunto. Permiten estudiar las interrelaciones entre unidades que interactúan unas con otras.



Los grafos pueden utilizarse para representar:

- Redes de Computadoras
Vértices: Equipos (computadoras, routers, etc.)
Aristas: Conexiones (cables o conexiones inalámbricas)
- Mapas
Vértices: Ciudades
Aristas: Carreteras
- Relaciones Humanas
Vértices: Personas
Aristas: Relaciones entre personas (padre, esposa, jefe, etc.)
- Redes Bayesianas
Técnica de I.A. que permite representar “conocimiento”
Vértices: Variables aleatorias
Aristas: Relaciones entre variables

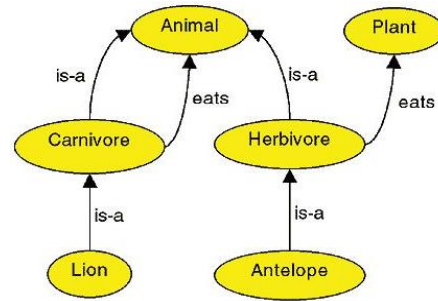


- Ontologías

Técnica de I.A. que permite representar “conocimiento”

Vértice: Objeto o Descripción

Arista: Relación



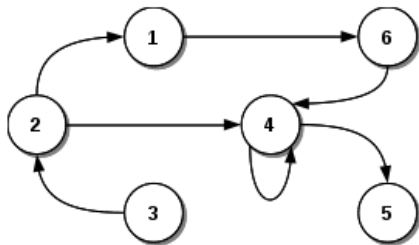
Como se puede ver, su estudio es muy importante para la generación de algoritmos que puedan solucionar problemas específicos en donde la información se represente mediante gráficas.

Gráficas dirigidas y no dirigidas

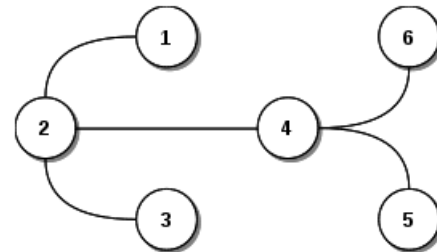
Gráficas dirigidas: Son aquellas en donde es necesario representar la dirección de la unión de los dos nodos. Algunos ejemplos son: Ontologías, Relaciones Humanas y mapa de las calles en una ciudad.

Gráficas no dirigidas: Se da cuando sólo es necesario representar la unión entre dos nodos. Algunos ejemplos son: Mapas de carreteras (si es que siempre una carretera va en dos sentidos) y Redes de Computadoras.

Gráfica dirigida



Gráfica no dirigida

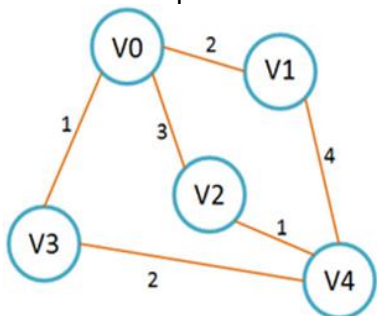


Gráficas ponderadas y gráficas no ponderadas

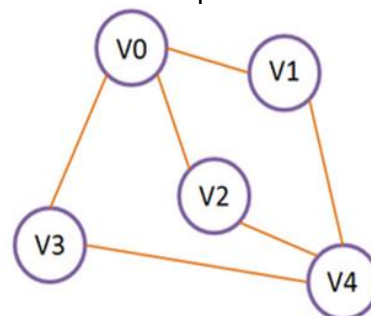
Gráfica ponderada: Son aquellas donde las aristas tienen un valor, que podría ser la distancia, importancia, tiempo, velocidad, etc. Algunos ejemplos son: Mapas de carreteras (distancias) y Redes de Computadora (velocidad de conexión).

Gráficas no ponderadas: Son aquellas donde las aristas no tienen un valor, simplemente se representa la relación entre dos nodos. Algunos ejemplos son: Redes Bayesianas.

Gráfica ponderada



Gráfica no ponderada

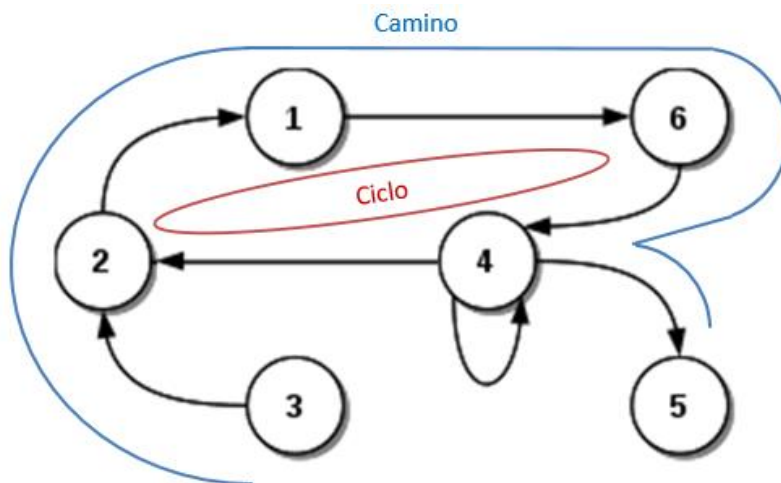


Algunos conceptos en grafos son:

Grado de un nodo: Es el número de aristas conectadas a un nodo. En el caso de las gráficas dirigidas se pueden distinguir dos tipos de grado, grado de salida: número de aristas que salen de un nodo y grado de entrada: número de aristas que entran a un nodo.

Camino: Es una secuencia de aristas consecutivas para llegar de un nodo a otro. La longitud del camino corresponde al número de aristas que contiene.

Ciclo o circuito: es un camino que empieza y termina en el mismo nodo.



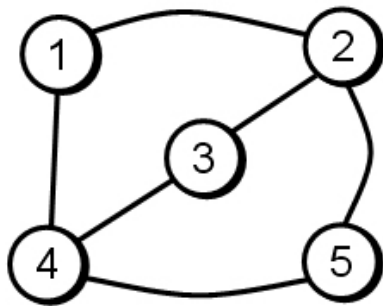
11.1 Representaciones de las gráficas

Existen dos representaciones clásicas para almacenar las conexiones entre nodos: Matriz de Adyacencias y Listas de Adyacencias.

Matriz de adyacencias

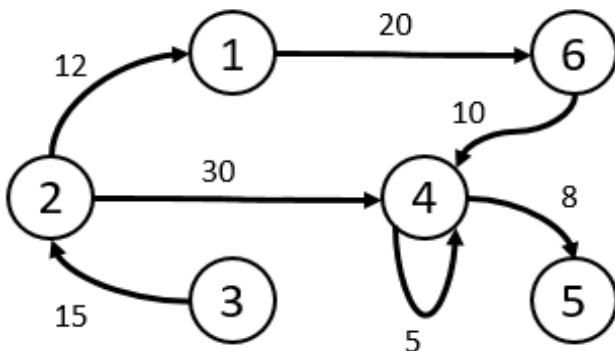
Es una matriz de tamaño $n \times n$, donde n es el número de nodos. En esta matriz se asocia a cada fila y a cada columna a un nodo del grafo, siendo el valor en cada celda la relación que existe entre el nodo de la fila y el nodo de la columna.

Si es una gráfica no ponderada y no dirigida: Cada celda con un 1 indica que hay conexión entre el nodo de la fila y el nodo de la columna; y con un 0 si no hay conexión entre ellos. En este caso la matriz es simétrica.



M	1	2	3	4	5
1	0	1	0	1	0
2	1	0	1	0	1
3	0	1	0	1	0
4	1	0	1	0	1
5	0	1	0	1	0

Si es una gráfica ponderada y dirigida: Las filas representan al nodo de donde sale la arista y las columnas representan el nodo a donde llega la arista, y el valor en la celda representa la ponderación de la arista. En este caso la matriz NO es simétrica.



Salida\Entrada	1	2	3	4	5	6
1	0	0	0	0	0	20
2	12	0	0	30	0	0
3	0	15	0	0	0	0
4	0	0	0	5	8	0
5	0	0	0	0	0	0
6	0	0	0	10	0	0

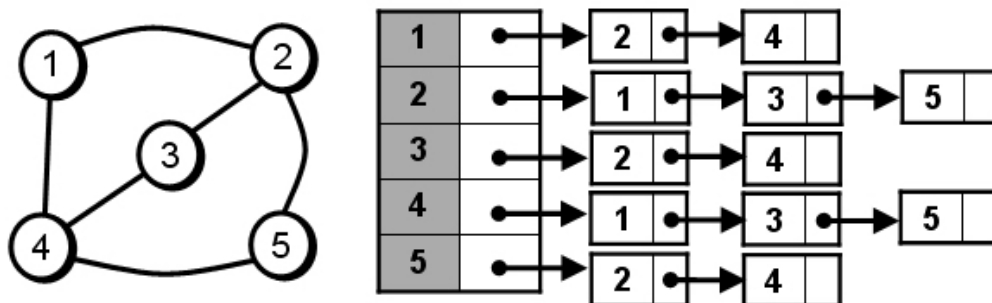
(+) Es sencillo de implementar

(-) Si la cantidad de nodos es grande se requiere mucha memoria para el almacenamiento de las conexiones

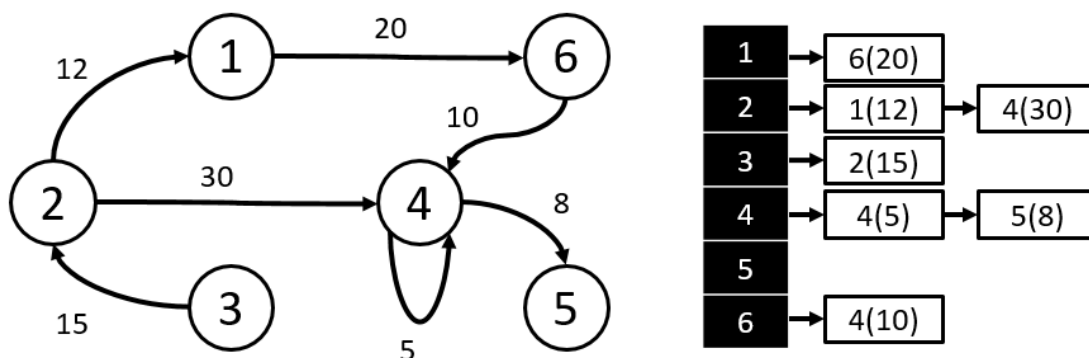
Listas de Adyacencias

Se representa cada nodo como un objeto que contiene una lista con todos aquellos nodos adyacentes a él.

Si es una gráfica no ponderada y no dirigida: Cada nodo contiene una lista con las ligas a los nodos que están conectados a él.



Si es una gráfica ponderada y dirigida: Cada nodo contiene una liga a todos los nodos que salen de él, además de incluir el nodo al que se conectan incluye la ponderación de la arista o de la conexión.



