

## **72.34 Estructura de Datos y Algoritmos**

### **Apuntes**



**1Q - 2024**

## Índice

<b>Unidad 1 - Algoritmos.....</b>	<b>4</b>
Tiempo de Ejecución.....	4
Maven.....	4
JUnit 5.....	9
Análisis de Algoritmos.....	12
Complejidad Espacial.....	17
<b>Unidad 2 - Algoritmos para Búsqueda en Texto.....</b>	<b>19</b>
Definiciones.....	19
Data Quality - Matching.....	21
Soundex.....	22
Metaphone.....	24
Algoritmo de Levenshtein (Levenshtein Distance).....	24
Q-Grams (N-Grams).....	28
Comparación de Algoritmos.....	30
Búsqueda Exacta.....	32
Algoritmo de Fuerza Bruta o Naive.....	32
Algoritmo Knuth-Morris-Pratt (KMP).....	32
Lucene.....	35
TermQuery.....	43
QueryBuilder.....	47
Query de un Término.....	50
Query Multi-Término.....	53
<b>Unidad 3 - Estructuras Lineales.....</b>	<b>55</b>
Características de Índices.....	56
Teorema Maestro.....	57
Ordenación de Arreglos.....	61
Generics.....	65
Stack.....	71
Parser de Precedencia de Operadores.....	73
Queue.....	80
Índices y Arreglos Ordenadas.....	81
<b>Unidad 4 - Hashing (Dispersión).....</b>	<b>86</b>
Tabla de Hashing.....	86
Prehash.....	87
Factor de Carga.....	89
Colisiones.....	90
Hashing de Java.....	93
<b>Unidad 5 - Árboles.....</b>	<b>94</b>
Árbol Binario.....	94
Árbol Binario de Expresiones.....	94
Árbol Binario Completo.....	98
Árbol de Búsqueda Binario.....	99

AVL: “BST balanceado por altura”.....	100
Árbol de Fibonacci.....	103
Árbol Multicamino M-ario (orden M).....	105
Árboles Multicamino Balanceados.....	105
Eficiencia de los árboles.....	108
<b>Unidad 6 - Grafos.....</b>	<b>110</b>
Casos de Uso.....	110
Tipos de Grafos.....	110
Grafo Simple.....	113
Generación de Grafos.....	114
Recorridos en Grafos.....	116
Algoritmo de Dijkstra.....	118
<b>Unidad 7 - Heurísticas.....</b>	<b>120</b>
Técnica de Búsqueda Exhaustiva.....	120
Técnicas de Backtracking.....	121
Backtracking + Programación Dinámica.....	121
Problema de las 8 Reinas.....	122
Técnicas Algorítmicas Estudiadas.....	123

## Unidad 1 - Algoritmos

### Análisis de Algoritmos

Si tenemos dos algoritmos que resuelven un problema, ¿Cómo sabemos cuál elegir?

El análisis de algoritmos fue introducido por Donald Knuth, y nos permite "caracterizar" la **cantidad de recursos computacionales** que usará el mismo cuando se aplique a ciertos datos y evaluar así "su performance".

Vamos a tener en cuenta:

1. El tiempo de ejecución (runtime analysis/time complexity)
2. El espacio que utilizan (space complexity)

### Tiempo de Ejecución

¿Cómo lo medimos? Tenemos dos maneras:

- Empírica (con cronómetro): es más fácil de medir.
- Teórico (la complejidad - como vimos en Discreta): hay que demostrar.

Si se desarrollan dos algoritmos, y tienen dos tiempos distintos de ejecución teóricos, uno va a ser mejor y no va a valer la pena seguir esforzándose con el más lento. Si el tiempo teórico es el mismo, entonces sí se puede discutir cómo mejorarlo (por ejemplo, recorrer una sola vez) tomando el tiempo empírico.

Consideraciones:

- Si usamos en las aplicaciones bibliotecas propias, crear un "proyecto Java" sirve.
- Pero si usamos bibliotecas externas (otros jars), es complicado mantener actualizaciones y versiones de esta manera, dado que "importamos" jars estáticamente.
- Existe algo muy útil, para los casos en que usamos bibliotecas externas, y por supuesto lo podemos usar aún para aplicaciones nuestras.

Vamos a usar una herramienta llamada Maven, que nos va a permitir setear proyectos a gran escala.

### Maven

Es una herramienta para crear y administrar proyectos en Java. Como objetivos se propone:

- Proporcionar un sistema de construcción uniforme
- Proporcionar información de calidad del proyecto
- Proporcionar pautas para el desarrollo de mejores prácticas
- Permitir la migración transparente a nuevas funcionalidades

Además, permite declarar dependencias para utilizar librerías externas (o nuestras).

Maven compila el proyecto y arma el .jar, además de permitir testear y guardar el proyecto localmente para usarlo como dependencia en otros proyectos.

### Goals & Build Phases

Goals: tareas específicas dentro del build

mvn jar:jar → armar un jar desde el código ya compilado

mvn dependency:tree → muestra las dependencias

mvn exec:java → corre el proyecto

### Build Phases: etapas del armado del proyecto

mvn compile → compila el código

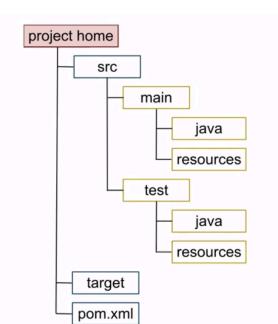
mvn test → corre los tests

mvn package → arma el jar

mvn install → guarda el proyecto en el repo local

Esta última es la que más vamos a usar, porque implica todas las demás.

### Estructura de Proyecto



Dentro de source (src) y main, en java incluimos todos los archivos .java con código. En resources se incluyen los archivos que no son código, como por ejemplo imágenes, texto, etc.

El archivo pom es de configuración único. Ahí se hacen todos los seteos que uno quiera. Vamos a ir viendo las distintas configuraciones.

### Configuraciones Mínimas

```

<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
  http://maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>
  <groupId>ar.edu.itba.eda</groupId>
  <artifactId>Timer</artifactId>
  <version>1.0</version>
</project>

```

El groupId es como el directorio. El artifact es el jar que estoy programando. Por ejemplo nosotros antes programamos un timer, no tiene nada que ver con el nombre de la clase MyTimer. La version es la versión del programa que estamos haciendo.

### Versión de Java

Si queremos estar seguros de que estamos compilando con Java 11 (sobre todo cuando usamos generics donde queremos garantizar cierta versión), agregamos:

Opción 1:

```

<project ... >
...
<properties>
  <maven.compiler.source>11</maven.compiler.source>
  <maven.compiler.target>11</maven.compiler.target>
  <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
</properties>
...
</project>

```

Hay otra forma (En la guía 1).

### Dependencias

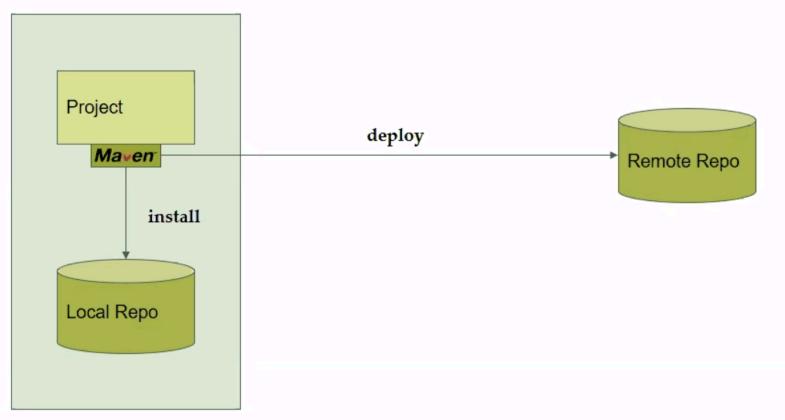
En el pom.xml se declaran, además, las dependencias a utilizar. Maven busca primero la dependencia en nuestro repositorio local. En caso de no encontrarla la descarga del repositorio correspondiente al repositorio local de nuestra computadora. Ese repositorio local típicamente se encuentra en \$HOME/.m2

Ejemplo:

- C:\Users\lgomez\.m2
- /Users/jabu/.m2

- /home/luis/.m2

Repositorios:



Nosotros vamos a trabajar en repos locales.

En Maven, cuando creamos un nuevo proyecto y queremos incluir dependencias, lo hacemos de la siguiente manera, tomando como ejemplo la librería de Joda Time:

```

<dependencies>
    <!-- https://mvnrepository.com/artifact/joda-time/joda-time -->
    <dependency>
        <groupId>joda-time</groupId>
        <artifactId>joda-time</artifactId>
        <version>2.10.10</version>
    </dependency>
</dependencies>
  
```

Una vez creado el timer, vamos a hacer install y vamos a tener nuestro .jar. Si ahora creamos un nuevo proyecto que queremos que use la clase MyTimer, lo hacemos de la misma manera:

```

<dependencies>
    <dependency>
        <groupId>ar.edu.itba.eda</groupId>
        <artifactId>JodaTime</artifactId>
        <version>1.0</version>
    </dependency>
</dependencies>
  
```

Pero esto solo va a funcionar localmente, por lo que si queremos enviarlo y que lo incluya debemos usar:

```

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                    <configuration>
                        <descriptorRefs>
                            <descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

Ahora, lo que debemos hacer es configurar la clase principal, porque sino no vamos a poder ejecutar. Para eso, agregamos dentro de configuration:

```

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-assembly-plugin</artifactId>
            <executions>
                <execution>
                    <phase>package</phase>
                    <goals>
                        <goal>single</goal>
                    </goals>
                    <configuration>
                        <descriptorRefs>
                            <descriptorRef>jar-with-dependencies</descriptorRef>
                        </descriptorRefs>
                        <archive>
                            <manifest>
                                <mainClass>main.Proof</mainClass>
                            </manifest>
                        </archive>
                    </configuration>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>

```

## Test Driven Development (TDD)

Es una metodología que comienza por los test y luego pasa a la implementación. Permite enfocarse en la definición de la interfaz y del comportamiento y no en los detalles internos de implementación. Ve al sistema que se va a desarrollar como una caja negra ya que todavía no existe!

## Unit Testing

Consiste en testear pequeñas unidades del código, normalmente una clase o de una función aisladas. Corren automáticamente y pueden ser ejecutados cada vez que se hacen cambios para comprobar que la funcionalidad anterior siga funcionando correctamente.

- Automático
- Verifican un único caso por test

- Repetible
- Independientes de otros tests o de condiciones externas
- Mantenible y Documentado ( comentado )
- Ejecuta en muy poco tiempo ( muy deseable )

¿Qué vamos a testear en un unit Test?

- En el caso de un método o función: Casos Típicos, Casos de Borde, Casos de Error y Casos de Excepción.
- En el caso de una clase: Secuencias de llamadas válidas, Secuencias de llamadas inválidas, Chequeo de invariantes.

## **JUnit 5**

Es un framework para realizar casos de prueba en aplicaciones Java. Se pueden comparar resultados de las invocaciones de métodos con los valores esperados, o verificar si una excepción fue lanzada o no. Un caso de prueba es abortado ni bien falla alguna verificación o se lanza una excepción no esperada.

### Comparación de Resultados

El test unitario más simple consiste en comparar el resultado obtenido con el resultado esperado.

Para ello, se pueden utilizar los siguientes métodos estáticos:

```
Assertions.assertEquals(valorEsperado, valorObtenido)
Assertions.assertTrue(valorObtenido)
...
```

Si un método lanza una excepción, el mismo se considera que falló. Para aquellos casos en que se espera que se lance esta excepción, se indica de la siguiente manera:

```
Assertions.assertThrows(RuntimeException.class, () -> ...);
```

Ejemplo:

```

import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.*;

class TimerFromScratchTest {
    @Test
    @DisplayName("Probar si el lapso de tiempo es correcto.")
    void getDurationTest() {
        TimerFromScratch timer = new TimerFromScratch();
        long expected = 2000; //ms
        long result = timer.getDuration( start: 0, stop: 2000);
        assertEquals(expected, result);
    }

}

```

Puede ser que debamos poner Assertion.assertEquals(...).

Otro ejemplo:

```

@Test
@DisplayName("Probar excepcion lanzado por: stop < start")
void getDurationExceptionTest() {
    TimerFromScratch timer = new TimerFromScratch();
    assertThrows(RuntimeException.class,
                 ()->timer.getDuration( start: 2000, stop: 1000));
}

```

## Ambiente de Prueba

Frecuentemente se desea tener un ambiente de prueba prefijado, por ejemplo con ciertas variables inicializadas. Para evitar repetir este código de inicialización en cada uno de los tests unitarios (en cada uno de los métodos @Test) se cuentan con varias anotaciones útiles:

- **@BeforeAll**: Se ejecutará antes de todos los casos de prueba de la clase
- **@BeforeEach**: Se ejecutará antes de cada **@Test**
- **@AfterEach**: Se ejecutará después de cada **@Test**
- **@AfterAll**: Se ejecutará después de todos los casos de prueba de la clase
- y otras más

Se usan de la siguiente forma:

```

@BeforeAll
static void initAll() {
    System.out.println("Empiezan los tests");
}

@BeforeEach
void init() {
    System.out.println("Empieza un test");
}

@AfterEach
void tearDown() {
    System.out.println("Termina un test");
}

@AfterAll
static void tearDownAll() {
    System.out.println("Terminaron todos los tests");
}

```

Ejemplo:

```

class TimerFromScratchTest {
    3 usages
    TimerFromScratch timer;
    @BeforeEach
    void instanceTimer(){
        this.timer = new TimerFromScratch();
    }

    @Test
    @DisplayName("Probar si el lapso de tiempo es correcto.")
    void getDurationTest(){
        //TimerFromScratch timer = new TimerFromScratch();
        long expected = 2000; //ms
        long result = timer.getDuration( start: 0, stop: 2000);
        assertEquals(expected,result);
    }
}

```

Antes de cada test, creamos un nuevo timer para hacer las pruebas.

Para que todo esto funcione, tenemos que agregar la dependencia al POM usando:

```

<dependencies>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-engine</artifactId>
        <version>5.8.0-M1</version>
        <scope>test</scope>
    </dependency>
</dependencies>

<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-surefire-plugin</artifactId>
    <version>2.22.2</version>
</plugin>

```

## Análisis de Algoritmos

Vamos a medir teórica y empíricamente.

### a) Empíricamente:

Tenemos dos algoritmos, algoA y algoB que van a encontrar el máximo elemento de un vector. Tomamos el caso de MyTimer y agregamos en el main un array descendiente para recorrer con los algoritmos.