(+) Optimiza la memoria utilizada para almacenar la información de las conexiones

(-) Es más complejo de implementar que la matriz de adyacencia

ACTIVIDAD: Representar en una Gráfica el Mapa de Aguascalientes

1. Utilizar la representación de Matriz de Adyacencias
2. Utilizar la representación de Lista de Adyacencias

El programa o los programas deben:

- Tener almacenada la información de los nombres de las ciudades, conexiones y distancias entre ellas.
- Imprimir la información del mapa utilizando texto, por ejemplo:

Ciudad de Aguascalientes, conectada con:

- Calvillo, 44 km
- Palo Alto, 40 km
- Jesús María, 11km
- San Francisco de los Romo 26 km

Calvillo, conectado con:

- Aguascalientes, 44 km



Implementación recomendada:

Matriz de adyacencias

Mapa

ArrayList<String> cds

int mAd[][]

agregarCd(String nombre)

agregarConexion(String cd1,String cd2,int dist)

imprimir()

Listas de adyacencias

Mapa

ArrayList<Ciudad> cds

agregarCd(String nombre)

agregarConexion(String cd1,String cd2,int dist)

imprimir()

Ciudad

String nombre;

ArrayList<Ciudad> cdAd

ArrayList<Integer> cdAdDist

agregarCdAd(Ciudad cd, int dist)

imprimir()

Ejemplo de uso de ambos programas:

```
Mapa ags = new Mapa();
```

```
ags.agregarCd("Aguascalientes");
```

```
ags.agregarCd("Calvillo");
```

```
ags.agregarConexion("Aguascalientes","Calvillo",44);
```

```
ags.imprimir();
```

11.2 Búsqueda en anchura y en profundidad

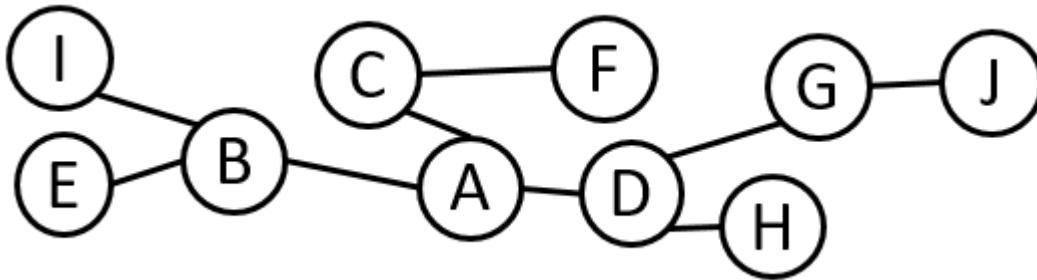
Recorrido o búsqueda en anchura o BFS (Breadth First Search)

El recorrido o búsqueda en anchura en un grafo debe partir de un nodo en específico y luego visitar primero los nodos más cercanos a ese nodo, luego los nodos cercanos a los que acaba de visitar y así sucesivamente.

Recorrido o búsqueda en profundidad o DFS (Depth First Search)

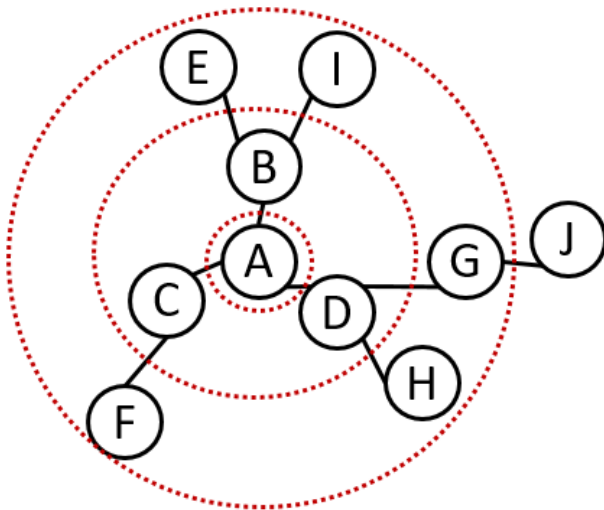
El recorrido o búsqueda en profundidad en un grafo debe partir de un nodo en específico y luego visitar todos los nodos de una rama en específico, y hasta terminar dicha rama visita los nodos de otra rama, y así sucesivamente.

Ejemplo: Recorrido
Suponga que se tiene el grafo:



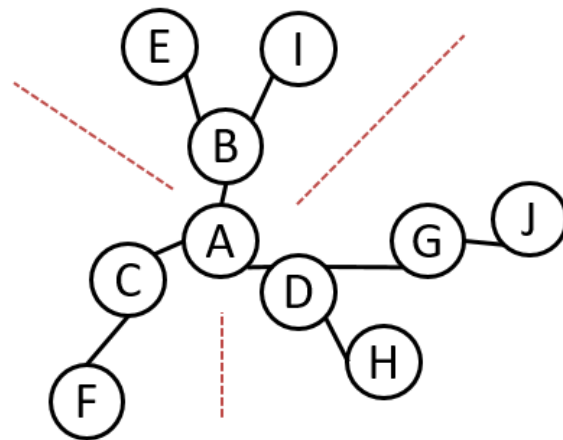
Los recorridos en anchura y profundidad comenzando desde el nodo A son:

En anchura:



A,CBD,FEIGH,J

En profundidad:



A,DHGGJ,BIE,CF

Algoritmo de recorrido en anchura

Crear Queue de Nodos

Visitar el primer nodo

Agregar el primer nodo al Queue

Mientras el Queue no este vacío

Nodo auxiliar = el primer nodo del Queue

Para todos los nodos que están conectados con el nodo auxiliar y que no han sido visitados

Visitar el nodo

Marcar el nodo como visitado

Agregar el nodo al Queue

Fin para

Fin mientras

Algoritmo de recorrido en profundidad

Crear una Stack de Nodos

Visitar el primer nodo

Agregar el primer nodo al Stack

Mientras el Stack no este vacío

Nodo auxiliar = el primer nodo del Stack

Para todos los nodos que están conectados con el nodo auxiliar y que no han sido visitados

Visitar el nodo

Marcar el nodo como visitado

Agregar el nodo al Stack

Fin para

Fin mientras

¿Qué hacemos para un algoritmo de búsqueda? ¿Qué cambios hay que hacer?

EJERCICIOS

1. En base al Mapa de Aguascalientes, genere un programa que imprima el recorrido de las ciudades en el mapa comenzando con Arad, utilizando el Recorrido en Profundidad.



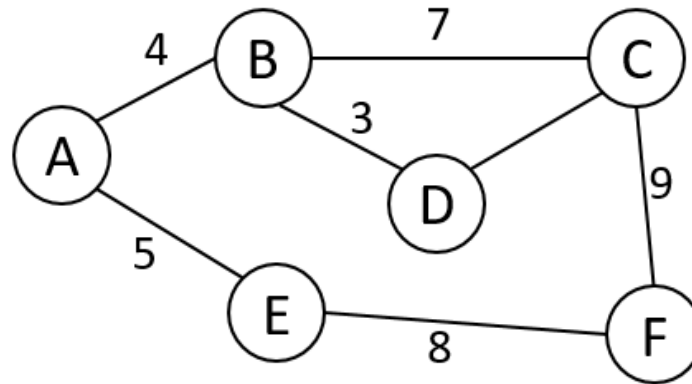
2. Crear un programa que al dar clic en una figura la coloree completamente.



11.3 Búsqueda Primero el Mejor (Best First Search)

La diferencia entre búsqueda primero el mejor y BFS o DFS es que primero visita el nodo aparentemente más prometedor. Es decir, debe existir un criterio que nos permita definir que nodo es el más prometedor hasta ahora.

Ejemplo: Encontrar un camino del nodo A al nodo F. Criterio: expandir el camino más corto



Paso 1:
 Visitados A
 PriorityQueue A, "A", 0

Paso 2:
 Visitados A
 B, "AB", 4
 E, "AE", 5
 E

Paso 3:
 Visitados A
 E, "AE", 5
 B, "C, "ABC", 11
 E, "D, "ABD", 7
 C
 D

Paso 4:
 Visitados A
 C, "ABC", 11
 B, "D, "ABD", 7
 E, "AEF", 13
 C
 D
 F

Ya se encontró la ruta: AEF con distancia 13

Algoritmo de Búsqueda Primero el Mejor

Crear un PriorityQueue de Nodos

Visitar el primer nodo

Agregar el primer nodo al PriorityQueue

Mientras el PriorityQueue no este vacío

 Nodo auxiliar = el primer nodo del PriorityQueue

 Para todos los nodos que están conectados con el nodo auxiliar y que no han sido visitados

 Visitar el nodo (en caso que sea el nodo buscado se termina la búsqueda)

 Marcar el nodo como visitado

 Agregar el nodo al PriorityQueue

 Fin para

Fin mientras

11.4 Búsqueda A*

La búsqueda A* es una Búsqueda Primero el Mejor que tiene por objetivo resolver un problema utilizando, además de una búsqueda con prioridad, una heurística. La prioridad $F(n)$ asignada a cada solución n está compuesta por dos elementos:

$$F(n) = g(n) + h'(n)$$

$g(n)$ Representa el costo real desde el estado inicial hasta la solución actual

$h'(n)$ Es una heurística, o una predicción, que representa el costo del estado actual hasta la solución final

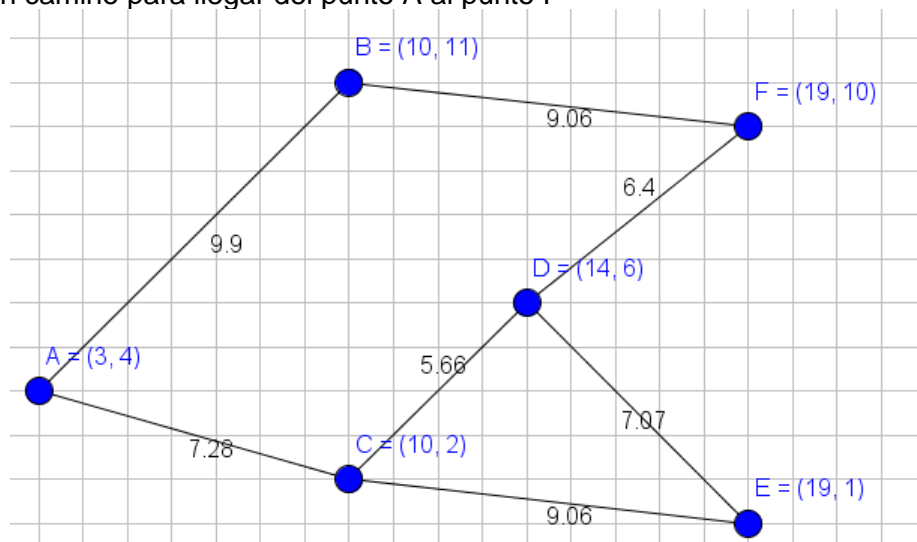
En grafos donde se quiere llegar de un nodo a otro, la heurística podría calcularse como la distancia del nodo actual n al nodo destino $D((x_1, y_1), (x_2, y_2)) = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$.

Es importante destacar que podemos implementar nuestro código de tal forma que un nodo pueda entrar a la PriorityQueue hasta que haya sido visitado, es decir, puede entrar varias veces por diferentes rutas.

Al realizar la búsqueda es necesario guardar la información del estado actual, es decir:

- Nodo o estado de la solución en el que nos encontramos
- Prioridad del nodo $F(n)$
- Guardar el cómo llegamos a la solución, por ejemplo: el camino
-

Ejemplo: Calcular un camino para llegar del punto A al punto F



Prioridad = Distancia recorrida + Distancia euclidiana desde el nodo actual hasta F

Paso 1:

PriorityQueue

A, "A" $P = 0 + \sqrt{(19 - 3)^2 + (10 - 4)^2} = 17.09$

Visitados: A

Paso 2:

PriorityQueue

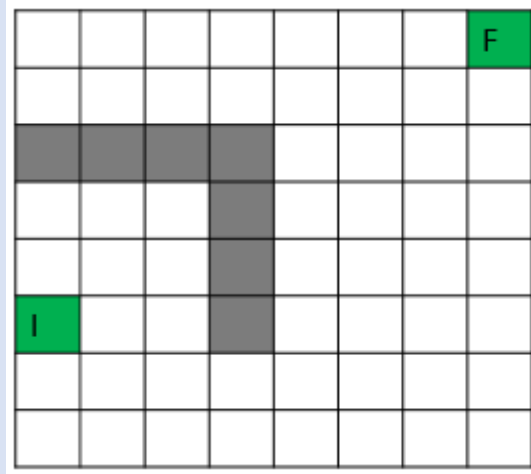
B, "AB" $P = 9.9 + \sqrt{(19 - 10)^2 + (10 - 11)^2} = 18.95$

C, "AC" $P = 7.28 + \sqrt{(19 - 10)^2 + (10 - 2)^2} = 19.32$

Visitados: A B C

EJERCICIOS

1. En base a una cuadrícula con obstáculos, calcular una camino para llegar de un punto a otro. Suponiendo que sólo se puede mover a la izquierda, derecha, arriba o abajo. Por ejemplo:

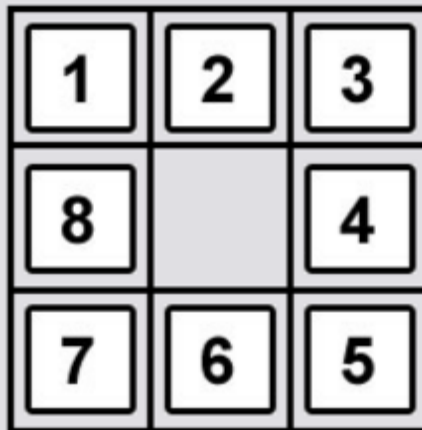


Resuelva el problema utilizando:

- Búsqueda primero el mejor, la prioridad se calcula en base a los movimientos realizados hasta ahora.
- Búsqueda A*, donde la prioridad se calcula como: $F(n) = g(n) + h(n)$
 $g(n)$: es el número de movimientos realizados hasta ahora
 $h(n)$: una heurística de la cantidad de movimientos necesarios para llegar al nodo destino, se puede calcular como la distancia Manhattan: $Distancia((x_1, y_1), (x_2, y_2)) = |x_1 - x_2| + |y_2 - y_1|$

2. Resuelva el problema 8-puzzle utilizando búsqueda A*, la prioridad se puede asignar como sigue:

- La suma de los movimientos realizados + el número de piezas colocadas en una casilla errónea
- El número de piezas colocadas en una casilla errónea + la sumatoria de la distancia manhattan de la posición de cada pieza y su posición correcta.



13. Algoritmos sobre gráficas

12.1 Algoritmo Unión – Buscar (Union Find)

El algoritmo Unión-Buscar tiene como objetivo determinar la conectividad en un grafo, o también para determinar si dos nodos en un grafo A y B están conectados, es decir, si existe un camino para llegar del nodo A al nodo B y viceversa.

En general, el algoritmo funciona de la siguiente manera: al inicio todos los nodos pertenecen a un subconjunto distinto, después de forma iterativa se van uniendo los subconjuntos según las aristas que conforman los nodos.

Las dos operaciones básicas son:

- Buscar: Determina a cual subconjunto pertenece un nodo.
- Unión: Une dos subconjuntos en uno sólo.

Algoritmo Unión-Buscar

Recibe como parámetro: Lista de nodos y lista de las conexiones del grafo

Crear un subconjunto para cada nodo

Repetir para cada conexión c en el grafo

nodo1 y nodo2 = los nodos de la conexión c

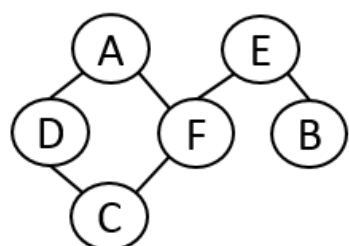
Buscar el subconjunto del nodo1, C1 = buscar(nodo1)

Buscar el subconjunto del nodo2, C2 = buscar(nodo2)

union(C1,C2);

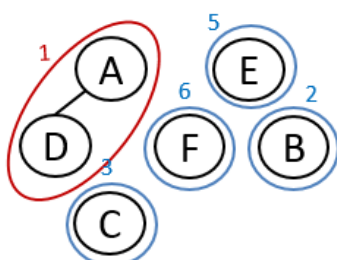
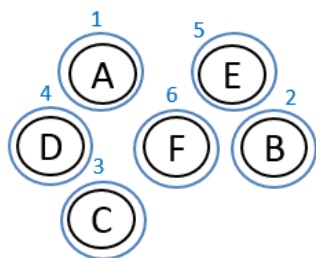
Fin repetir

Ejemplo:

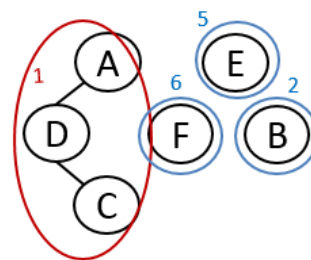


Aristas:

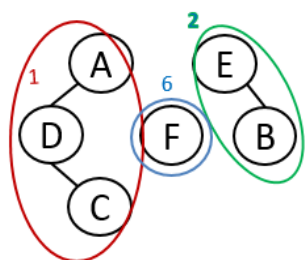
A – D
D – C
E – B
A – F
F – C
E – F



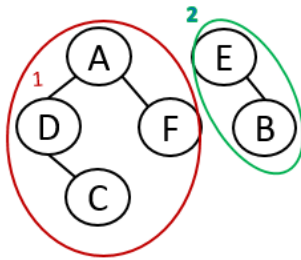
Unión A,D



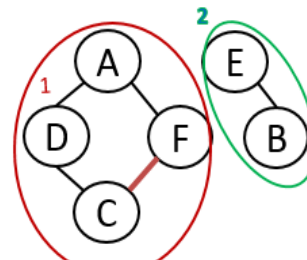
Unión D,C



Unión E,B



Unión A,F



Unión C,F

Las operaciones de este algoritmo se podrían ver como operaciones sobre conjuntos, por lo que una implementación eficiente es la siguiente:

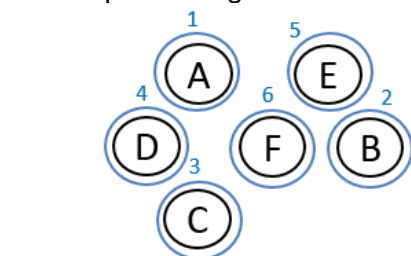
```
int Buscar(int v){
    if( P[v]==v )
        return v;
    else
        return Buscar(P[v]);
}
```

```
void Union(int a,int b){
    int ra = Buscar(a);
    int rb = Buscar(b);
    P[ra] = rb;
}
```

```
bool Pertenencia(int a, int b){
    if(Buscar(a)==Buscar(b))
        return true;
    else
        return false;
}
```

Tomado de Problemas y Algoritmos, Luis E. Vargas Azcona, OMI

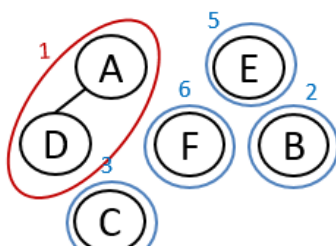
Una explicación gráfica de esta implementación es la siguiente:



Arreglo:

1	2	3	4	5	6
A	B	C	D	E	F
1	2	3	4	5	6

Representación arreglo:

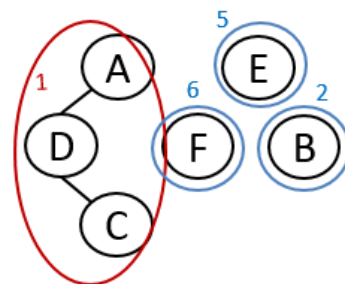
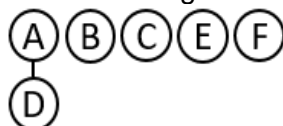


Unión A,D

Arreglo:

1	2	3	4	5	6
A	B	C	D	E	F
1	2	3	1	5	6

Representación arreglo:

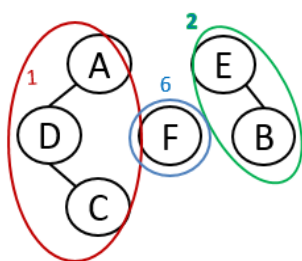
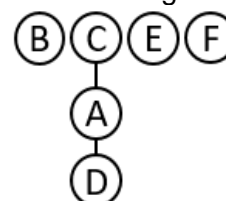


Unión D,C

Arreglo:

1	2	3	4	5	6
A	B	C	D	E	F
3	2	3	1	5	6

Representación arreglo:

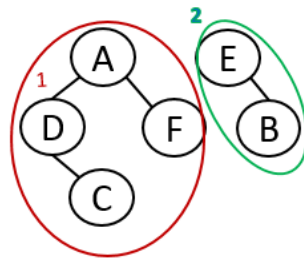
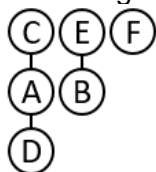


Unión E,B

Arreglo:

1	2	3	4	5	6
A	B	C	D	E	F
3	5	3	1	5	6

Representación arreglo:

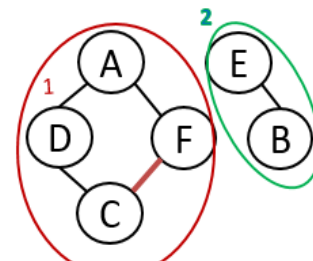
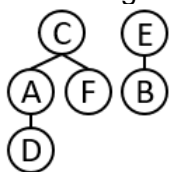


Unión A,F

Arreglo:

1	2	3	4	5	6
A	B	C	D	E	F
3	5	3	1	5	3

Representación arreglo:

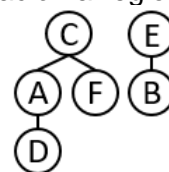


Unión C,F

Arreglo:

1	2	3	4	5	6
A	B	C	D	E	F
3	5	3	1	5	3

Representación arreglo:

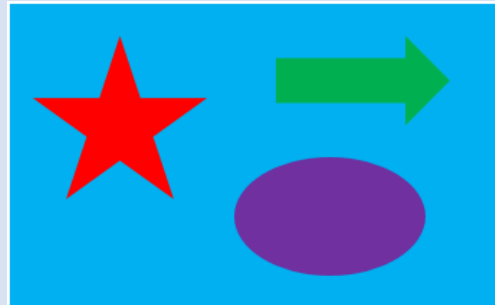
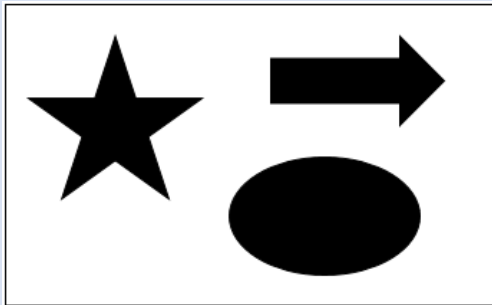


Para mejorar el rendimiento del algoritmo se realiza lo siguiente:

```
int Buscar(int v){  
    if( P[v]==v )  
        return v;  
    else  
        return P[v] = Buscar(P[v]);  
}
```

ACTIVIDAD

Programar una función que reciba una imagen en blanco y negro con diferentes figuras, y devuelva otra imagen donde cada figura tenga un color distinto utilizando el algoritmo Union Find Por ejemplo:

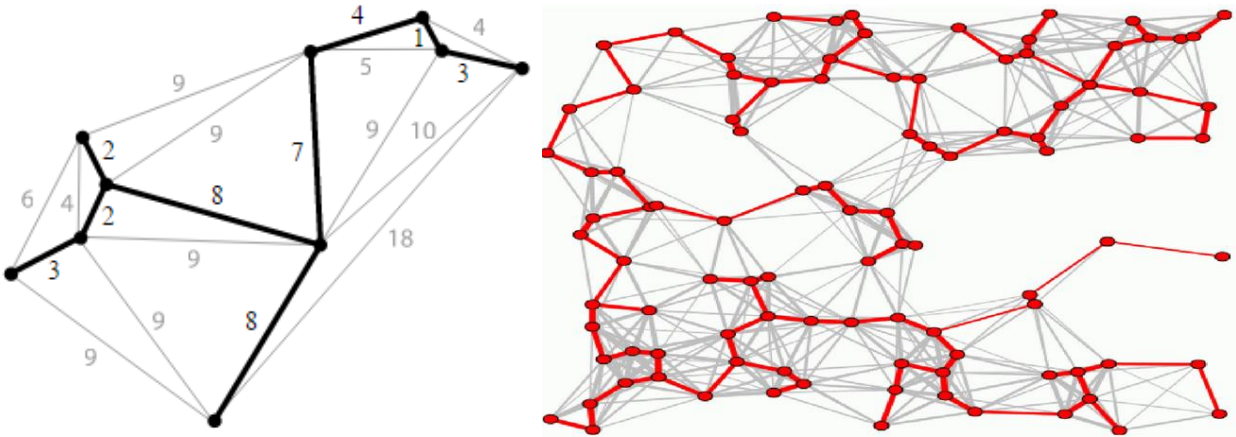


12.2 Árbol de mínima expansión

Un árbol de mínima expansión es un subgrafo de un grafo conexo y ponderado que:

- Incluye todos los vértices del grafo original
- Todos los vértices están conectados
- Incluye sólo algunas aristas, aquellas que minimizan la suma de sus valores

Ejemplos de grafos y su correspondiente árbol recubridor mínimo:



Las aplicaciones de encontrar el árbol recubridor mínimo son:

- Diseño de red (telefónica, eléctrica, hidráulica, cable TV, computadoras, carretera).
Ejemplo: Se quiere diseñar una red telefónica en un negocio con varias oficinas en diferentes ciudades. La compañía telefónica cobra diferentes cantidades de dinero para conectar diferentes pares de ciudades. La forma de calcular el conjunto de líneas que conecta a todas las oficinas a un costo mínimo es obteniendo el árbol de expansión mínimo.
- Modelación de Redes Bayesianas, para encontrar las relaciones entre variables aleatorias a partir de un conjunto de datos de entrenamiento.

12.2.1 Kruskal

El algoritmo Kruskal encuentra el árbol recubridor mínimo en un grafo conexo y ponderado. Fue publicado por primera vez en 1965 por Joseph Kruskal en Proceedings of the American Mathematical Society.

Algoritmo de Kruskal

Recibe como parámetro: Un grafo g conexo y ponderado

Crear árbol, con todos los nodos del grafo g pero sin aristas

Crear uf, como una estructura de datos Unión-Buscar con un conjunto para cada nodo

Crear pq, como un priority queue con todas las aristas del Grafo, ordenadas de menor a mayor

Mientras (el número de conjuntos en uf sea mayor a 1)

Sacar de pq la arista con menor valor

Conjunto1 = el conjunto del primer nodo en la arista

Conjunto2 = el conjunto del segundo nodo en la arista

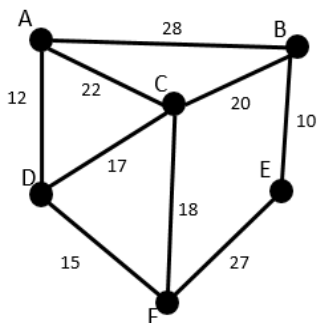
Si Conjunto1 es diferente del Conjunto2

Agregar la arista al arbol

Unir Conjunto1 y Conjunto2

Regresar el árbol

Por ejemplo:



Estructura Unión-Buscar

A	B	C	D	E	F
1	2	3	4	5	6

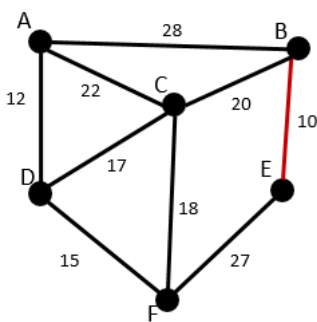
Cola de Prioridad

Nodo 1	Nodo 2	Distancia
B	E	10
A	D	12
D	F	15
C	D	17
C	F	18
B	C	20
A	C	22
E	F	27
A	B	28

Crea el árbol

Crea la estructura Unión-Buscar

Crea la Cola de Prioridad



Estructura Unión-Buscar

A	B	C	D	E	F
1	2	3	4	2	6

Cola de Prioridad

Nodo 1	Nodo 2	Distancia
B	E	10
A	D	12
D	F	15
C	D	17
C	F	18
B	C	20
A	C	22
E	F	27
A	B	28

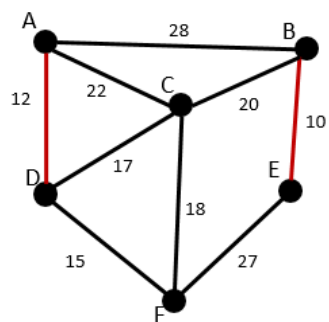
C1 = Buscar(B) = 2

C2 = Buscar(E) = 5

Como no son iguales

Se inserta la arista

Union(2,5)



Estructura Unión-Buscar

A	B	C	D	E	F
1	2	3	1	2	6

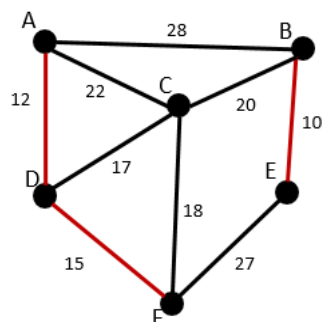
Cola de Prioridad

Nodo 1	Nodo 2	Distancia
B	E	10
A	D	12
D	F	15
C	D	17
C	F	18
B	C	20
A	C	22
E	F	27
A	B	28



C1 = Buscar(A) = 1
C2 = Buscar(D) = 4

Como no son iguales
Se inserta la arista
Union(1,4)



Estructura Unión-Buscar

A	B	C	D	E	F
1	2	3	1	2	1

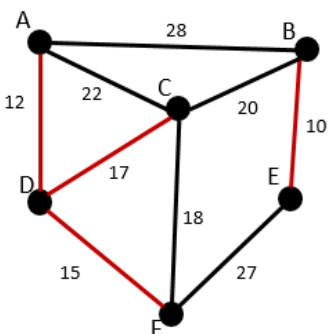
Cola de Prioridad

Nodo 1	Nodo 2	Distancia
B	E	10
A	D	12
D	F	15
C	D	17
C	F	18
B	C	20
A	C	22
E	F	27
A	B	28



C1 = Buscar(D) = 1
C2 = Buscar(F) = 6

Como no son iguales
Se inserta la arista
Union(1,6)



Estructura Unión-Buscar

A	B	C	D	E	F
1	2	1	1	2	1

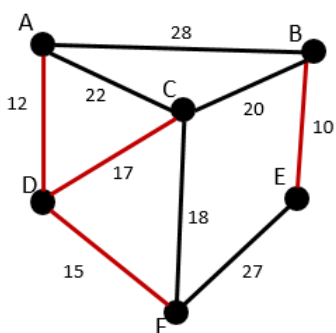
Cola de Prioridad

Nodo 1	Nodo 2	Distancia
B	E	10
A	D	12
D	F	15
C	D	17
C	F	18
B	C	20
A	C	22
E	F	27
A	B	28



C1 = Buscar(C) = 3
C2 = Buscar(D) = 1

Como no son iguales
Se inserta la arista
Union(3,1)



Estructura Unión-Buscar

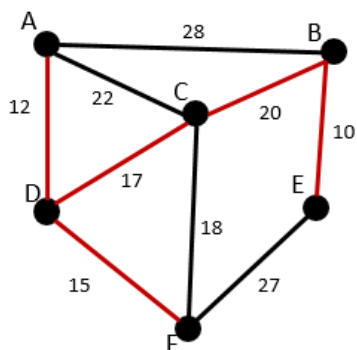
A	B	C	D	E	F
1	2	1	1	2	1

Cola de Prioridad

Nodo 1	Nodo 2	Distancia
B	E	10
A	D	12
D	F	15
C	D	17
C	F	18
B	C	20
A	C	22
E	F	27
A	B	28

C1 = Buscar(C) = 1
C2 = Buscar(F) = 1

Como son iguales
No hacer nada



Estructura Unión-Buscar

A	B	C	D	E	F
1	1	1	1	1	1

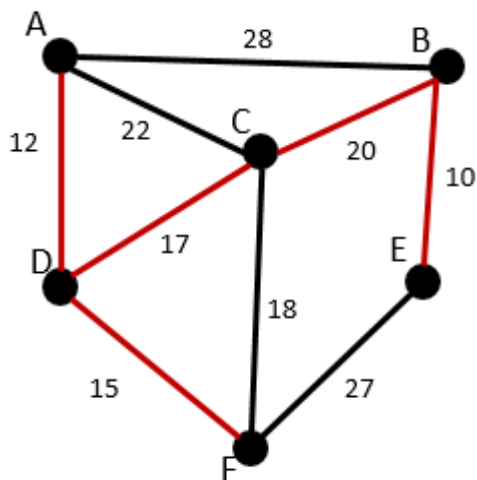
Cola de Prioridad

Nodo 1	Nodo 2	Distancia
B	E	10
A	D	12
D	F	15
C	D	17
C	F	18
B	C	20
A	C	22
E	F	27
A	B	28