```
public class AlgoA {  
    public static int max(int[] array) {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        int candidate = array[0];  
        for (int rec = 1; rec < array.length; rec++)  
            if (candidate < array[rec])  
                candidate = array[rec];  
  
        return candidate;  
    }  
}  
  
import java.util.Arrays;  
  
public class AlgoB {  
    public static int max (int[] array) {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        Arrays.sort(array); // ordena ascendente  
  
        return array[array.length - 1];  
    }  
}
```

Podemos comparar los tiempos y ver cuál va a ser el algoritmo más rápido.

La idea de usar la métrica "tiempo de ejecución calculada empíricamente" para rankear algoritmos tiene varias dificultades.

- Como los algoritmos tardan diferente dependiendo de los datos con los que opera (input), para probar realmente con datos grandes, debería generar estos valores. Podría tardar días chequear los tiempos en grandes inputs.
- Si mi algoA lo ejecutó en mi compu y tarda X ms, y otro propone un algoB que ejecuta en su compu y tarda X/2 ms, este ranking puede ser engañoso.

Por lo tanto, el empírico va a ser un **complemento** al teórico.

### b) Teóricamente

Consiste en usar una descripción de alto nivel del algoritmo para evaluar su eficiencia independientemente del hardware y software donde execute. Se lo describe con una "expresión (fórmula)."

Idea básica: "contar la cantidad de operaciones primitivas" que ejecuta el algoritmo, que no importa cuánto tardan (unidades de tiempo genéricas). Dichas operaciones

son las más costosas en ejecutar en cualquier computadora: comparaciones, operaciones (aritméticas), transferencia de control desde una in hacia otra. (las asignaciones llevan tiempo despreciable, se pueden ignorar). Como el tamaño del input afecta la performance del algoritmo, entonces la "fórmula" se realiza contando la cantidad de operaciones primitivas que se realiza expresada en términos del tamaño de entrada.

Tomamos algoA del ejercicio anterior:

```
public class AlgoA {

    public static int max (int[] array) {
        if (array == null || array.length == 0)
            throw new RuntimeException("Empty array");

        int candidate= array[0];
        for (int rec= 1; rec < array.length;  rec++)
            if ( candidate < array[rec] )
                candidate= array[rec];

        return candidate;
    }

}
```

3 operaciones fijas

Luego: 1 comparación+1 suma +  
1 comparación. Esto se hace N-1  
veces

Las 3 operaciones fijas son las dos comparaciones y el or (||).

Vemos que la primer comparación es rec < length, luego sumamos 1 a rec y la última comparación es candidate < array. Todo esto lo hacemos n-1 veces (porque cuando rec >= length no entra).

Entonces la cuenta nos queda:

$$T(\text{algoA}) = 3 + 3 * (N - 1) = 3 * N$$

Ahora tomemos algoB:

```
public class AlgoB {

    public static int max (int[] array) {
        if (array == null || array.length == 0)
            throw new RuntimeException("Empty array");

        // ordena ascendente
        Arrays.sort(array);

        return array[array.length-1];
    }

}
```

3 operaciones fijas

Tenemos nuevamente las mismas 3 operaciones y ahora vamos a tener que buscar cómo funciona el algoritmo de sort. En la documentación de java se indica que es  $O(n \log n)$ , por lo que nos queda:

$$T(\text{algoB}) = 4 + N * \ln(N)$$

Si comparamos ambos resultados:

$$T(\text{algoA}) = 3 * N$$

$$T(\text{algoB}) = 4 + N * \ln(N)$$

La descripción que buscamos para comparar algoritmos es una “asíntota” (cota) expresada en términos de  $N$  que nos permita caracterizar la “tasa de crecimiento u orden de crecimiento de la fórmula”

Definición de **Comportamiento asintótico superior** u  $O$  grande (asymptotic upper bound running time u  $O$ -notation) de un algoritmo.

Sean  $T(N)$  y  $g(N)$  funciones con  $N > 0$ .

Se dice que  $T(n)$  es  $O(g(N))$  si  $\exists c > 0$  (constante no dependiente de  $N$ ) y  $\exists n_0 > 0$  tal que  $\forall N \geq n_0$  se cumple que  $0 \leq T(N) \leq c * g(N)$ .

En algoA, obtuvimos  $T(N) = 3N$ , pero para este caso digamos que  $T(N) = 1 + 3N$ .

Vemos que  $g(N)$  puede ser  $N$ . Entonces:

$$0 \leq 1 + 3N \leq c * N$$

Pero, ¿cuánto vale  $c$ ?

Si  $N$  es 2 entonces  $c \geq 3.5$

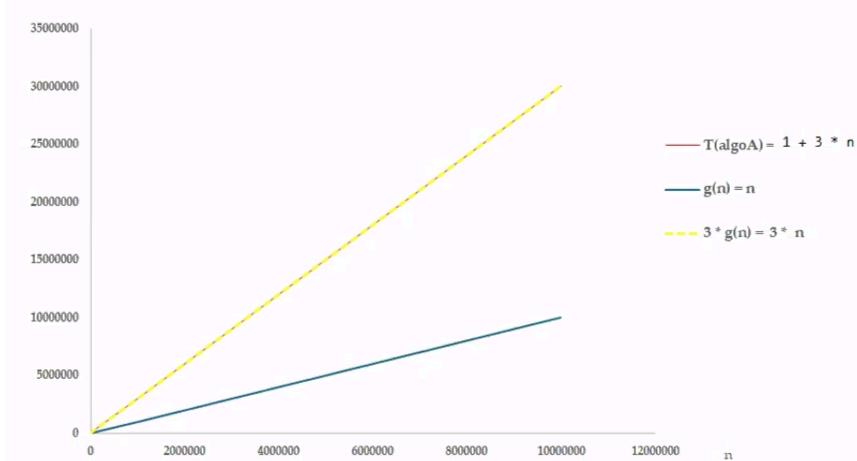
Si  $N$  es 5 entonces  $c \geq 3.2$ , mejor la cota.

Si  $N$  es 500,  $c \geq 3.002$ , mejor aun.

Yo quiero  $N \rightarrow \infty$ , entonces  $c \geq 3$ . Así, el orden de algoA es  $O(N)$ .

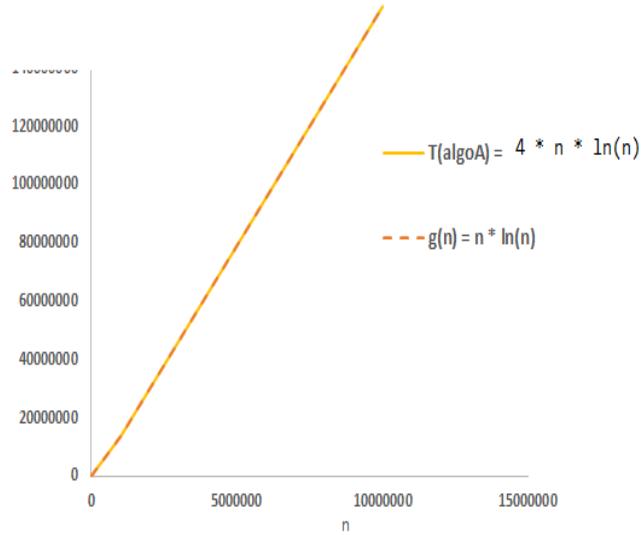
Encontré caracterizar una constante  $c$ , que es 3, para el cual se verifica la fórmula para  $N \rightarrow \infty$  ( $\forall N \geq n_0$ ). El algoritmo es  $O(N)$ .

Si lo miramos gráficamente:



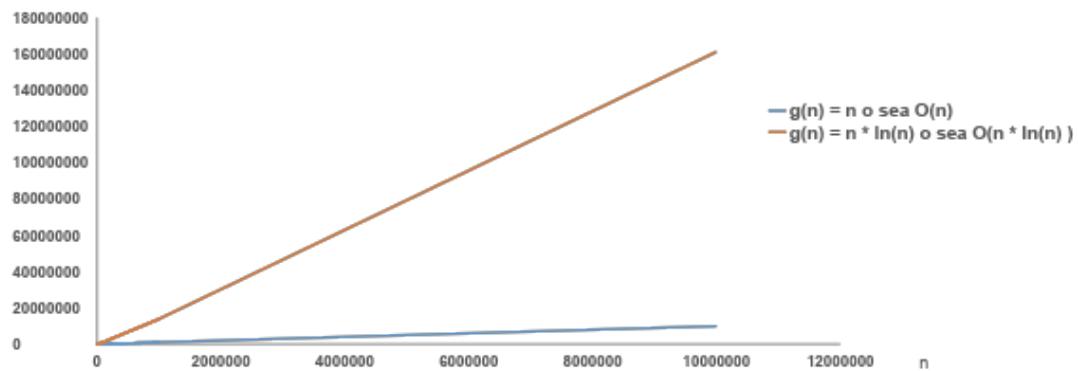
Siempre la cota que busquemos tiene que ser la que esté lo más pegado que se pueda.

Tomemos ahora el algoB. Ya dijimos que tiene complejidad  $N \cdot \log(N)$ , por lo que si graficamos obtenemos:

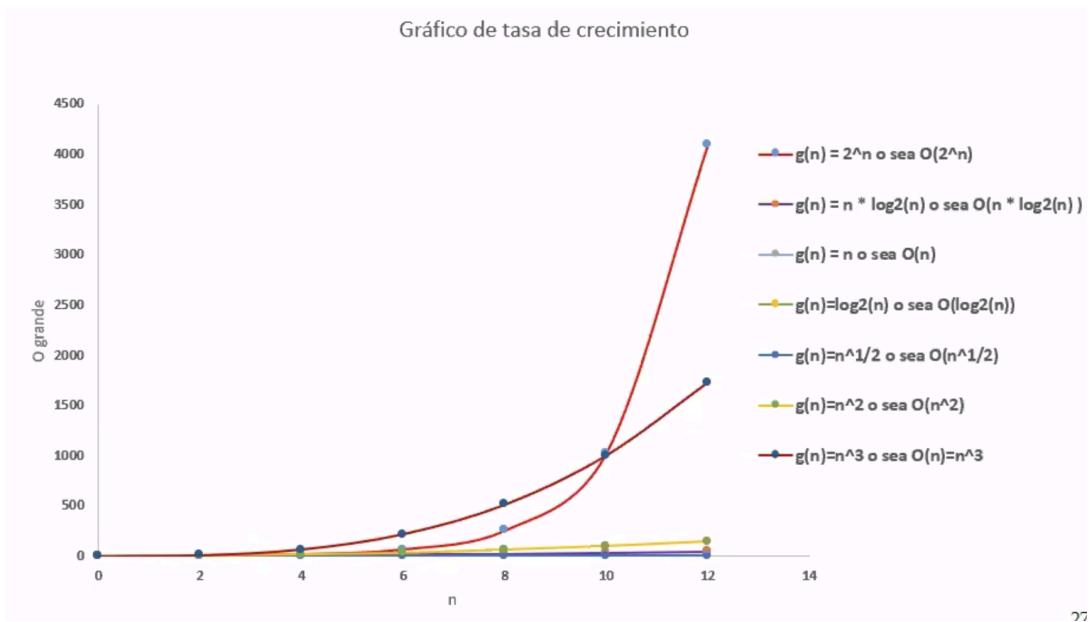


Entonces, comparando ambos algoritmos y graficando obtenemos que:

Gráfico de tasa de crecimiento



Vemos una clara diferencia entre ambos algoritmos. Comparemos ahora los distintos órdenes:



Notemos que podemos establecer un orden según las tendencias, y ya vamos a saber qué algoritmos son más rápidos que otros según su orden:  
 $\dots < O(\log_2(N)) < O(VN) < O(N) < O(N \cdot \log_2(N)) < O(N^2) < O(N^3) < O(2^N) < O(N!)$   
 $< \dots$

### Importante:

Para realizar el cálculo de la complejidad temporal de un algoritmo, no basta con analizar el "tamaño de los datos de entrada". La performance del algoritmo también puede depender de cómo vienen los datos. Esto lo podemos ver si un arreglo nos viene casi ordenado y tenemos un algoritmo que ordena, va a ser mucho más rápido. Se puede hacer un análisis del "mejor caso", "promedio" o "peor caso" de input. Nosotros, salvo que digamos lo contrario, vamos a realizar siempre el análisis del peor caso.

Veamos los siguientes casos:

$$T(N) = 6 * N + 2 \Rightarrow O(N)$$

$$T(N) = 2 * N^3 + 100 * N^2 \Rightarrow O(N^3)$$

$$T(N) = 2^N + N^3 \Rightarrow O(2^N)$$

$$T(N) = N + 6 * \log_{10} N \Rightarrow O(N)$$

En el último caso, si no estuviera el  $N +$ , la complejidad sería  $\log(N)$  sin importar si es en base 2, 10, etc. Es logarítmico y listo.

## Complejidad Espacial

Anteriormente medimos el tiempo de ejecución, pero también necesitamos medir el espacio que puede utilizar un algoritmo. Esto se va a medir teóricamente.

En el caso anterior, el arreglo no forma parte de la medición porque ya se lo pasan al algoritmo. Si en cambio creara otro para no modificarlo, ahí si va a utilizar memoria el algoritmo.

Para que un algoritmo ejecute algo en el procesador, los datos deben estar en RAM. O sea, puede ser que los datos residan en disco, pero el procesador no va directo a disco. Va a disco, lo carga en RAM y ejecuta. O sea, que ese es el espacio que tendremos en cuenta.

Por lo tanto, la **complejidad espacial** consiste en usar una descripción de alto nivel del algoritmo para evaluar cuánta espacio extra precisa para sus variables (parámetros formales, invocaciones a otras funciones, variables locales). Se lo describe con una “expresión (fórmula) en términos del tamaño de entrada del problema.

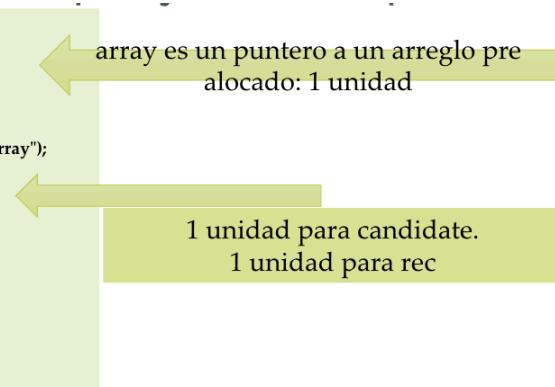
La idea es la misma, buscan una cota (O grande) para el espacio RAM (stack y heap). Busca independizarse de software y hardware, es decir no va tener en cuenta si una computadora es de 32 bits o 64 bits, etc. Se expresa a través de “N”.

Para caracterizar el espacio en nuestros algoritmos que escribimos en Java, tenemos que tener en cuenta el espacio que se aloca para:

- Heap => cada vez que hacemos “new” reservamos lugar en esta zona. El GC es el proceso que libera esa zona cuando detecta que una zona ya no es más referenciada por ninguna variable.
- Stack => cada vez que se invoca un método se genera un stack frame para el mismo, conteniendo: los parámetros formales con sus valores, variables auxiliares declaradas dentro del mismo y el lugar de la próxima sentencia que falta a ejecutar (así, cuando se retorne, continúa la ejecución). O sea, no resulta gratis “invocar funciones”, se generan stack frames...

Tomando los casos anteriores, comenzando por algoA:

```
public class algoA {  
    public static int max (int[] array)  
    {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        int candidate= array[0];  
        for (int rec= 1; rec < array.length-1; rec++)  
            if ( candidate < array[rec] )  
                candidate= array[rec];  
  
        return candidate;  
    }  
}
```



$S(\text{algoA}) = 3$ , osea que el espacio utilizado es de  $O(1)$ .

Veamos ahora algoB:

```
public class algoB {  
    public static int max (int[] array) {  
        if (array == null || array.length == 0)  
            throw new RuntimeException("Empty array");  
  
        Arrays.sort(array); // ordena ascendentemente  
  
        return array[array.length-1];  
    }  
}
```

array es un puntero a un arreglo pre  
alocado: 1 unidad

????

Miren la implementación:  
[https://github.com/frohoff/jdk8u-dev-jdk/  
blob/master/src/share/classes/java/util/Du  
alPivotQuicksort.java](https://github.com/frohoff/jdk8u-dev-jdk/blob/master/src/share/classes/java/util/Du alPivotQuicksort.java)

Debemos chequear la implementación del código del método sort para determinar las unidades que utiliza. Vemos que es muy difícil de entender el código, pero las dos implementaciones conocidas para sort es de manera lineal o logarítmica, pero cualquiera sea, algoA es mejor.

Como conclusión, algoA es tiene menor complejidad tanto temporal como espacial que algoB, por lo que va a ser mejor. Es muy común que se contrapongan dos algoritmos que comparamos, es decir que uno sea más rápido que el otro, pero que ese tenga menor complejidad espacial.

## Unidad 2 - Algoritmos para Búsqueda en Texto

### Algoritmos para Textos

Veamos algunos ejemplos:

- Política: este discurso ya lo escuché... Está repitiendo lo que dijo otro político.
- Alumnos/Autores: esta respuesta coincide con la de este otro alumno => se copió.
- Plagio en música

Estos son casos de Coincidencia de textos **completa o parcial** (exact string matching).

- Política: este discurso se parece a uno que escuché...
- Biología: ADN
- Tipos

Estos otros son casos de coincidencia de textos **aproximada**.

### Definiciones

**Alfabeto**  $\Sigma$ : conjunto de símbolos o caracteres.

Dado un alfabeto  $\Sigma$  y  $K \geq 0 \in N$ , un **string**  $S$  es un elemento  $\in \Sigma^K$ .

Para  $S \in \Sigma^K$ , se dice que  $|S|$  es  $\Sigma$ , y denota su longitud. Si  $K = 0$ ,  $S$  se dice que es el **string vacío**, se lo denota con  $\lambda$ .

¿Por qué se lo denota con  $\lambda$ ?

Para evitar los problemas que tienen los compiladores, ya que un meta-símbolo no debería ser al mismo tiempo parte del alfabeto (regla básica). Los lenguajes de programación violan estas reglas y ahí surgen los problemas.

Ejemplo: En Java, el carácter comilla doble delimita el comienzo y fin del string. No son parte de las operaciones. El string “EDA” tiene 3 símbolos, no 5. Hasta ahí parece sencillo: quitemos los 2 caracteres externos.

Pero, si el compilador encontrara “hola”que” daría error, porque no sabe dónde termina el string: ¿Es “hola” y lo que sigue está mal? ¿Debería ser “hola”que”?

Para evitar ambigüedades obliga a escapar al carácter comillas dobles cuando participa del string. Se lo escapa con la barra invertida: \”. Pero yo lo veo como un doble carácter, aunque representa uno solo.

Ejemplo: “hola\”que” estaría queriendo representar al string hola”que, que es de 8 caracteres y no de 9 caracteres.

Pero la ambigüedad no está solucionada. Otra vez, el símbolo barra invertida es meta-símbolo y parte del alfabeto .

Ejemplo: “\hola”que” estaría queriendo representar al string \hola”que, de 9 caracteres y no de 11.

Otras definiciones:

**Conjunto de todos los strings posibles sobre cierto alfabeto**

Dado un alfabeto  $\Sigma$ ,  $\Sigma^* = \bigcup \Sigma^k$  con  $k_{\geq 0} \in \mathbb{N}$

**Concatenación de Strings**

Dado un alfabeto  $\Sigma$ , y  $u \in \Sigma^*$ ,  $w \in \Sigma^*$ .

Se llama concatenación al string definido como  $uw$  (un elemento a continuación del otro, sin símbolos extra entre ellos).

**Prefijos, Sufijos y Substrings**

Dados un alfabeto  $\Sigma$  y los strings  $x \in \Sigma^*$ ,  $w \in \Sigma^*$ ,  $z \in \Sigma^*$ . Sea  $p=xwz$ .

Se dice que  $x$  es un prefijo de  $p$ . Se dice que  $w$  es un substring de  $p$ . Se dice que  $z$  es un sufijo de  $p$ .

**Bordes**

Dados un alfabeto  $\Sigma$  y los strings  $x \in \Sigma^*$ ,  $w \in \Sigma^*$ ,  $z \in \Sigma^*$ . Si  $p = wx = zw$  donde  $|x| = |z|$ , se dice que  $w$  es un border de  $p$ .

Ejemplos:

$$\Sigma = \{0, 1, 2, 3, 4, 5\}$$

Sea  $s = "01230"$

¿Cuáles son los prefijos de  $s$ ?

Rta: "", "0", "01", "012", "0123", s

¿Cuáles son los sufijos de  $s$ ?

Rta: "", "0", "30", "230", "1230", s

¿Cuáles son los borders de  $s$ ?

Rta: "", s, "0". Como mínimo hay 2 borders: "" y s

¿Cuáles son los substrings de  $s$ ?

Rta: "", "0", "01", "012", "0123", s, "30", "230", "1230", "1", "12", "123", "2", "23", "3", "30", "0"

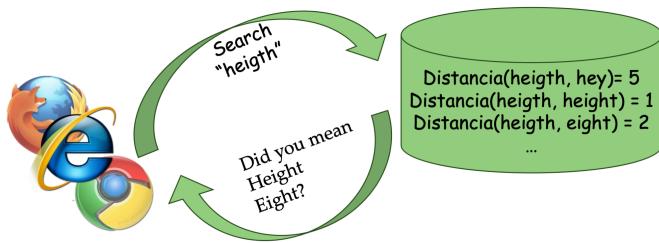
Para obtener los bordes, básicamente calculo los prefijos y los sufijos y obtengo la intersección.

## Data Quality - Matching

Los buscadores usan alguna estrategia, en el caso de que la búsqueda lanzada no sea reconocida en el “corpus” que poseen sobre búsquedas y documentos indexados. Es decir, si buscamos una palabra con errores, debe poder encontrarlo de igual manera. Vamos a tener que manejar “qué tanto se parece”.

Las estrategias son:

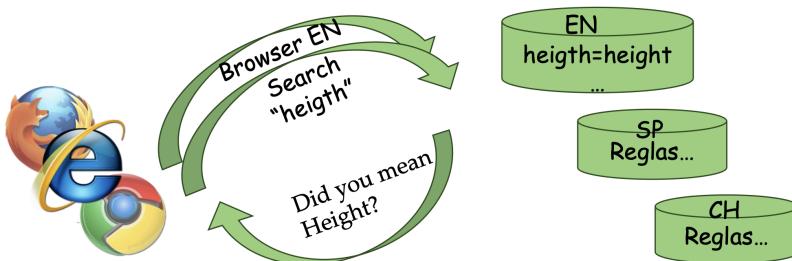
- Buscar las palabras y si las palabras no están en el corpus de los documentos indexados, encontrar las que mayor similitud posean y sugerirlas.



Vemos que devuelve las que más se parecen a nuestro ingreso “heighth”.

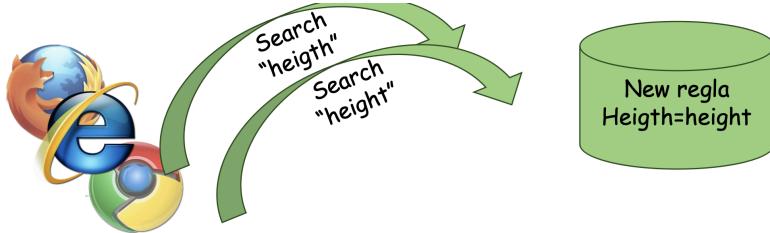
- Tomar el idioma que tiene configurado el browser para saber en qué corpus buscar las palabras usadas. Hay reglas conocidas por el idioma en cuestión. Ejemplo, en inglés HT con TH: height vs. width, length.

En las comunicaciones entre los browsers y los servidores, siempre hay intercambio de información, como la IP, el momento de la búsqueda, el idioma local de la máquina, etc. La aplicación servidora puede tomar toda esta información o no, depende de cada una.



Vemos que la información que pasa el browser es, además de la query, el idioma.

- Sabiendo que los usuarios buscan palabras y cuando fue un error de tipeo/ortografía/etc., no cliquean nada del resultado e intentan realizar inmediatamente la búsqueda arreglada, almacenan esas búsquedas erróneas con la que arrojó resultados navegados. Es decir, hay un MATCHING entre errores viejos y soluciones que los mismos usuarios hicieron. Si esos errores son frecuentes, tendrán solución rápida.



Un ejemplo de estrategia en sitios de compra por internet, es que si un producto no se encuentra (en la categoría esperada), entonces el usuario intenta usar el botón de búsqueda. Pero si esta no da coincidencia => el usuario abandona el sitio y va a otro (en no más de 2 intentos).

## Reglas

Mínimas reglas que deberían aplicarse

- **Sacar blancos del comienzo y final (trim).** Pero no es suficiente. Si la palabra es compuesta habría que sacar blancos internos. Ej: ‘ yogurt beible ’
- **Buscar pasando todo a mayúscula o minúscula.** Ej: YogUrt= YOGURT
- **Si se conocen abreviaturas, usarlas.** Ej: BA por Buenos Aires
- **Los símbolos de puntuación, eliminarlos.** Ej: Bs. As. por Bs As
- **Si se conocen sinónimos, usarlos.** Ej: computadora por ordenador, teléfono celular por teléfono móvil. Inclusive entre diferentes idiomas.

Reglas específicas que deberían aplicarse para los *tipos de datos*:

- Fechas y sus formatos  
Ej: 12/10/2016 = 12 Oct 2016 = 2016-10-12
- La hora y sus formatos  
Ej: 15:30 = 3:30 PM
- Números  
Ej: 12.300.140 = 12300140
- Números Decimales  
Ej: 12,1 = 12.1
- String correspondientes a nombre y apellidos  
Ej: JohnPeterDoe = JohnP.Doe = J.D.Doe = Doe,JohnPeter = Doe,JohnP. = Doe,J.P.

## Soundex

Es un algoritmo fonético, es decir codifica a una palabra según “sueña”. Intenta solucionar problemas de pronunciación. Fue creado para el alfabeto inglés (o sea, codifica las 26 letras del mismo). Existen otras adaptaciones como Soundex\_FR para idioma francés. Aquellas palabras que “suenan igual”, aunque no se escriban igual, deben ser codificadas de la misma manera.

26 Letras	Pesos fonéticos
A, E, I, O, U, Y, W, H	0 -- no se codifica
B, F, P, V	1
C, G, J, K, Q, S, X, Z	2
D, T	3
L	4
M, N	5
R	6

Soundex siempre devuelve una código OUT de exactamente 4 caracteres, formados por: primero una letra y luego 3 dígitos (pesos fonéticos). Si hace falta, para completar el código de 4 caracteres, se completan con 0s (ceros) al final.

La primera letra es siempre la actual del input, y se llevan el peso fonético de esa letra.

Veamos el paso por paso:

Paso 1 (opcional): Pasar a mayúsculas y dejar sólo las letras (dígitos, símbolos de puntuación, espacios, etc. se eliminan).

Paso 2: Colocar OUT[0]=IN[0].

Paso 3: Se calcula vble. last como el peso fonético de IN[0]

Paso 4: Para cada letra iter siguiente en IN y hasta completar 3 dígitos o terminar de procesar IN, hacer

4.1) calcular vble current con peso fonético de iter. Si es diferente a 0 y no coincide con last, appendear current en OUT.

4.2) independiente del paso anterior, tapar last = current.

Paso 5: si hace falta completar con '0's y devolver OUT.

Si miramos el código:

```

char IN[] = new String("...").toCharArray();
char OUT[] = {'0', '0', '0', '0'};
OUT[0] = IN[0];
int count = 1;
char current, last = getMapping(IN[0]);
for(int i = 1; i < IN.length && count < 4; i++, last = current)
{
    char iter = IN[i];
    current = getMapping(iter);
    if (current != '0' && current != last)
        OUT[count++] = current;
}
return new String(OUT);

```

Soundex codifica, pero ¿Cómo usarlo para comparar la proximidad entre palabras? Soundex NO es una métrica. Hay que definir cómo obtener una métrica a partir de soundex.

### Similitud para Soundex

Es la proporción de caracteres coincidentes entre los encodings respecto a la longitud del encoding.

Ej: Soundex ("threshold") = T624

Soundex( "hold" ) = H430

Soundex( "zresjoulding" ) = Z624

Soundex( "phone" ) = P500

Soundex ( "foun" ) = F500

Soundex ("threshold", "hold" ) = 0

Soundex("threshold", "zresjoulding" ) =  $\frac{3}{4} = 0.75$

Soundex("phone", "foun" ) =  $3/4 = 0.75$

### Metaphone

Este encoding genera símbolos de longitud arbitraria.

Ej: Metaphone ( "threshold" ) = 0RXLT

Metaphone( "hold" ) = HLT

Metaphone( "zresjoulding" ) = SRSJLTNK

Metaphone( "phone" ) = FN

Metaphone ( "foun" ) = FN

SimilitudMetaphone ("phone", "fown" ) = 1

(mejoró!!!)

La similitud entre 2 textos puede pensarse como la proporción de caracteres coincidentes entre los encodings respecto de la máxima longitud de los encodings obtenidos, ya que son de longitud variable.

### Algoritmo de Levenshtein (Levenshtein Distance)

Es un algoritmo que calcula la **MÍNIMA** cantidad de operaciones necesarias para transformar un string en otro. Las operaciones válidas son: insertar, borrar o sustituir un carácter.

Aquellos strings que son iguales, deben tener distancia 0 porque no hace falta transformar uno en otro.

Ej: Levenshtein('big data', 'bigdata') = 

IMPORTANTE: ahora estamos hablando de **distanzia**, NO similitud como con los dos algoritmos anteriores. Cuando la distancia es 0, la similitud es grande, y cuando la distancia es grande, la similitud es pequeña.

En el caso de big data, vemos que es lo mismo sacarle el espacio al primero o agregarlo al segundo, por lo que es simétrico.

Notemos que hay otras maneras de llegar de “big data” a “bigdata”, como por ejemplo:

'big data' => BBBBIII ó sea 7  
'big data' => BSSS o sea 4  
'big data' => ---SSSSD o sea 5  
Etc, etc, etc.

Con B = borrar, I = insertar, S= sustituir. El guión significa skip.

Como Levenshtein es una métrica de distancia, cumple con propiedades:

- Simetría:  $\text{Levenshtein}(\text{str1}, \text{str2}) = \text{Levenshtein}(\text{str2}, \text{str1})$
- Desigualdad:  $\text{Levenshtein}(\text{str1}, \text{str2}) + \text{Levenshtein}(\text{str2}, \text{str3}) \geq \text{Levenshtein}(\text{str1}, \text{str3})$

El problema es que una implementación ensayo & error es muy ineficiente. Por lo que vamos a usar otra técnica de programación.

### Programación Dinámica

Es una técnica que consiste en rehusar valores previamente calculados para no tener que recalcularlos repetidamente. Los valores deben almacenarse en una estructura de datos (vector, matriz, etc) con el objetivo de “buscarlos” (lookup - orden 1) y no calcularlos nuevamente cuando se los precise.

### String Matching - Levenshtein Distance

Vamos a tener un cuadro de doble entrada, que no importa qué string está dónde por la simetría. Vemos qué representa cada celda:

	♥	B	I	G	D	A	T	A
I	♥							ig
B								
I								
G								
D								
A								
T								
A								

La celda representa Levenshtein('BIG', 'BI')

Entonces, si vamos completando la tabla:

	♥	B	I	G	D	A	T	A	
♥	0	1	2	3	4	5	6	7	8
B	1								
I	2								
G	3								
D	4								
A	5								
T	6								
A	7								

Entonces  
 $\text{Levenshtein}("", "") = 0$   
 $\text{Levenshtein}("B", "") = 1$   
 $\text{Levenshtein}("BI", "") = 2$   
 $\text{Levenshtein}("BIG", "") = 3$

Entonces  
 $\text{Levenshtein}("", "B") = 1$   
 $\text{Levenshtein}("", "BI") = 2$   
 $\text{Levenshtein}("", "BIG") = 3$   
 $\text{Levenshtein}("", "BIGD") = 4$

En este caso, completamos la distancia del string vacío a cada carácter acumulado.  
Tomemos ahora el caso de transformar BIG en BI.

	♥	B	I	G	D	A	T	A	
♥	0	1	2	3	4	5	6	7	8
B	1								
I	2								
G	3								
D	4								
A	5								
T	6								
A	7								

La celda representa Levenshtein('BIG', 'BI')

$\text{Levenshtein}(\text{BIG}', \text{BI}') =$   
 $\min (\text{Levenshtein}(\text{BI}', \text{B}') + \text{G}' == \text{I}' ? 0 : 1,$   
 $\text{Levenshtein}(\text{BIG}', \text{B}') + 1,$   
 $\text{Levenshtein}(\text{BI}', \text{BI}') + 1,$   
 $)$

Notemos que estamos avanzando de a una letra en vertical y en horizontal, por lo que tenemos dos candidatos de valor, que es sumarle 1 al de inmediatamente arriba y al de la izquierda. Esto para las operaciones de borrado o agregado, nos falta sustitución, que nos va a dar otro candidato. Vamos a comparar con el de la diagonal (superior izquierda). Si coinciden, le sumamos 0 y sino 1. Luego, tomamos el mínimo de los 3 candidatos y completamos la tabla.

	♥	B	I	G		D	A	T	A
♥	0	1	2	3	4	5	6	7	8
B	1								
I	2								
G	3								
D	4								
A									
T									
A									