C1 = Buscar(B) = 2
C2 = Buscar(C) = 1

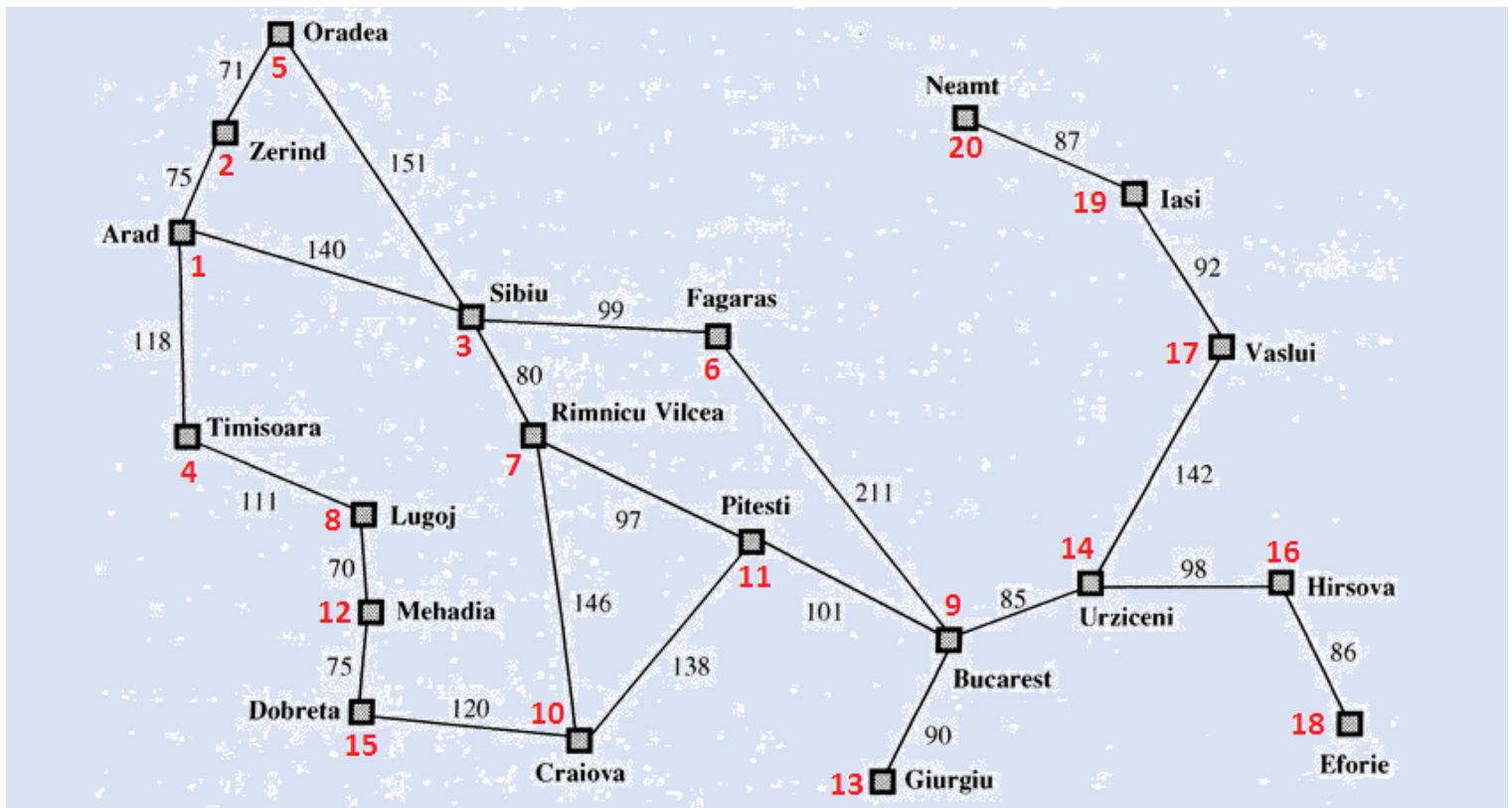
Como no son iguales
Se inserta la arista
Union(2,1)

Solo hay un conjunto, por lo tanto el árbol de expansión mínima esta creado.



EJERCICIOS

1. Realice un programa que implemente el Algoritmo de Kruskal para calcular el árbol de expansión mínima del Mapa de Rumania.



12.2.2 Prim

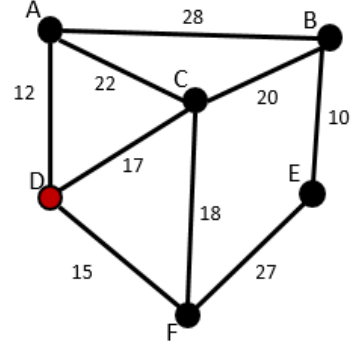
El algoritmo de Prim, al igual que el algoritmo de Kruskal, encuentra el árbol de expansión mínima en un grafo conexo y ponderado. Fue publicado en 1957 por Robert C. Prim.

Algoritmo de Prim
 Recibe como parámetro: Un grafo g conexo y ponderado

Crear árbol, sin nodos y sin aristas
 Crear pq, como un priority queue de aristas (inicialmente vacía)
 Seleccionar un nodo inicial al azar
 Agregar el nodo al árbol
 Agregar a la pq todas las aristas de ese nodo

Mientras (pq no este vacía)
 Sacar de pq la arista con menor valor
 Nodo1 = el primer nodo de la arista
 Nodo2 = el segundo nodo de la arista
 Si un nodo si está en el árbol pero el otro no
 Agregar el nodo que faltaba al árbol
 Agregar la arista al árbol
 Agregar a pq las aristas del nuevo nodo que cumplan con la característica de que:
 un nodo si esté en el árbol y el otro no
 Regresar el árbol

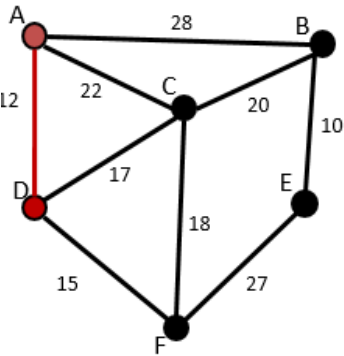
Por ejemplo:



Nodo 1	Nodo 2	Distancia
D	A	12
D	C	17
D	F	15

Crea el árbol
 Crea la priority queue pq

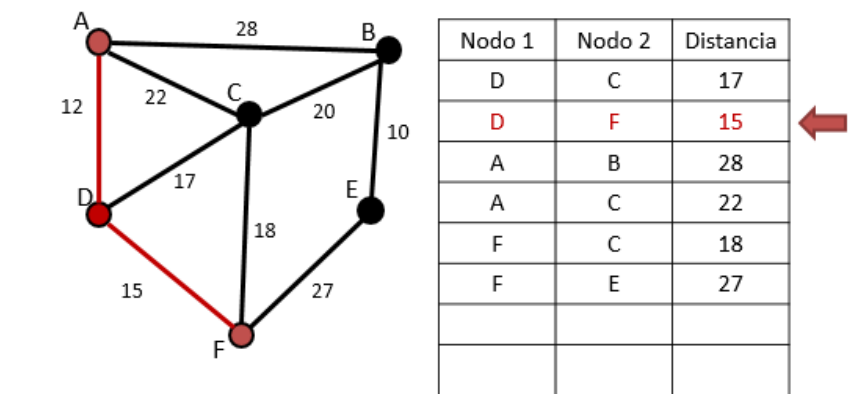
 Agrega D al árbol
 Agrega a pq las aristas de D



Nodo 1	Nodo 2	Distancia
D	A	12
D	C	17
D	F	15
A	B	28
A	C	22

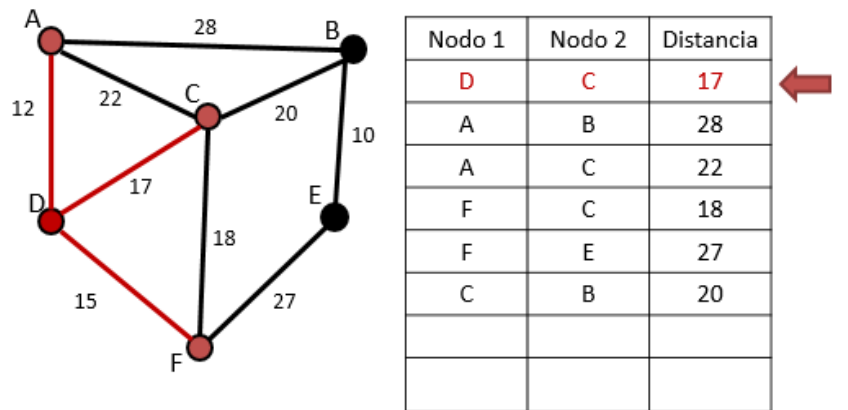
Selecciona la arista con menor distancia

 Como D si estaba en el árbol y A no:
 Agrega el nodo A al árbol
 Agrega la arista al árbol
 Agrega las aristas de A, que cumplen que
 un nodo si está en el árbol y el otro no



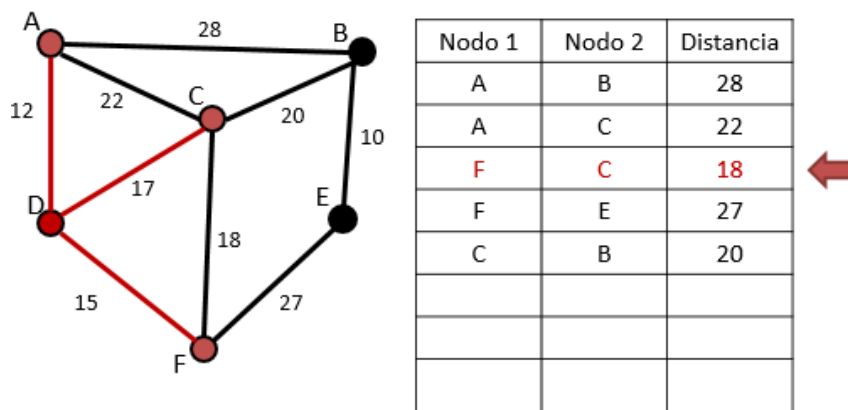
Selecciona la arista con menor distancia

Como D si está en el árbol y F no:
Agrega el nodo F al árbol
Agrega la arista al árbol
Agrega las aristas de F, que cumplen que un nodo si está en el árbol y el otro no