```

Levenshtein(aw,xz)=
Min ( Levenshtein(a, x) + w==z?0:1,
      Levenshtein(aw, x) + 1,
      Levenshtein(a, xz) + 1
    )
  
```

Vamos completando y nos queda:

	♥	B	I	G		D	A	T	A
♥	0	1	2	3	4	5	6	7	8
B	1	0	1	2	3	4	5	6	7
I	2	1	0	1	2	3	4	5	6
G	3	2	1	0	1	2	3	4	5
D	4	3	2	1	1	1	2	3	4
A	5	4	3	2	2	2	1	2	3
T	6	5	4	3	3	3	2	1	2
A	7	6	5	4	4	4	3	2	1



Vemos que para la celda de BIG con BI que marcamos antes, tenemos:

Arriba:  $2 (+ 1) = 3$

Izquierda:  $0 (+ 1) = 1$

Diagonal:  $1 (+ 1 \text{ porque } B \neq G) = 2$

Como el mínimo es 1, completamos con 1.

Entonces, la distancia va a ser el número al final de la tabla (en este caso distancia 1).

Ejemplo: ¿Cuál es la distancia Levenshtein (“exkusa”, “ex-amigo”) ?

	-	e	x	-	a	m	i	g	o
-	0	1	2	3	4	5	6	7	8
e	1	0	1	2	3	4	5	6	7
x	2	1	0	1	2	3	4	5	6
k	3	2	1	1	2	3	4	5	6

u	4	3	3	2	2	3	4	5	6
s	5	4	4	3	3	3	4	5	6
a	6	5	5	4	4	4	4	5	6

Por lo tanto la distancia es 6.

El número obtenido mediante Levenshtein se puede **normalizar** para que el número obtenido esté entre 0 y 1, siendo 1 el valor de la coincidencia.

$$\text{LevenshteinNormalized}(\text{str1}, \text{str2}) = 1 - \frac{\text{Levenshtein}(\text{str1}, \text{str2})}{\max(\text{str1.len}, \text{str2.len})}$$

Existen variantes para este algoritmo, como por ejemplo Damerau-Levenshtein: las operaciones no son sólo borrado, inserción, y sustitución. También se agrega transposición.

Otras variantes no consideran que las operaciones valen todas igual. Alguna es más cara que otra y cambia la fórmula de distancia, entonces.

Para saber qué operaciones realizó según la tabla:

**IMPORTANTE:** La inserción y eliminación dependen de si estamos mirando el str1 o el str2.

- **Sustitución:** Si el valor en una celda es igual al valor de la celda diagonal superior izquierda más 1, entonces significa que se realizó una operación de sustitución. Esto indica que se cambió un carácter en la cadena original para hacer coincidir los caracteres entre las dos cadenas.
- **Inserción:** Si el valor en una celda es igual al valor de la celda a la izquierda más 1, entonces significa que se realizó una operación de inserción. Esto indica que se insertó un carácter en la cadena original para hacer coincidir los caracteres entre las dos cadenas.
- **Eliminación:** Si el valor en una celda es igual al valor de la celda superior más 1, entonces significa que se realizó una operación de eliminación. Esto indica que se eliminó un carácter en la cadena original para hacer coincidir los caracteres entre las dos cadenas.

### Q-Grams (N-Grams)

Es un algoritmo que consiste en generar los pedazos que componen un string. La distancia entre 2 strings estará dada por la cantidad de componentes que tengan en común. Si Q es 1 se generan componentes de longitud 1, si Q es 2 se generan bi-gramas y si Q es 3 se generan tri-gram (los de 3 son de los más usados).

Por ejemplo, para el string “JOHN” si se quiere generar hasta tri-gramas ( $Q \leq 3$ ), puede completarse al comienzo y al final con  $Q-1$  símbolos especiales (que no pertenezcan al alfabeto) y deslizando la ventana imaginaria de tamaño  $Q$ , se va generando los  $Q$ -gramas. Sea ‘##JOHN##’.

$$\text{Q-grams (John)} = \{ \textcolor{pink}{'J'}, \textcolor{pink}{'O'}, \textcolor{green}{'H'}, \textcolor{pink}{'N'}, \textcolor{pink}{'\#J'}, \textcolor{green}{'JO'}, \textcolor{green}{'OH'}, \textcolor{green}{'HN'}, \textcolor{green}{'N\#'}, \textcolor{orange}{'##J'}, \textcolor{orange}{'\#JO'}, \textcolor{orange}{'JOH'}, \textcolor{orange}{'OHN'}, \textcolor{orange}{'HN\#'}, \textcolor{orange}{'N##'} \}$$

Vemos en rosa los  $Q-1$ , en verde los  $Q-2$  y amarillo los  $Q-3$ . La cantidad de  $\#$  que agregamos es  $N-1$  adelante y atrás. En este caso agregamos 1 para los  $Q-2$  y 2 para los  $Q-3$ .

Comparamos con “Joe”:

$$\text{Q-grams (Joe)} = \{ \textcolor{pink}{'J'}, \textcolor{pink}{'O'}, \textcolor{green}{'E'}, \textcolor{pink}{'\#J'}, \textcolor{green}{'JO'}, \textcolor{green}{'OE'}, \textcolor{green}{'E\#'}, \textcolor{orange}{'##J'}, \textcolor{orange}{'\#JO'}, \textcolor{orange}{'JOE'}, \textcolor{orange}{'OE\#'}, \textcolor{orange}{'E##'} \}$$

Vemos que los Q-gramas en común son:

Entonces distancia(John, Joe) = 6

Ahora vamos a necesitar una fórmula para pasarlo a [0,1]. Algunas son:

$$\text{Q-Gram (str1, str2)} =$$

$$\frac{\#TG(str1) + \#TG(str2) - \#TGNoShared(str1, str2)}{\#TG(str1) + \#TG(str2)}$$

Siendo:

$\#TG(str1)$  la cantidad de trigramas que se generaron de str1.

$\#TG(str2)$  la cantidad de trigramas que se generaron de str2.

$\#TGNotShared(str1, str2)$  son la cantidad que no matchearon.

Otra implementación consiste en tomar nada más los  $Q-3$  y no los menores como hicimos antes. Cuando tengamos que resolver ejercicios, se nos va a especificar si es  $Q$  exactamente 3 o  $Q$  menor igual a 3.

$$\text{Q-grams (John)} = \{ \textcolor{orange}{'##J'}, \textcolor{orange}{'\#JO'}, \textcolor{orange}{'JOH'}, \textcolor{orange}{'OHN'}, \textcolor{orange}{'HN\#'}, \textcolor{orange}{'N##'} \}$$

$$\text{Q-grams (Joe)} = \{ \textcolor{orange}{'##J'}, \textcolor{orange}{'\#JO'}, \textcolor{orange}{'JOE'}, \textcolor{orange}{'OE\#'}, \textcolor{orange}{'E##'} \}$$

Vemos que los Q-gramas en común son ‘##J’ y ‘#JO’. Aplicando la fórmula anterior:

$$\text{Q-Gram(John, Joe)} = (6 + 5 - 7) / (6 + 5) = 0.3636$$

Notar que si tuvieran TODOS los Q-grams en común (matching exacto) tendríamos  $(N+N-0) / (N+N) = 1$ .

En conclusión, Q-Gram de 2 parámetros da “similitud” entre 2 strings: 1 es máxima similitud, 0 es nula.

Ejercicio:

Calcular para Q exactamente 2 la similitud entre *salesal* y *vale*.

Q-grams(*salesal*) = {'#s', 'sa', 'al', 'le', 'es', 'sa', 'al', 'l#'}

Q-grams(*vale*) = {'#v', 'va', 'al', 'le', 'e#'}

Q-gramas en común: {'al', 'le'}

Entonces:  $QGram(salesal, vale) = \frac{(8+5-9)}{(8+5)} = 0,3076$

Notar que en #TGNotShared no contamos dos o más veces los repetidos.

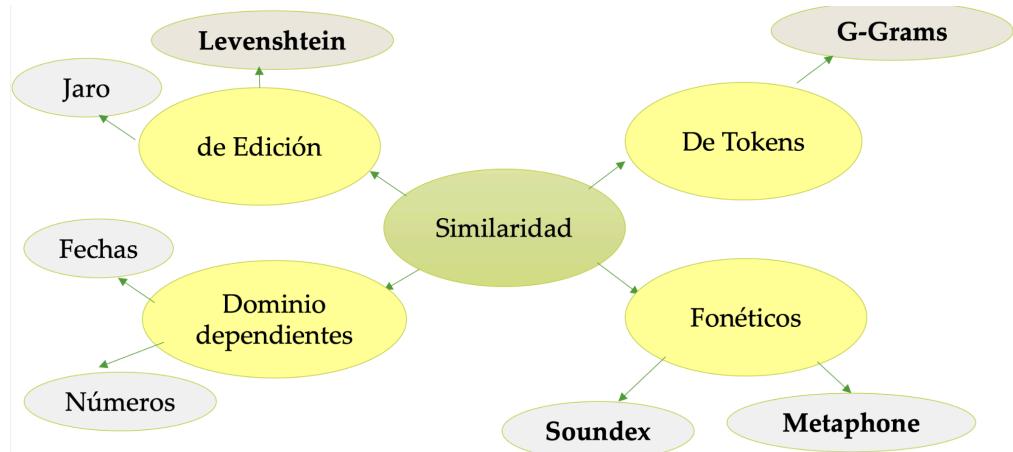
Si queremos programar este algoritmo en java, va a ser mucho más fácil hacerlo mediante HashMap en vez de utilizando ArrayList.

### Comparación de Algoritmos

Si quiero la similitud entre 2 frases (por lo menos 2 palabras), ¿en qué tipo de frases resulta mejor Qgrams que Levenshtein/Metaphone/Soundex?

Notemos que si tenemos las frases: “Análisis de Datos y Algoritmos” y “Análisis de Algoritmos y Datos”, si aplicamos Qgrams vamos a obtener un resultado muy bueno. Por otro lado, en Levenshtein vamos a tener la misma longitud pero varios cambios de letras para obtener la segunda frase, por lo que no va a dar tan bien. Además, fonéticamente suenan completamente diferentes.

En resumen, los algoritmos que vimos para el procesamiento de strings es:



Recordar que de Levenshtein vimos dos variantes. Jaro es otro algoritmo reconocido. Para implementaciones como TTS (Text To Speech) vamos a usar los fonéticos.

### String Matching - Bibliotecas Externas

Existen bibliotecas externas que implementan estos algoritmos que estuvimos trabajando. Un ejemplo son las de Apache Commons (<https://commons.apache.org/>) que tienen un manejo avanzado de strings.

Lo que buscamos con esto es obtener:

- El soundex("maven"), soundex("meibem") y la similitud de ambos, según soundex, que en este caso es 1.
- El soundex("threshold") y soundex("hold") y la similitud de ambos, según soundex, que en este caso es 0.
- El soundex("hold") y soundex("joul") y la similitud de ambos, según soundex, que en este caso es 0.5
- LevenshteinDistance("exkusa", "ex-amigo") y la similitud de ambos que es 1-6/8, o sea 0.25

Pero notemos que con esta biblioteca no pudimos trabajar con QGrams. Para incluir las bibliotecas, agregamos en el POM:

*Soundex:*

```
<!-- https://mvnrepository.com/artifact/commons-codec/commons-codec -->
<dependency>
    <groupId>commons-codec</groupId>
    <artifactId>commons-codec</artifactId>
    <version>1.15</version>
</dependency>
```

*Levenshtein:*

```
<!-- https://mvnrepository.com/artifact/org.apache.commons/commons-text -->
<dependency>
    <groupId>org.apache.commons</groupId>
    <artifactId>commons-text</artifactId>
    <version>1.9</version>
</dependency>
```

En este último, podemos utilizar la clase LevenshteinDetailedDistance para obtener detalles como cantidad de sustituciones necesarias, cantidad de borrados, cantidad de inserciones, etc.

Para poder trabajar con QGrams, vamos a agregar al proyecto Maven la biblioteca java-string-similarity de la siguiente manera:

```
<!-- QGrams -->
<dependency>
    <groupId>info.debatty</groupId>
    <artifactId>java-string-similarity</artifactId>
    <version>2.0.0</version>
</dependency>
```

Un ejemplo de uso:

```

import org.apache.commons.codec.language.Metaphone;
import org.apache.commons.codec.language.Soundex;
import org.apache.commons.text.similarity.LevenshteinDetailedDistance;
import org.apache.commons.text.similarity.LevenshteinDistance;
import info.debatty.java.stringsimilarity.QGram;

Soundex s = new Soundex();
s.difference("HELLO", "ALO");
Metaphone m = new Metaphone();
m.encode("HELLO");
LevenshteinDistance l = new LevenshteinDistance();
l.apply( "HELLO", "ALO" );

QGram qg = new QGram( 2 );
qg.distance( "Hello", "Alo" );
qg.getProfile( "Hello" );

```

## Búsqueda Exacta

Ya no vamos a tratar con similitudes. La búsqueda exacta nos va a servir para realizar búsquedas exactas.

Ejemplo: Tenemos dos arreglos de char, target y query. Queremos un código Java que permita calcular la primera aparición de source en target o -1 si no hay tal aparición.

## Algoritmo de Fuerza Bruta o Naive

Empezamos a recorrer el string comparando con el target. Cuando un char del target no matchea, volvemos para atrás tanto en el target como en el query. Se le dice naive porque no tiene ningún tipo de ventaja. Vemos que las complejidades son:

- $n \cdot m$  temporal (pero caso no está) - siendo  $n$  y  $m$  las longitudes.
- 1 espacial

El problema de este algoritmo es que no aprovecha lo que aprendió durante el recorrido cuando encuentra un **mismatch**. Hace **backtracking** en el *query* y en el *target*.

## Algoritmo Knuth-Morris-Pratt (KMP)

Este algoritmo no vuelve a chequear un carácter que ya sabe que matecheó, es decir que no hace backtracking en el target.

*Idea del algoritmo:* escanea el target de izquierda a derecha, pero usa conocimiento sobre los caracteres comparados antes de determinar la próxima posición del patrón a usar. Preprocesa el query antes de la búsqueda una vez, con el objetivo de analizar la estructura (las características del patrón query). Para ello, construye una

tabla Next del mismo tamaño del query. La **tabla de Next** tiene en cada posición “i” la longitud del borde propio más grande para el substring query desde 0 hasta i.

Ahora vamos a calcular los bordes manualmente (un borde es aquel que coincide prefijo y sufijo al mismo tiempo - la intersección de ambos conjuntos):

query	X	E	E	E
Next	0	0	0	0

Next[3]

Prefijos: lambda, X, XE, XEE, XEEE

Sufijos: lambda, E, EE, EEE, XEEE

Borde: lambda, XEEE

Next[2]

Prefijos: lambda, X, XE, XEE

Sufijos: E, EE, XEE

Borde: lambda, XEE

Next[1]

Prefijos: lambda, X, XE

Sufijos: E, XE, lambda

Borde: lambda, XE

Next[0]

Siempre 0

En todos se cumple que los bordes son el vacío y sí mismos, no tienen bordes propios.

Otro caso:

query	A	B	R	A	C	A	D	A	B	R	A
Next											

Next[10]

Prefijos:

Sufijos:

## TERMINAR

Para facilitar este cálculo, Knuth y los otros encontraron un algoritmo para encontrar también los bordes sin tener que buscar prefijos y sufijos, utilizando propiedades del cuadro (simetría). Va a encontrar coincidencias dentro del mismo string mediante los bordes.

Código Original:

```
private static int[] nextComputation(char[] query) {  
  
    int next[] = new int[query.length];  
    next[0]= 0;      // Always. There's no proper border.  
    int border = 0;  // Length of the current border  
  
    for (int rec = 1; rec < query.length; rec++) {  
        while ((border > 0) && (query[border] != query[rec]))  
            border = next[border-1];      // Improving previous computation  
        if (query[border] == query[rec])  
            border++;  
        // else border = 0; // redundant|  
        next[rec]= border;  
    }  
    return next;  
}
```

Otra forma:

```
private static int[] nextComputation(char[] query) {  
    int[] next = new int[query.length];  
  
    int border=0; // Length of the current border  
  
    int rec=1;  
    while(rec < query.length){  
        if(query[rec]!=query[border]){  
            if(border!=0)  
                border=next[border-1];  
            else  
                next[rec++]=0;  
        }  
        else{  
            border++;  
            next[rec]=border;  
            rec++;  
        }  
    }  
    return next;  
}
```

Lo primero que hace es crear un arreglo paralelo al query (ahí vemos la complejidad espacial). Comienza tomando que el primer valor es 0, lo que ya vimos antes ( $\text{Next}[0] = 0$ ).

La complejidad termina siendo  $O(N)$ , porque si en algún momento no puedo avanzar, vuelvo para atrás en igual medida de lo que avancé, por lo que a lo sumo recorremos  $2N$ , que termina siendo  $N$ .

Entonces, si la query tiene longitud  $m$ :

- Complejidad especial:  $O(m)$
- Complejidad temporal:  $O(m)$

Ahora que tenemos next, ¿cómo lo utilizamos para la búsqueda exacta? Supongamos un rec que apunta al carácter en target y pquery que apunta a un carácter en query. Mientras haya coincidencia, avanza en ambos. Cuando no la haya, se “shiftea” query a next[pquery-1], salvo que pquery sea 0, en cuyo caso hay que avanzar rec en target.

El algoritmo KMP busca en forma eficiente la aparición exacta de un query en un texto, tomando ventaja del procesamiento de la query. Pero si quisieramos buscar una query en un “grupo de documentos”, nos conviene generar un índice de dicha colección para luego buscar a través del índice.



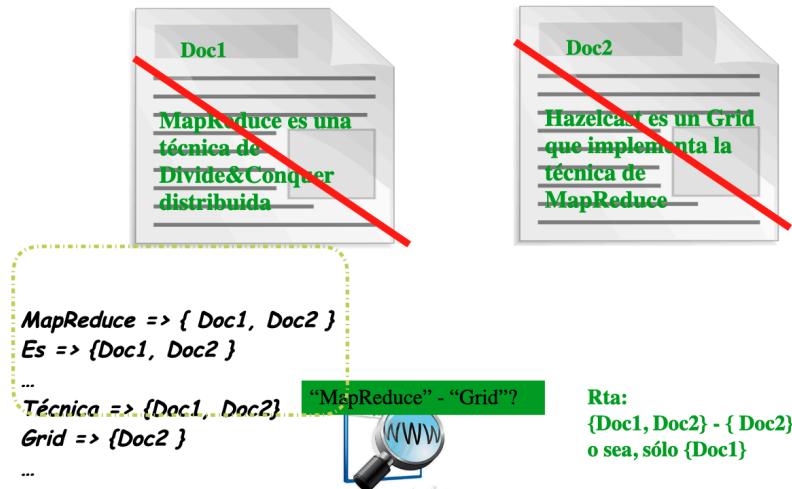
```
MapReduce => { Doc1, Doc2 }
es => {Doc1, Doc2 }
...
técnica => {Doc1, Doc2}
Grid => {Doc2 }
...
```

## Lucene

Es una biblioteca de código abierto, que sirve para armar nuestro propio motor de búsqueda. Se escribió en 1999 y sigue vigente hoy en día.

**Índice:** Es una estructura que permite llegar rápidamente al dato buscado. Si se agrega/elimina/actualiza un documento en la colección debe actualizarse. Su ventaja está en la búsqueda.

Lucene utiliza un **Archivo Invertido**: conjunto de términos que dicen a qué término pertenece. Es un mapping: término → documento.



Cuando hacemos un query, ya no buscamos más los documentos. Gracias a los índices decimos dónde está (en el ejemplo anterior, en Doc1 o Doc2). Por ejemplo, para MapReduce, el resultado es {Doc1, Doc2}.

Vamos a tratar los resultados como conjuntos. Si queremos MapReduce y grid, la intersección, es decir {Doc2}. Si queremos MapReduce pero no grid, la resta.

### Conceptos de Lucene

*Documento en Lucene*: es una secuencia de Campos (fields). Cuando se ingresa un documento, automáticamente se le asociará un ID.

*Campo en Lucene*: es un par nombre y secuencia de 1 o más términos. Por ejemplo, creamos un field "author" y contiene "Leticia Gomez". Según las configuraciones, podría hacer que "Leticia Gomez" sea un único término (token) o dos. También lo podemos pensar como un mail, donde los campos son "from", "to", etc.

*Término en Lucene*: es una secuencia de bytes (pueden interpretarse como String, números, etc.) asociada a cierto campo. Dos secuencias de bytes con igual contenido pero asociadas a 2 campos diferentes se consideran diferentes.

### Creación de un Campo

Lucene en el fondo es como una base de datos. Al crear un campo y asociarlo a cierto documento podemos:

- Almacenarlo en Lucene pero fuera del archivo invertido. Se lo almacena literal, es decir sin procesamiento (sin separar en tokens, sin pasar a mayúsculas, etc.). Esto significa que está en Lucene pero no está indexado, no participa de las búsquedas.
- Indexarlo en el archivo Lucene. Así, tokenizado o no va a participar de las búsquedas.

En código, lo creamos de la siguiente manera:

```
FieldType aField = new FieldType();
```

Sobre su almacenamiento literal, fuera del archivo invertido:

```
aField.setStored( );
```



true: se almacena

false: no se almacena

Si lo queremos indexar:

```
aField.setIndexOptions( );
```

IndexOptions.NONE : no se indiza

IndexOptions.DOCS: se indiza cada término solo con los doc ids en los que participa

IndexOptions.DOCS\_AND\_FREQS: se indiza cada término con los doc ids que participa y frecuencia en ellos

IndexOptions.DOCS\_AND\_FREQS\_AND\_POSITIONS: se indiza cada término con los doc ids en los que participa, junto con frecuencia y posiciones ordinales dentro de ellos

IndexOptions.DOCS\_AND\_FREQS\_AND\_POSITIONS\_AND\_OFFSETS: se indiza cada término con los doc ids en los que participa, frecuencia en ellos, posiciones ordinadas dentro de ellos v offsets dentro de ellos

El offset está solamente para visualización. Como cuando buscamos y se subrayan en el PDF. No le vamos a dar mucha importancia porque no estamos trabajando con frontend.

Lucene lo que hace es armar una tabla ordenada ascendente por valor del término (token) por cada campo de la siguiente manera:

Por cada campo tenemos una tabla ordenada ascendente por valor del término (token).  
Hay tantas tablas como Fields indexados hayamos creado.  
Por eso la búsqueda se realizar por campos.

Value term	Freq en docs	[ docid:freqs in docid:[positions in docid] : [startOffsets - endOffsets in docid] ]

19

Hay tantas tablas como fields indexados hayamos creado. Por eso la búsqueda se realiza por campos.

Ejemplo:

Docid 0 (a.txt) store,, game	Docid 1 (b.txt) video
Docid 3 (d.txt) Game video, review game.	Docid 2 (c.txt) game

Docid 0:

Value term	Freq en docs	[ docid ]
game	1	0 0 0 0
store	1	0 0 0 0

Docid 1:

Value term	Freq en docs	[ docid ]
game	1	0 0 0 0
store	1	0 0 0 0
video	1	1 0 0 0

Docid 3:

Value term	Freq en docs	[ docid ]
game	1 2	0 0 0 0 3 0 0 0
review	1	3 0 0 0
store	1	0 0 0 0
video	1 2	1 0 0 0 3 0 0 0

Docid 2:

Value term	Freq en docs	[ docid ]
game	2 3	0 0 0 0 3 0 0 0 2 0 0 0
review	1	3 0 0 0
store	1	0 0 0 0
video	2	1 0 0 0 3 0 0 0

Notemos que en el docid 3, la segunda vez que aparece "games" no modificamos nada porque no estamos contabilizando esto sino la frecuencia en documentos, que ya incrementamos con la primera aparición.

En el siguiente caso, vamos a guardar también la posición en el doc y vemos como ahí sí se guarda “algo” para la segunda aparición de game en el docid 3. El cuadro finalmente queda:

Value term	Freq en docs	[ docid:freqs in docid:[positions in docid] ]			
game	3	0	1	[1]	
		3	2	[0, 3]	
		2	1	[0]	
review	1	3	1	[2]	
store	1	0	1	[0]	
video	2	1	1	[0]	
		3	1	[1]	

Por último, vemos cómo funciona el offset:

The diagram illustrates the mapping of terms to offsets in documents. It shows three examples:

- Row 1:** store, game → [0, 1]
- Row 2:** review, game. → [0, 1] and [8-12]
- Row 3:** game → [0, 1]

Arrows point from the terms in each row to the corresponding offsets in the Lucene index table.

Value term	Freq en docs	[ docid:freqs in docid:[positions in docid]:[startOffsets - endOffsets in docid] ]			
game	3	0	1	[1]	[ [8-12) ]
		3	2	[0, 3]	[ [0-4), [23-27) ]
		2	1	[0]	[ [0, 4) ]
Etc etc etc					

Value term	Freq en docs	[ docid:freqs in docid:[positions in docid]:[startOffsets - endOffsets in docid] ]			
game	3	0	1	[1]	[ [8-12) ]
		3	2	[0, 3]	[ *[0-4), [23-27) ]
		2	1	[0]	[ [0, 4) ]
Etc etc etc					

Value term	Freq en docs	[ docid:freqs in docid:[positions in docid]:[startOffsets - endOffsets in docid] ]			
game	3	0	1	[1]	[ [8-12) ]
		3	2	[0, 3]	[ [0-4), [23-27) ]
		2	1	[0]	[ [0, 4) ]
Etc etc etc					

Para los archivos físicamente (en disco), se decidió en Lucene que no se almacene porque ya está la información. No nos va a dejar almacenar en estos casos. Lo que vamos a poder hacer es almacenar el path.

Si los archivos no son físicos y los queremos almacenar sin indexar, lo que hace es guardar todo completo:

Docid 0 (a.txt)	
store,, game	
<b>Doc Id</b>	<b>Field</b>
0	content
Etc etc etc	

Resumiendo: la base de la indexación corresponde a definir fields con cierta características => configurable. Lucene viene equipado con un Field muy especial que corresponde a cierta configuración/combinación muy usada. Muchas veces es suficiente y en vez de generar un field y asociarles todas sus características podemos usarlo (como si fueran un shortcut a cierta configuración).

El field predefinido es el **TextField**, que maneja tipo de dato texto. Se indexa y por default se tokenizada (se pasa a minúscula, se separa por signos de puntuación). Es como usar `IndexOptions.DOCS_AND_FREQS_POSITIONS`.

Como muchas veces este tipo de campos no se lee desde un string (de pocos caracteres) sino que se lo indiza desde un “stream” (archivo grande), Lucene solo ofrece: almacenarlo él por fuera del índice si no proviene de un stream. Caso contrario estaría triplicado el espacio: en el stream (file system), en el índice (para buscar) y además otra vez literal fuera del archivo invertido. Demasiado!

Es decir, la opción de almacenar sólo se podrá hacer si no viene de un stream.

Usar un TextField es como usar todo esto:

```
FieldType fieldD = new FieldType();
fieldD.setStored( ?? ); // DEPENDE DE DONDE VENGA LA FUENTE
fieldD.setIndexOptions(IndexOptions.DOCS_AND_FREQS_AND_POSITIONS); // SIN OFFSET
aDoc.add(new Field("content", text, fieldD ));
```

**Como setStored(false)**

*Pero solo una sentencia creo esa configuración*

```
String text="bla bla bla";
aDoc.add(new TextField("content", text, Field.Store.NO ));
```

*O bien*

```
aDoc.add(new TextField("content", text, Field.Store.YES ));
```

*O bien si tengo un reader sobre un archivo:*

```
aDoc.add(new TextField("content", aReader ));
```

**Como setStored(true)**

## Aplicaciones

Las 2 aplicaciones independientes que precisamos realizar con Lucene son: IndexBuilder y Searcher. Cuando hablamos de indexar hay dos procesos. Uno cuando indicamos los archivos los archivos que vamos a indexar (creación) y otro para revisar documentos modificados. La creación y actualización del índice es muy

importante. La frecuencia de indexación puede ser mucho más baja que la de búsqueda.

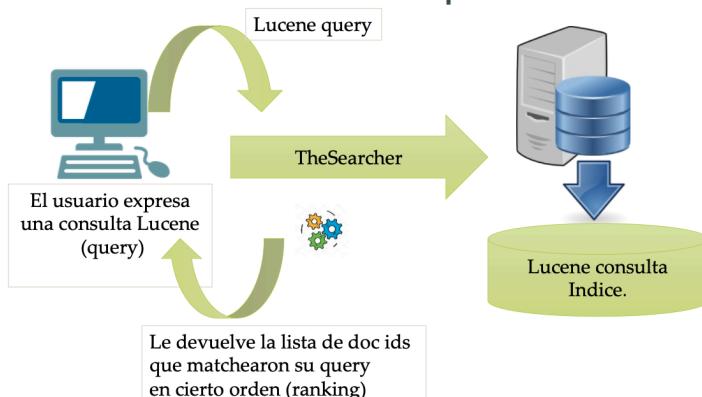
IndexBuilder: Aplicación que se encarga de generar el índice a partir de un conjunto de documentos Lucene y lo deja almacenado en cierto directorio que le indiquemos para tal efecto. Dado un directorio, itera e indexa los documentos que están allí dentro según lo que predefinidos.



Si la búsqueda de dichos archivos no anda bien, entonces Lucene no va a funcionar. Es nuestra responsabilidad mantener el índice actualizado, es decir, re ejecutar si queremos agregar documentos, o re generar el índice si los documentos en disco fueron modificados.

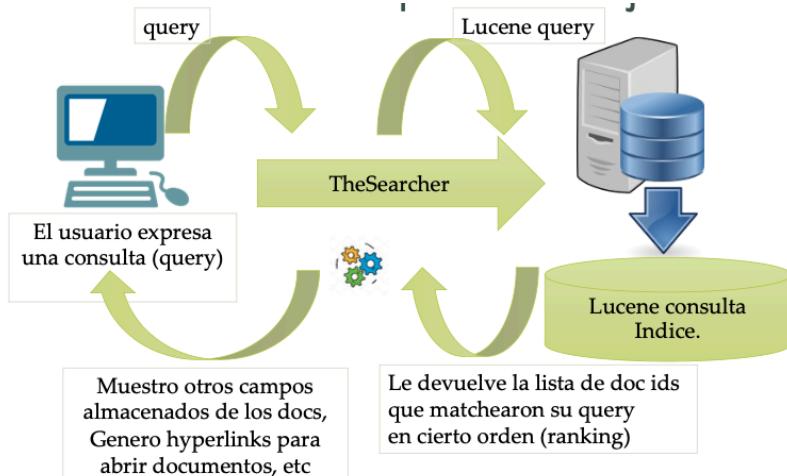
TheSearcher: Es una aplicación que se encarga de aceptar consultas y utiliza el índice construido para retornar los documentos (ids) que “matchearon” la consulta rankeados en cierto orden.

El proceso de búsqueda se ve de la siguiente manera:



Esto nos va a devolver que tal palabra está en el documento 3 (que es el id). Para saber realmente cuál es el archivo, es lo que vamos a guardar dentro de Lucene pero NO indexado. Sin esto no vamos a saber qué documento tiene la palabra.

Si quisieramos, podríamos mejorar esta búsqueda de la siguiente manera:



57

Esto no es obligatorio, tiene más que ver con el frontend que no es parte de EDA.

Las dependencias de Lucene para el pom son:

```
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-core</artifactId>
    <version>7.4.0</version>
</dependency>
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-analyzers-common</artifactId>
    <version>7.4.0</version>
</dependency>
```

Nuestros documentos Lucene han de estar formados por dos campos:

- **TextField:** "*content*" que contiene el contenido de los archivos txt que le digamos.
- **FieldType:** "*path*" que contiene el lugar físico donde están los archivos txt. No se busca en ellos, solo se lo quiere almacenar en Lucene. Idea: si consulto y Lucene me da matching quiero saber no solo que hubo X cant de documentos que matchearon sino cuáles son. Si conozco su path, podría desarrollar un front end que los muestre.

Ya vamos a poder escribir consultas al índice. Un término es la unidad básica que puede buscarse en un índice. El lenguaje para escribir consultas tiene 2 formatos:

- API Query
- Query Builder (menos programática)

API para las queries

- TermQuery: busca un solo término
- PrefixQuery: busca por prefijo
- TermRangeQuery: busca por rangos
- PhraseQuery

- WildcardQuery
- FuzzyQuery // Damerau-Levenshtein con MaxEdit 2
- Boolean Query
- Etc., etc., etc.

## TermQuery

Para hacer la búsqueda de un término, vamos a usar:

```
Term myTerm = new Term(fieldName, queryStr);
```

Por ejemplo, en nuestro TheSearcer.java:

```
// field of interest
String fieldName = "content";
String queryStr= "game";

Term myTerm = new Term(fieldName, queryStr);
Query query= new TermQuery(myTerm );
```

Para correr la query, utilizamos:

```
// run the query
long startTime = System.currentTimeMillis();
TopDocs topDocs = searcher.search(query, n: 20);
long endTime = System.currentTimeMillis();
```

Así, obtenemos un resultSet que debemos recorrer para mostrar la información.

```
// show the resultset
System.out.println(String.format("Query=> %s\n", query));
System.out.println(String.format("%d topDocs documents found in %d ms.\n",
    (endTime - startTime) ) );

ScoreDoc[] orderedDocs = topDocs.scoreDocs;

for (ScoreDoc aD : orderedDocs) {

    // print info about finding
    int docID= aD.doc;
    double score = aD.score;
    System.out.println(String.format("position=%-10d score= %10.7f", position, score ));

    // print docID, score
    System.out.println(aD);

    // obtain ALL the stored fields
    Document aDoc = searcher.doc(docID);
    System.out.println("Stored fields: " + aDoc);
    System.out.println(aDoc.get("path"));
    System.out.println(aDoc.get("content"));
    /*
    Explanation rta = searcher.explain(query, docID);
    System.out.println(rta);*/

    position++;
    System.out.println();
}
```

Todo esto ya está implementado en el archivo TheSearcher, lo único que tenemos que hacer es buscar el parámetro de queryStr para buscar las distintas palabras. Cuando corremos, obtenemos el resultado:

```
Query=> content:game

3 topDocs documents found in 7 ms.

Resultset=>

position=1      score=  1.2231436
doc=2 score=1.2231436 shardIndex=0
Stored fields: Document<stored<path:/users/nachopedemonte/documents/code/lucene/docs/c.txt>>
/users/nachopedemonte/documents/code/lucene/docs/c.txt
null

position=2      score=  0.8648931
doc=0 score=0.8648931 shardIndex=0
Stored fields: Document<stored<path:/users/nachopedemonte/documents/code/lucene/docs/a.txt>>
/users/nachopedemonte/documents/code/lucene/docs/a.txt
null

position=3      score=  0.8648931
doc=3 score=0.8648931 shardIndex=0
Stored fields: Document<stored<path:/users/nachopedemonte/documents/code/lucene/docs/d.txt>>
/users/nachopedemonte/documents/code/lucene/docs/d.txt
null

Process finished with exit code 0
```

Vemos que nos devuelve el id del doc, no el nombre. Pero si tenemos el path donde se encuentra. Esto es para no almacenarlo en Lucene, porque pueden haber archivos de fácil 2gb.

Si por ejemplo buscamos “Game” en lugar de game, vemos que no hay resultados. Esto se debe a que para indexar pasa las palabras a minúscula y las mete en la tabla. Tenemos que ser coherentes con lo que preguntamos.

Para el caso de “ga”, obtenemos el mismo resultado, ninguno. No lo encuentra porque “ga” no llega a ser un término. La búsqueda es exacta por términos, y “ga” no lo es.

Si nuevamente tomamos como queryStr “game” pero cambiamos el fieldName a “path” y buscamos, no vamos a encontrar nada porque la búsqueda debe ser por field.

(VER DENUENO)

## PrefixQuery

Ahora vamos a utilizar:

```
Query query = new PrefixQuery(myTerm);
```

y probamos los siguientes casos:

String queryStr= "game"; → Lo encuentra.

String queryStr= "ga"; → Lo encuentra también.

String queryStr= "Ga"; → No lo encuentra. Tener en cuenta que la indexación es con pasaje a minúsculas como dijimos antes.

String queryStr= "me"; → Tampoco lo encuentra. No es un prefijo.

### TermRangeQuery

Rango implica buscar si el "término" se encuentra en el intervalo especificado. El mismo puede ser abierto/cerrado a izq, abierto/cerrado a derecha:

- [BytesRefIzq, BytesRefDer]:  
fieldName, BytesRefIzq, BytesRefDer, true, true

- (BytesRefIzq, BytesRefDer):  
fieldName, BytesRefIzq, BytesRefDer, false, false

- [BytesRefIzq, BytesRefDer):  
fieldName, BytesRefIzq, BytesRefDer, true, false

- (BytesRefIzq, BytesRefDer]:  
fieldName, BytesRefIzq, BytesRefDer, false, true

Hay 2 casos. El primero es probar si son ByteRef (usando new BytesRef("string")) del campo content pertenece a los intervalos (fieldName, BytesRefIzq, BytesRefDer, boolean, boolean). El segundo es usar:

```
Query q = TermRangeQuery.newStringRange( fieldName, stringFrom, stringTo,  
boolean, boolean);
```

La segunda es la más práctica.

Vamos a probar las siguientes búsquedas:

["gam", "gum"] → Obtenemos los documentos 1, 2 y 3, porque todos los documentos que tienen game "cayeron" dentro.

["game", "game"] → ???

["game", "game") → No obtenemos nada, porque es ilógico. Le estamos pidiendo que esté y no al mismo tiempo.

I"gam", "gam"] → Tampoco obtenemos nada, porque ???

("game", "gum"] → No obtenemos nada.

("gaming", "gum") → No obtenemos nada.

En el ejemplo, vemos que para el archivo a.txt (store,game) la palabra store se encuentra dentro del intervalo que especificamos, entonces va a entrar dentro de lo que nos tiene que devolver. El intervalo toma todos los términos que estén alfabéticamente entre "gam" y "gum".

En el intervalo, el izquierdo debe ser menor o igual al derecho, sino vamos a obtener un resultado vacío, pues no tiene sentido.

### PhraseQuery

Importa el orden en que coloquemos las palabras en la búsqueda. Vamos a utilizar:

```
Query query= new PhraseQuery(fieldName, word1, word2, ... wordN );
```

Buscamos:

Frase: "store" "game" → devuelve el doc 0 (a.txt) que contiene (store,,game)

Frase: "store,," "game" → no encuentra, porque store,, no existe. Busca los tokens, y "store" se guardó sin las comas.

Frase: "game" "store" → no encuentra, porque en ningún archivo se encuentran ambos términos en ese orden.

Frase: "store game" → no encuentra, porque ambos términos están tokenizados.

Frase: "store,, game" → no encuentra

Frase: "game" "review" → no encuentra

Frase: "game" "video" "game" → no encuentra

Frase: "game" "video" "review" → devuelve el doc 3 (c.txt) que contiene (Game video, review game.)

¿Qué información del índice le permite a Lucene responder a las consultas por PhraseQuery? Recordemos que no tiene almacenado el contenido. Lo que queremos saber es cuál de todas las columnas de la tabla utiliza. Lo que utiliza son los *ordinales*. Sin ellos no podría responder esta query. Por esto TextField guarda hasta la posición inclusive.

### WildcardQuery

Para esta query, \* representa cualquier secuencia de caracteres inclusive vacío, y ? representa un carácter cualquiera. Vamos a utilizar:

```
Query query= new WildcardQuery(myTerm);
```

para buscar:

queryStr= "g\*e" → Devuelve todos los documentos donde se encuentra "game".

queryStr= "g?me" → Devuelve lo mismo que el anterior.

queryStr= "g?m" → No encuentra, porque el ? es para un único carácter y no existe ningún término de 3 letras que comience con "g" y termine con "m".

queryStr= "G??e" → Tampoco encuentra, porque no hay término que comience con "G", recordar que la indexación pasa a minúsculas.

queryStr= "\*" → Devuelve todos los documentos, porque en todos hay secuencias de caracteres.

### FuzzyQuery (Damerau-Levenshtein con MaxEdit 2)

Va a buscar utilizando únicamente dos conversiones (recordar Levenshtein), si requiere más no lo encuentra. MaxEdit es operaciones, no similitud (no está normalizado). Vamos a utilizar:

```
Query query= new FuzzyQuery(myTerm);
```

Donde los casos a probar son:

queryStr= “gno” → No encuentra nada. Hay que hacer más de 2 operaciones para convertirlo en “game” o cualquier otro término, por lo que obtenemos dicho resultado.

queryStr= “gem” → Encuentra aquellos donde esté el término “game”. Con dos operaciones obtenemos “game”.

queryStr= “agem” → Igual que el anterior.

queryStr= “hm” → No encuentra nada.

queryStr= “ham” → Nuevamente devuelve aquellos donde esté “game”.

### BooleanQuery

Las consultas no siempre son tan puntuales. Muchas veces se necesita combinar esas características. Inclusive entre varios campos. Tomemos los siguientes casos:

- que cierta frase aparezca en el campo “content” pero que no aparezca tal término.
- que cierta frase aparezca en el campo “content” y también tal término parezca en otro campo.
- que empiece con tal prefijo en el campo “content” o bien se parezca en otro campo indexado.

Si bien BooleanQuery soluciona muchas de estas problemáticas, con API es tedioso usarlo porque hay que combinar varios constructores (para el AND, para el OR, para el NOT). Entonces, vamos a utilizar **QueryBuilder** (sin la API).

### QueryBuilder

Lucene define un *lenguaje de consulta* y él se encarga de parsearlo y transformarlo en varias invocaciones de APIs (las mismas que vimos antes). Resulta muy práctico, pero para poder usarlo debemos conocer dicho lenguaje. Si no lo respetamos, obtenemos error en tiempo de ejecución en el parser.

Veamos las equivalencias de la API a la QueryBuilder:

API	QueryBuilder
1.1TermQuery	fieldName:termino
1.2 PrefixQuery	fieldName:term*
1.3 TermRangeQuery	fieldName:[start TO end]]
1.4 PhraseQuery	fieldName：“term1 ... termN”
1.5 WildcardQuery	fieldName: *subterm?
1.6 FuzzyQuery	fieldName:termino~2
1.7 BooleanQuery	AND OR NOT (+ -)

En TermRangeQuery, si queremos que se incluya, usamos corchetes, sino llaves. Cuando con API preguntamos por TermQuery por “Game”, ¿lo encontró? ¿Por qué?

No, porque para ingresar al índice le hemos aplicado un StandardAnalyzer() que detectó tokens por espacios y símbolos de puntuación y además los insertó en minúsculas. Si la query no es igual, no la va a encontrar, como ya nos venía sucediendo.

Lo que podríamos hacer es aplicar un Analyzer (como los de la construcción del índice) para que modifique la query (pase a minúsculas, elimine stopwords, etc.).