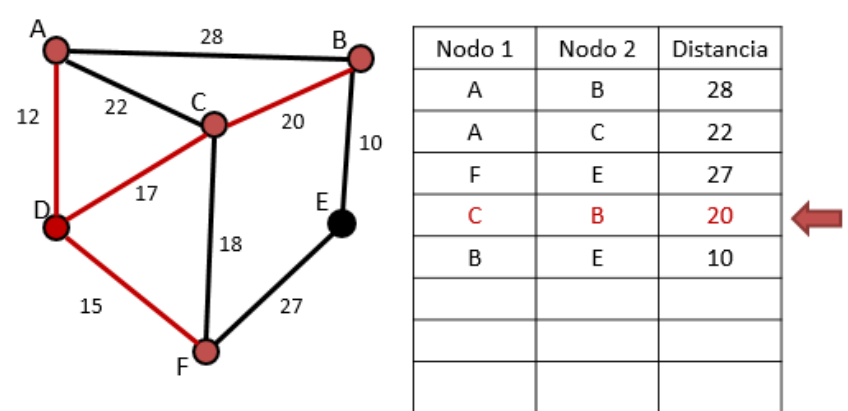
Selecciona la arista con menor distancia

Como D si está en el árbol y C no:
Agrega el nodo C al árbol
Agrega la arista al árbol
Agrega las aristas de C, que cumplen que un nodo si está en el árbol y el otro no



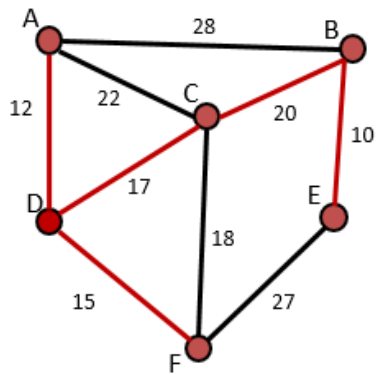
Selecciona la arista con menor distancia

Como F y C están en el árbol
Sólo quita la arista de pq



Selecciona la arista con menor distancia

Como C si está en el árbol y B no:
Agrega el nodo B al árbol
Agrega la arista al árbol
Agrega las aristas de B, que cumplen que un nodo si está en el árbol y el otro no

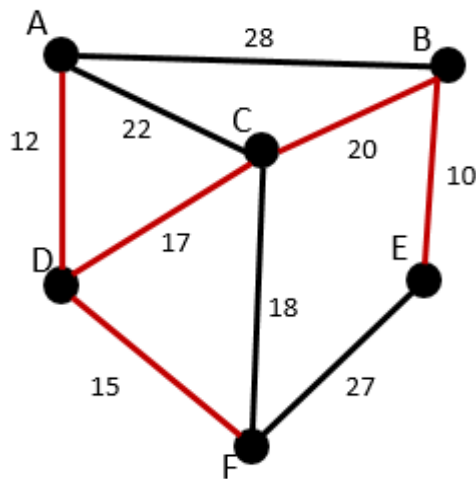


Nodo 1	Nodo 2	Distancia
A	B	28
A	C	22
F	E	27
B	E	10

Selecciona la arista con menor distancia

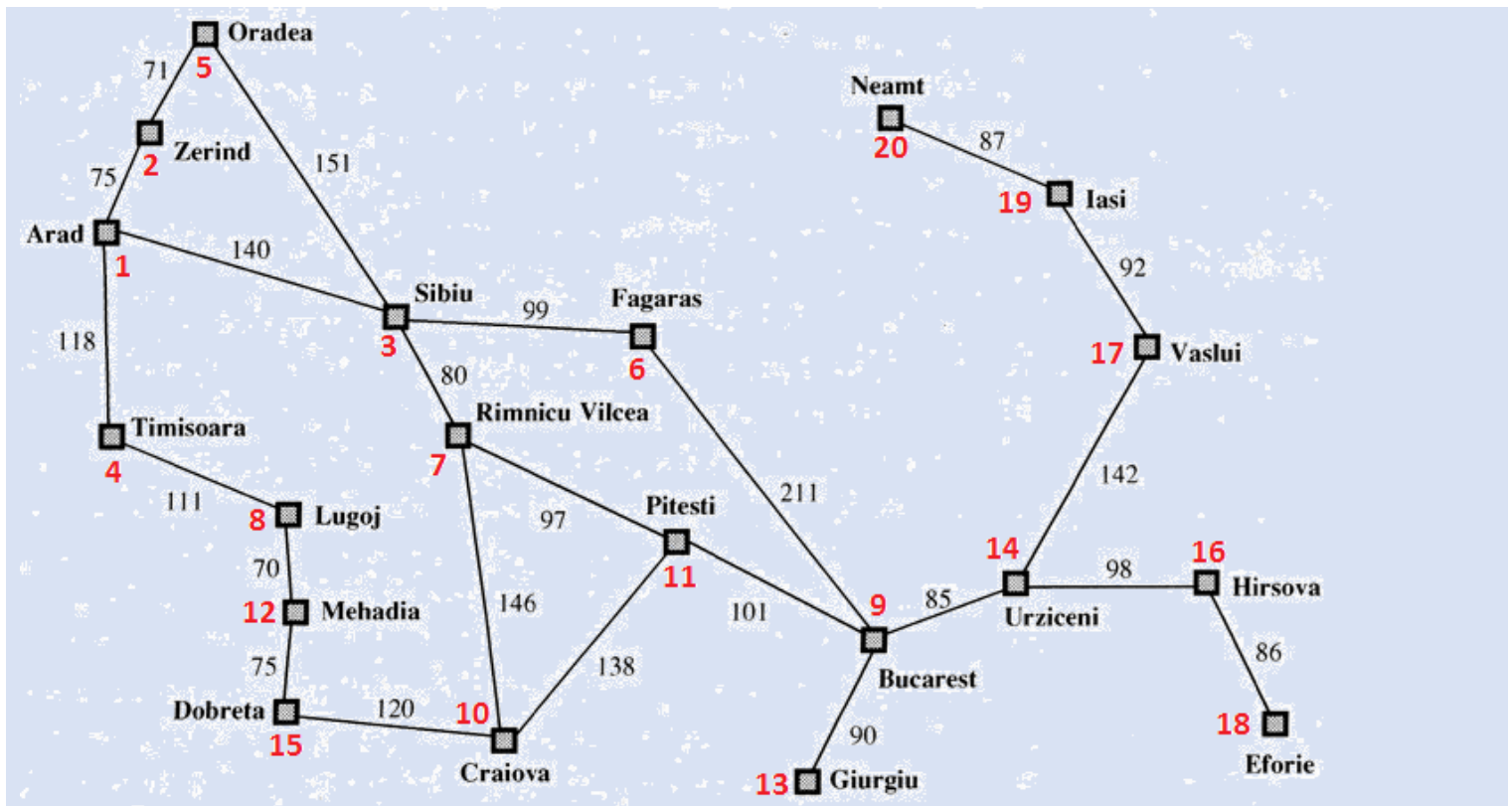
Como B si está en el árbol y E no:
Agrega el nodo E al árbol
Agrega la arista al árbol
Agrega las aristas de E, que cumplen que un nodo si está en el árbol y el otro no

Como ya todos los nodos están en el árbol, se regresa el árbol de expansión mínima.



EJERCICIOS

1. Realice un programa que implemente el Algoritmo de Prim para calcular el árbol de expansión mínima del Mapa de Rumania.



12.3 Distancias mínimas

12.3.1 Algoritmo de Dijkstra

El algoritmo Dijkstra, también llamado Algoritmo de Caminos Cortos, calcula las distancias mínimas desde un nodo y el resto de los nodos en un grafo conexo y ponderado. Es un algoritmo Greedy y fue descrito por Edsger Dijkstra en 1959.

Algoritmo de Dijkstra

Recibe como parámetro: Grafo conexo y ponderado
Nodo inicial, a partir del cual se calcularán los caminos más cortos

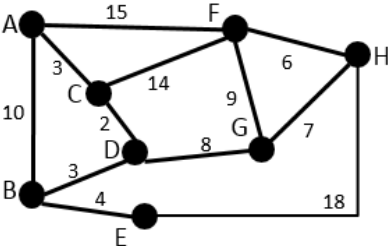
Crear una estructura de datos para almacenar: los nodos visitados, las distancias mínimas a cada nodo y los caminos más cortos.
Crear una Cola de Prioridad cp

Marcar todos los nodos como no visitados y las distancias como infinito
Marcar la distancia del nodo inicial como 0, y como camino el nodo inicial
Agregar el nodo inicial a cp con la distancia como su prioridad, en este caso 0.

Mientras (cp no este vacía)
 Sacar de cp el nodo con menor distancia, al cual llamaremos n
 Marcar el nodo n como visitado
 Para todos los nodos conectados al nodo n que no hayan sido visitados
 Si la distancia de n + la distancia de n al nodo conectado < distancia del nodo conectado
 Actualizar la distancia y el camino del nodo conectado
 Si el nodo conectado no ha sido visitado
 Agregarlo a la cola de prioridad

Regresar los caminos más cortos

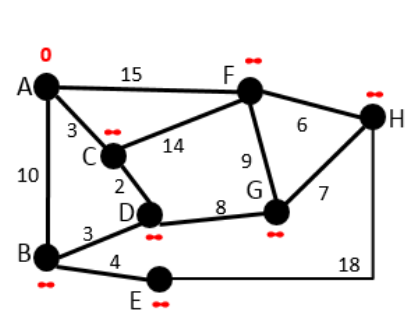
Por ejemplo: Suponga que se desea calcular la distancia más corta del nodo A hacia todos los nodos del grafo.



Nodo	Visitado	Distancia	Camino
A			
B			
C			
D			
E			
F			
G			
H			

Crear una Estructura de Datos para guardar los valores de:

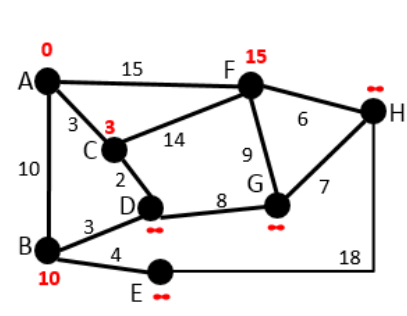
- Distancias
- Visitados
- Caminos



Nodo	Visitado	Distancia	Camino
A		0	A
B		Inf	
C		Inf	
D		Inf	
E		Inf	
F		Inf	
G		Inf	
H		Inf	

Inicializar las distancias con un valor muy grande en todos los nodos, menos el inicial: A.

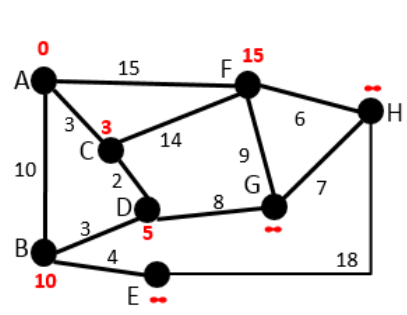
En el nodo inicial la distancia será 0 y el camino el nodo inicial.



Nodo	Visitado	Distancia	Camino
A	x	0	A
B		10	A,B
C		3	A,C
D		Inf	
E		Inf	
F		15	A,F
G		Inf	
H		Inf	

Visitar el nodo con menor distancia que no haya sido visitado:

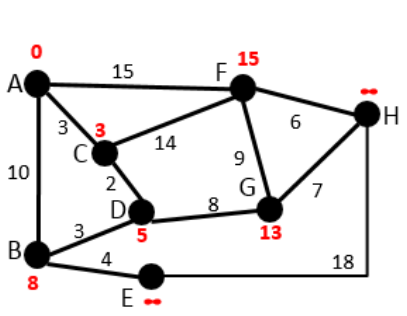
- Actualizar los caminos y las distancias de los nodos conectados a él si las distancias por ese camino son menores a las actuales



Nodo	Visitado	Distancia	Camino
A	x	0	A
B		10	A,B
C	x	3	A,C
D		5	A,C,D
E		Inf	
F		15	A,F
G		Inf	
H		Inf	

Visitar el nodo con menor distancia que no haya sido visitado:

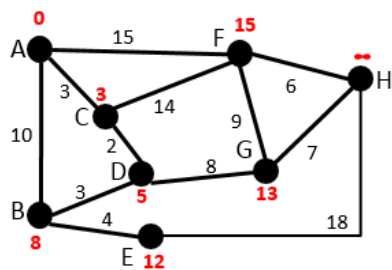
- Actualizar los caminos y las distancias de los nodos conectados a él si las distancias por ese camino son menores a las actuales



Nodo	Visitado	Distancia	Camino
A	x	0	A
B		8	A,C,D,B
C	x	3	A,C
D	x	5	A,C,D
E		Inf	
F		15	A,F
G		13	A,C,D,G
H		Inf	

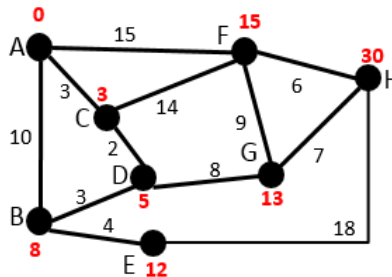
Visitar el nodo con menor distancia que no haya sido visitado:

- Actualizar los caminos y las distancias de los nodos conectados a él si las distancias por ese camino son menores a las actuales



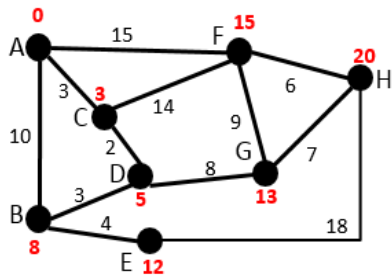
Nodo	Visitado	Distancia	Camino
A	x	0	A
B	x	8	A,C,D,B
C	x	3	A,C
D	x	5	A,C,D
E		12	A,C,D,B,E
F		15	A,F
G		13	A,C,D,G
H		Inf	

Visitar el nodo con menor distancia que no haya sido visitado:
- Actualizar los caminos y las distancias de los nodos conectados a él si las distancias por ese camino son menores a las actuales



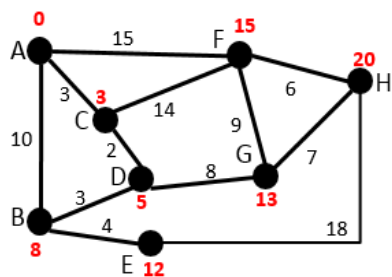
Nodo	Visitado	Distancia	Camino
A	x	0	A
B	x	8	A,C,D,B
C	x	3	A,C
D	x	5	A,C,D
E	x	12	A,C,D,B,E
F		15	A,F
G		13	A,C,D,G
H		30	A,C,D,B,E,H

Visitar el nodo con menor distancia que no haya sido visitado:
- Actualizar los caminos y las distancias de los nodos conectados a él si las distancias por ese camino son menores a las actuales



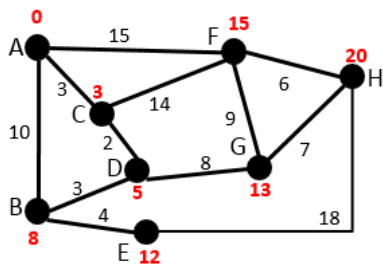
Nodo	Visitado	Distancia	Camino
A	x	0	A
B	x	8	A,C,D,B
C	x	3	A,C
D	x	5	A,C,D
E	x	12	A,C,D,B,E
F		15	A,F
G	x	13	A,C,D,G
H		20	A,C,D,G,H

Visitar el nodo con menor distancia que no haya sido visitado:
- Actualizar los caminos y las distancias de los nodos conectados a él si las distancias por ese camino son menores a las actuales



Nodo	Visitado	Distancia	Camino
A	x	0	A
B	x	8	A,C,D,B
C	x	3	A,C
D	x	5	A,C,D
E	x	12	A,C,D,B,E
F	x	15	A,F
G	x	13	A,C,D,G
H		20	A,C,D,G,H

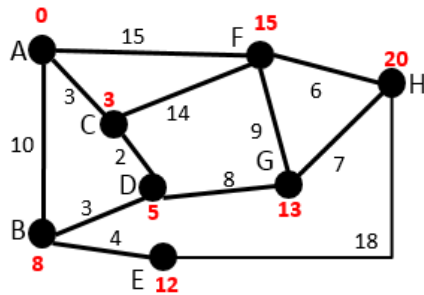
Visitar el nodo con menor distancia que no haya sido visitado:
- Actualizar los caminos y las distancias de los nodos conectados a él si las distancias por ese camino son menores a las actuales



Nodo	Visitado	Distancia	Camino
A	x	0	A
B	x	8	A,C,D,B
C	x	3	A,C
D	x	5	A,C,D
E	x	12	A,C,D,B,E
F	x	15	A,F
G	x	13	A,C,D,G
H	x	20	A,C,D,G,H

Visitar el nodo con menor distancia que no haya sido visitado:
 - Actualizar los caminos y las distancias de los nodos conectados a él si las distancias por ese camino son menores a las actuales

Cuando todos los nodos se hayan visitado, ya se pueden obtener las distancias mínimas y los caminos más cortos.



Nodo	Distancia	Camino
A	0	A
B	8	A,C,D,B
C	3	A,C
D	5	A,C,D
E	12	A,C,D,B,E
F	15	A,F
G	13	A,C,D,G
H	20	A,C,D,G,H

EJERCICIOS

1. Realice un programa que implemente el Algoritmo de Dijkstra para calcular los caminos más cortos de la ciudad que el usuario indique a las ciudades restantes.

12.3.2 Algoritmo de Floyd - Warshall

El algoritmo Floyd – Warshall calcula el camino de distancia mínima entre dos nodos cualesquiera utilizando programación dinámica. Fue descrito en 1962 por Robert Floyd y por Stephen Warshall.

Algoritmo de Floyd - Warshall

Recibe como parámetros: Grafo conexo y ponderado

Crear una estructura de datos para almacenar: los caminos y distancias más cortas entre todos los pares de nodos, inicializándola como sigue:

- Para dos nodos conectados: la distancia es igual al valor de la arista
- Para dos nodos NO conectados: la distancia es infinito

Para todos los nodos n

Revisar todos los pares de nodos (i,j) que no incluyan a n

Calcular la distancia i – n – j

Si la distancia i – n – j es más corta que la distancia i – j

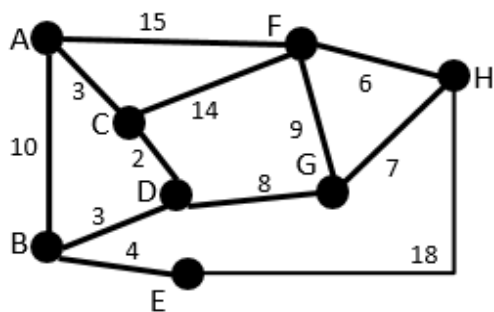
Actualizar las distancias y caminos del par de todos (i,j) incluyendo al nodo n

Regresar los caminos más cortos

Ejemplo:

Crear una matriz que almacene la información de los caminos y distancias más cortas entre nodos, inicializándola como sigue:

- Nodos no conectados: distancia infinita
- Nodos conectados: distancia igual al valor de la arista

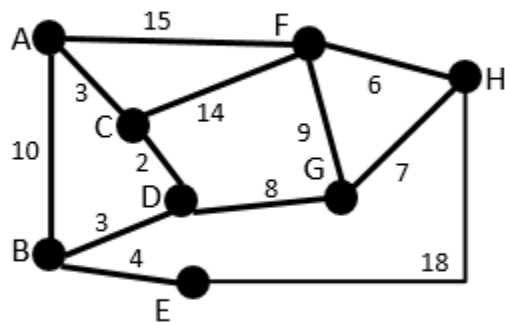


	A	B	C	D	E	F	G	
B	10 BA							
C	3 CA	∞ CB						
D	∞ AD	3 DB	2 DC					
E	∞ EA	4 EB	∞ EC	∞ ED				
F	15 FA	∞ FB	14 FC	∞ FD	∞ FE			
G	∞ GA	∞ GB	∞ GC	8 GD	∞ GE	9 GF		
H	∞ HA	∞ HB	∞ HC	∞ HD	18 HE	6 HF	7 HG	

Visitar el nodo A

Revisar todas las casillas (i,j) de la matriz, excepto la fila y la columna del nodo A

- Calcular la distancia Nodo i – Nodo A – Nodo j
- Si la distancia i – A- j es más corta que Nodo i – Nodo j
Se actualiza la distancia y el camino de i – j incluyendo a A

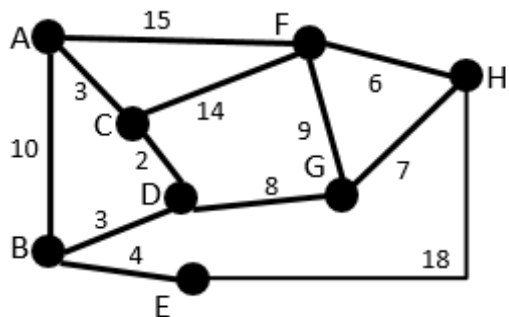


	A	B	C	D	E	F	G
B	10 BA						
C	3 CA	13 CAB					
D	∞ AD	3 DB	2 DC				
E	∞ EA	4 EB	∞ EC	∞ ED			
F	15 FA	25 FAB	14 FC	∞ FD	∞ FE		
G	∞ GA	∞ GB	∞ GC	8 GD	∞ GE	9 GF	
H	∞ HA	∞ HB	∞ HC	∞ HD	18 HE	6 HF	7 HG

Visitar el nodo B

Revisar todas las casillas (i,j) de la matriz, excepto la fila y la columna del nodo B

- Calcular la distancia Nodo i – Nodo A – Nodo j
- Si la distancia i – A- j es más corta que Nodo i – Nodo j
Se actualiza la distancia y el camino de i – j incluyendo a A



	A	B	C	D	E	F	G
B	10 BA						
C	3 CA	13 CAB					
D	13 ABD	3 DB	2 DC				
E	14 EBA	4 EB	17 EBAC	7 EBD			
F	15 FA	25 FAB	14 FC	28 FABD	29 FABE		
G	∞ GA	∞ GB	∞ GC	8 GD	∞ GE	9 GF	
H	∞ HA	∞ HB	∞ HC	∞ HD	18 HE	6 HF	7 HG

Seguir con el nodo C, D, E, F, y G y regresar los caminos más cortos.

12.4 Orden topológico

La ordenación topológica de un grafo dirigido consiste en poner los nodos de un grafo en una línea en orden ascendente de tal forma que se cumpla la regla de precedencia para todas las conexiones.

Ejemplo: ordenación topológica de tareas

Apéndice A. Processing

Processing es un lenguaje de programación y entorno de desarrollo basado en Java. Es de fácil utilización y que sirve como medio para la enseñanza y producción de proyectos multimedia e interactivos.

Interface gráfica

Configuración ventana

```
size( ancho_ventana, alto_ventana);
width
height
```

Descripción

Establece el tamaño de la pantalla
Variable global que almacena el ancho de la pantalla
Variable global que almacena el largo de la pantalla

Colores

Color en escala de grises

Descripción

Consiste en establecer un color en escala de grises desde el negro (0) hasta el blanco (255). Entonces:

$$0 \leq \text{color_escala_grises} \leq 255$$

Color en RGB

Consiste en establecer un color utilizando el formato RGB, es decir, combinando los tres colores básicos: rojo, verde y azul.

$$0 \leq \text{rojo, verde, azul} \leq 255$$

```
background( color_escala_grises );
background( rojo, verde, azul );
fill( color_escala_grises );
fill( rojo, verde, azul);
stroke( color_escala_grises);
stroke( rojo, verde, azul);
color c;
c = get(x,y);
set(x,y,c);
```

Establece el color de fondo

Carga un color para rellenar figuras y pintar texto

Carga un color para pintar líneas y bordes de figuras

La clase color, permite almacenar información del color en un píxel
get -> obtiene el color en un píxel en específico
set -> define el color del píxel en la posición x, y

Figuras básicas

```
rect( x, y, ancho, alto);
```

```
ellipse(x, y, ancho, alto);
```

```
line( x1, y1, x2, y2);
```

Descripción

Pinta un rectángulo con la esquina superior derecha en el punto (x, y). O un cuadrado si el ancho y alto son iguales.

Pinta una elipse centrada en (x, y). O un círculo si el ancho y el alto son iguales.

Pinta una línea desde el punto (x1, y1) y el punto (x2, y2).

Texto

```
print o println( texto + variable + ... );
textSize( tamaño);
text( texto, x, y);
```

Descripción

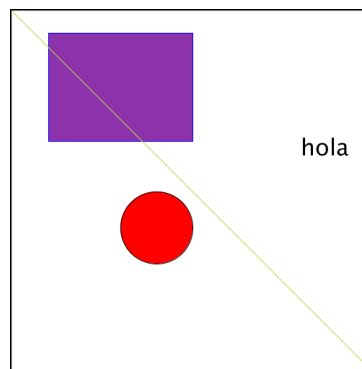
Imprime en la ventana de resultados

Establece el tamaño del texto

Pinta el texto (que puede ser un texto "Hola mundo" o una variable string) en la posición x, y.

Ejemplo:

```
size(500,500);
background(255);
fill(140,51,172);
stroke(0,0,255);
rect(50,30,200,150);
fill(255,0,0);
stroke(0);
ellipse(200,300,100,100);
stroke(196,215,53);
line(0,0,500,500);
fill(0);
textSize(32);
text("hola",400,200);
```



Funciones: setup() y draw()

Setup(): Se utiliza para configurar la pantalla e inicializar variables. Se ejecuta al inicio del programa una única vez.

Draw(): Se ejecuta varias veces, hasta que el programa se termine. El objetivo de esta función es mostrar un frame por ejecución, es decir, se pueden realizar animaciones.

Ejemplo: Pelota que cae

```
int x = 0;
int y = 0;
void setup(){
  size(500,500);
  background(255);
}
void draw(){
  x = x+5;
  y = y+5;
  background(255);
  fill(0);
  ellipse(x,y,50,50);
}
```

Ejemplo: Pelota que rebota

```
int x;
int y;
String direccion;
void setup(){
  size(500,400);
  background(255);

  x=0;
  y = height/2;
  direccion="derecha";
}
void draw(){
  if( direccion.equals("derecha") ){
    x += 5;
    if( x>= width) direccion = "izquierda";
  }else{
    x -= 5;
    if( x<= 0 ) direccion = "derecha";
  }
  background(255);
  fill(0);
  ellipse(x,y,50,50);
}
```

Función: mouseClicked()

mouseClicked(): Se manda llamar cuando se hace click con el mouse

mouseX: Es la posición actual en X del mouse

mouseY: Es la posición actual en Y del mouse

Ejemplo: Al dar click, se pinta un rectángulo en la posición de mouse

int x, y;

```
void setup(){
  background(255);
  size(600,400);
}
void draw() {
  background(255);
  fill(0);
  rect( x, y, 50,50);
}
void mouseClicked() {
  x = mouseX;
  y = mouseY;
  System.out.println(x+", "+y);
}
```

Función: keyPressed()

keyPressed(): Se manda llamar cuando se presiona una tecla

key: Es el valor de la tecla que se presionó, cuando es un carácter imprimible (letras y símbolos)

keyCode: Es el valor de la tecla que se presionó, cuando es un carácter no imprimible (ENTER, RETURN, ALT, SHIFT, CONTROL, TAB, BACKSPACE, ESC, RETURN, UP, DOWN, LEFT, RIGHT)

Ejemplo: Al dar click, se pinta un rectángulo en la posición de mouse

int x, y;

String direccion;

```
void setup(){
  background(255);
  size(600,400);
  direccion = "derecha";
}
void draw() {
  if( direccion.equals("derecha"))    x = x+1;
  else if( direccion.equals("izquierda")) x = x-1;
  else if( direccion.equals("abajo"))   y = y+1;
  else if( direccion.equals("arriba"))  y = y-1;
  background(255);
  fill(0);
  ellipse(x,y,50,50);
}
void keyPressed() {
  if( key=='i' || keyCode == UP )      direccion = "arriba";
  else if( key=='j' || keyCode == LEFT ) direccion = "izquierda";
  else if( key=='m' || keyCode == DOWN ) direccion = "abajo";
  else if( key=='l' || keyCode == RIGHT ) direccion = "derecha";
}
```

Apéndice B. Java

Graphics

```
import java.awt.EventQueue;
import java.awt.Graphics;
import java.awt.Graphics2D;
import javax.swing.JFrame;
import javax.swing.JPanel;
import java.awt.Color;

class Surface extends JPanel {
    private void doDrawing(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        g2d.setPaint(Color.pink);
        g2d.drawString("Java 2D", 50, 50);
        g2d.setPaint(Color.blue);
        g2d.drawLine(0, 0, 100, 100);
    }
    @Override
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        doDrawing(g);
    }
}

public class Graficos extends JFrame {
    public Graficos() {
        initUI();
    }
    private void initUI() {
        add(new Surface());
        setTitle("Simple Java 2D example");
        setSize(300, 200);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    }
    public static void main(String[] args) {

        EventQueue.invokeLater(new Runnable() {
            @Override
            public void run() {
                Graficos ex = new Graficos();
                ex.setVisible(true);
            }
        });
    }
}
```

ArrayList

Es una estructura de datos que almacena datos como un arreglo, pero con la diferencia que es dinámico, es decir, su tamaño puede cambiar.

Función	Descripción
<code>void add(Object o)</code>	Agrega un elemento al arreglo
<code>int size()</code>	Devuelve el número de elementos en el arreglo
<code>void set(int i, Object o);</code>	Cambia el elemento de la posición i, con el objeto o. Equivalente a: <code>arreglo[i] = o;</code>
<code>Object get(i);</code>	Devuelve el elemento de la posición i Equivalente a: <code>Objeto o = arreglo[i];</code>

Ejemplo:

<code>ArrayList<Integer> lista = new ArrayList<Integer>();</code>	1. 90
<code>lista.add(90);</code>	2. 25
<code>lista.add(25);</code>	3. 28
<code>lista.add(28);</code>	4. 57
<code>lista.add(57);</code>	
<code>for(int i=0; i<lista.size(); i++)</code>	1. 90
<code> println((i+1)+" ". "+lista.get(i));</code>	2. 18
<code>set(1,18);</code>	3. 28
<code>lista.add(64);</code>	4. 57
<code>println("");</code>	5. 64
<code>for(int i=0; i<lista.size(); i++)</code>	
<code> println((i+1)+" ". "+lista.get(i));</code>	

Wrapper o envoltorio

Muchas de las funcionalidades de Java no funcionan con los datos primitivos: int, float, double, char, etc. Los wrappers son clases que contienen la información de variables de tipos de datos primitivos más algunas funciones de conversiones.

Tipo de dato primitivo	Wrapper
int	Integer
float	Float
double	Double
char	Character
boolean	Boolean

Stack

Es una estructura de datos, pila en español, que almacena valores tipo LIFO last in first out (último en entrar primero en salir).

Función	Descripción
<code>void add(Object o);</code>	Agrega un elemento a la pila
<code>Objet pop();</code>	Saca el último elemento que fue agregado
<code>Objet peek();</code>	Muestra el último elemento que fue agregado, sin sacarlo de la pila
<code>boolean isEmpty();</code>	Indica si la pila está o no vacía

Queue

Es una estructura de datos, cola en español, que almacena valores tipo FIFO first in first out (primero en entrar primero en salir).

Función	Descripción
<code>void add(Object o);</code>	Agrega un elemento
<code>Objet poll();</code>	Saca el primer elemento que fue agregado
<code>Objet peek();</code>	Muestra el primer elemento que fue agregado
<code>boolean isEmpty();</code>	Indica si hay elementos o no

Ejemplo:

```
void setup(){
  Queue<Integer> queue = new LinkedList<Integer>();
  queue.add( 1 );      1
  queue.add( 2 );      2
  queue.add( 3 );      3
  while( !queue.isEmpty() ){
    println(queue.poll()); 3
  }                      2
  println();             1
  Stack<Integer> stack = new Stack<Integer>();
  stack.add( 1 );
  stack.add( 2 );
  stack.add( 3 );
  while( !stack.isEmpty() ){
    println(stack.pop());
  }
  println();
}
```

PriorityQueue

```

class Nodo{
    float probabilidad;
    char valor;
    Nodo(float probabilidad , char valor){
        this.probabilidad = probabilidad;
        this.valor = valor;
    }
    public String toString(){
        return ""+valor;
    }
}

public class CodigoHuffman {

    PriorityQueue<Nodo> Q = new PriorityQueue<Nodo>(6, new Comparator<Nodo>(){
        public int compare(Nodo n1, Nodo n2){
            if( n1.probabilidad > n2.probabilidad ) return -1;
            else if ( n1.probabilidad == n2.probabilidad) return 0;
            else return 1;
        }
    });

    Q.add( new Nodo(0.25f, '1') );
    Q.add( new Nodo(0.12f, '2') );
    Q.add( new Nodo(0.3f, '3') );
    Q.add( new Nodo(0.11f, '4') );
    Q.add( new Nodo(0.10f, '5') );
    Q.add( new Nodo(0.12f, '6') );
    while(!Q.isEmpty())
        System.out.println(Q.poll());
}

```

3
1
2
6
4
5