```
QueryParser qp = new QueryParser(null, new StandardAnalyzer());
```

Hasta ahora vimos StandardAnalyzer para la separación en tokens, pero Lucene viene con otros:

- SimpleAnalyzer()
- StandardAnalyzer()
- WhitespaceAnalyzer()
- StopAnalyzer()=>  
    CharArraySet sw = StopFilter.makeStopSet("de", "y");  
    new StopAnalyzer(sw);
- EnglishAnalyzer() // opcional stop words
- SpanishAnalyzer() // opcional stop words.
- CustomAnalyzer()

Al StopAnalyzer le puedo pasar una serie de “stop words” que van a funcionar como separadores. Al CustomAnalyzer le podemos modificar algunos parámetros que queramos.

El código sirve para ver cómo hace Lucene por dentro (para inspeccionar la separación en tokens). Como verán, no estamos creando documentos, solo usando un Low Level API para ver qué tokens genera.

Vamos corriendo los distintos analyzers con un mismo string y vemos las diferencias:

```
String = "Estructura de datos. Y algoritmos; 2021-Q1 en eda.itba.edu"
```

Standard: estructura - de - datos - y - algoritmos - 2021 - q1 - en - eda.itba.edu

Simple: estructura - de - datos - y - algoritmos - q - en - eda - itba - edu

Whitespace: Estructura - de - datos. - Y - algoritmos; - 2021-Q1 - en - eda.itba.edu

Stop: estructura - datos - algoritmos - q - en - eda - itba - edu

Spanish: estructur - dat - algoritm - 2021 - q1 - eda.itba.edu

El spanish le sacó el género a la palabra, para tener la raíz de las palabras, usa el “stemmer algorithm”.

Para usar el CustomAnalyzer:

```

Analyzer analyzer = CustomAnalyzer.builder()
    .withTokenizer("standard")
    .addTokenFilter("lowercase")
    .addTokenFilter("stop")
    .addTokenFilter("porterstem")
.build();

```

Ahora vamos a utilizar el QueryParser. Veamos la diferencia con la API:

API	QueryParser
<pre> queryStr="game"; Term myTerm = new Term("content", queryStr); Query query= new TermQuery(myTerm ) </pre>	<pre> queryStr="content:game"; Query query= queryparser.parse(queryStr); </pre>

Notemos que si ahora hacemos la búsqueda:

String queryStr= "content:Game";

Nos va a encontrar los 3 archivos donde está “game”, porque le estamos pasando el StandardAnalyzer. Lo mismo sucede si buscamos “game,,”.

Si ahora buscamos por ejemplo “game\*”, “ga\*”, “Ga\*”, también nos va a encontrar las apariciones de “game”.

Cuando queremos buscar con el RangeQuery, lo hacemos:

“content:{gaming TO gum}”

Obtenemos los siguientes resultados:

[“gam”, “gum”] → Donde está game

[“game”, “game” ] → Lo mismo

[“game”, “game”) → nada

[“gum”, “gam”] → nada

(“game”, “gum”] → nada

(“gaming”, “gum”) → nada

Para usar el PhraseQuery:

“content:\”store game\””

Obtenemos

Frase: “store game” → doc0

Frase: “store,,,” “game” → doc0

Frase: “game” “store” → nada

Frase: “store game” → doc0

Para usar el WildcardQuery:

“content:g\*e”

Obtenemos:

queryStr= “g\*e” → Encuentra los de game

queryStr= “g?me” → Encuentra los de game

queryStr= "g?m" → No encuentra  
queryStr= "G??e" → Encuentra los de game  
queryStr= "\*" → NO podemos poner \* en la primera posición.

Para usar FuzzyQuery:

"content:gn~2"

Obtenemos:

queryStr= "gno" → No encuentra nada  
queryStr= "agen" → Tampoco  
queryStr= "agem" → Encuentra las de game  
queryStr= "hm" → No encuentra nada  
queryStr= "ham" → Encuentra las de game

Ahora vamos a ver el BooleanQuery que no llegamos a ver con la API:

- Si queremos por UNA de dos palabras:
  - content:store OR content: game
  - content:store content:game
  - content:store || content:game
- Si queremos buscar por AMBAS palabras:
  - content:store AND content: game
  - content:store && content: game

Notemos que:

content:review OR (content:game AND NOT content:yo)

(content:review OR content:game) AND NOT content:yo

No son equivalentes por un tema de lógica de AND y OR. Vamos a necesitar

Aclaración:

Si todas las query se hace sobre el mismo fieldName puede especificarse una sola vez (un default)

En vez de :

```
String queryStr= "content:game AND content:store";
QueryParser queryparser = new QueryParser(null, new
StandardAnalyzer());
Query query= queryparser.parse(queryStr);
```

Escribimos:

```
String queryStr= "game AND store";
QueryParser queryparser = new QueryParser("content", new
StandardAnalyzer());
Query query= queryparser.parse(queryStr);
```

## Query de un Término

Veamos cómo rankear a aquellos documentos que matchean con la consulta. Dada una colección de N documentos  $D = \{DOC1, DOC2, \dots, DOCn\}$  y una query=term, para aquellos documentos que matchean la consulta:

$$\begin{aligned} \text{Score}(DOC_i, \text{query}) &= \\ \text{FormulaLocal}(DOC_i, \text{term}) * \text{FormulaGlobal}(D, \text{term}) \end{aligned}$$

**FormulaLocal:** quiere calcular qué tan relevante es esa query respecto a un documento en particular a rankear. Por esto, es de interés la frecuencia de la aparición del término en el documento normalizado por la longitud del mismo.

La fórmula queda:

$$\text{FormulaLocal}(DOC_i, \text{query}) = \sqrt{\frac{\# \text{freq(term in } DOC_i)}{\# \text{term existentes en } DOC_i}}$$

**FormulaGlobal:** quiere calcular qué tan relevante es esa query respecto a la colección de documentos existente. Tampoco se quiere linealidad, y por eso se aplica un logaritmo. La fórmula queda:

$$\begin{aligned} \text{FormulaGlobal}(DOC, \text{query}) &= \\ 1 + \log_e \frac{1 + \# \text{docs en la colección}}{1 + \# \text{docs que contienen term}} \end{aligned}$$

Ej. Tomamos el caso anterior de los cuatro documentos y tomamos “game” para la query.

Para la fórmula global tenemos:

$$\begin{aligned} \text{FormulaGlobal}(DOC, \text{query}) &= \\ 1 + \log_e \frac{1 + \# \text{docs en la colección}}{1 + \# \text{docs que contienen term}} \\ &= 1 + \log_e \frac{1+4}{1+3} \\ &= 1.2231436 \end{aligned}$$

Luego, vimos que se encuentra en los documentos 0, 2 y 3 (2 veces en este último). Entonces calculamos las fórmulas globales:

- Doc 0:

$$\begin{aligned} \text{FormulaLocal}(DOC_0, \text{query}) &= \\ \sqrt{\frac{\# \text{freq(term in } DOC_0)}{\# \text{term existentes en } DOC_0}} \\ &= \sqrt{\frac{1}{2}} = 0.7071067 \end{aligned}$$

Y el score:

$$\begin{aligned} \text{Score}(DOC_0, \text{query}) &= \\ \mathbf{0.70710677} \\ * \mathbf{1.2231436} \\ = \mathbf{0.8648931} \end{aligned}$$

- Doc 2:

$$\begin{aligned} \text{FormulaLocal}(DOC_2, \text{query}) &= \\ \sqrt{\frac{\# \text{freq(term in } DOC_2\text{)}}{\#\text{term existentes en } DOC_2}} \\ = \sqrt{\frac{1}{1}} &= 1 \end{aligned}$$

Y el score:

$$\begin{aligned} \text{Score}(DOC_2, \text{query}) &= \\ \mathbf{1} * \mathbf{1.2231436} &= \mathbf{1.2231436} \end{aligned}$$

- Doc 3:

$$\begin{aligned} \text{FormulaLocal}(DOC_3, \text{query}) &= \\ \sqrt{\frac{\# \text{freq(term in } DOC_3\text{)}}{\#\text{term existentes en } DOC_3}} \\ = \sqrt{\frac{2}{4}} &= \mathbf{0.7071} \end{aligned}$$

15

Y el score:

$$\begin{aligned} \text{Score}(DOC_3, \text{query}) &= \\ \mathbf{0.70710677} \\ * \mathbf{1.2231436} &= \mathbf{0.8648931} \end{aligned}$$

Por lo tanto, si ordenamos según score:

- Score(DOC2,query) = 1.2231436
- Score(DOC0,query) = 0.8648931
- Score(DOC3,query) = 0.8648931

Donde podemos alternar entre DOC3 y DOC0 ya que tienen el mismo score.

Si tenemos que el query usa un solo **término modificado** por FuzzySearch, Range, Prefix, Wildcard : se devuelve como score 1 a los documentos que matchean.

Ej: query=ga\* sobre los mismos docs

Dado que tenemos:

- Score(DOC0,query) = 1
- Score(DOC2,query) = 1
- Score(DOC3,query) = 1

No podemos ordenar.

## Query Multi-Término

Para aquellos documentos que matchearon la consulta, les aplica la siguiente fórmula:

$$\text{Score}(\text{DOC}_i, \text{query}) = \sum_{\substack{\text{term in query y no tiene NOT}}} \text{FormulaLocal}(\text{DOC}_i, \text{term}) * \text{FormulaGlobal}(D, \text{term})$$

O sea, hacemos la sumatoria de los cálculos parciales de los scores de cada término participante de la query, siempre que NO esté modificado por NOT.

Ej. Calcular el ranking de documentos cuando se busca por el término “game AND NOT store” en la colección de documentos que tomamos antes.

Vemos primero que:



El score debido al término “game” ya lo hemos calculado.. El score debido al término “store” no aplica a la fórmula de score porque está afectado por NOT. Obtendremos para esos 2 documentos los valores de score anteriores.

- $\text{Score}(\text{DOC}2, \text{query}) = 1.2231436$
- $\text{Score}(\text{DOC}3, \text{query}) = 0.8648931$

Y en el resultado primero aparece Doc2 y luego Doc3.

Ej. Calcular el ranking de documentos cuando se busca por el término “game OR store” en la misma colección de documentos.

Vemos que:



Primero calculamos la fórmula global de store:

$$\begin{aligned} \text{FormulaGlobal}(\text{DOC}, "store") &= 1 + \log_e \frac{1 + \# \text{docs en la colección}}{1 + \# \text{docs que contienen "store"}} \\ &= 1 + \log_e \frac{1+4}{1+1} = 1.9162907 \end{aligned}$$

Calculamos las fórmulas locales:

$$\begin{aligned}
 & \text{FormulaLocal}(\text{DOC}_0, "store") \\
 &= \sqrt{\frac{\# \text{freq}("store" \text{ in } \text{DOC}_0)}{\#\text{terms existentes en } \text{DOC}_0}} \\
 &= \sqrt{\frac{1}{2}} = \mathbf{0.7071067} \quad \text{Score}(\text{DOC}_0, \text{query}) = \\
 & \quad \mathbf{0.70710677} \\
 & \quad * \mathbf{1.9162907} \\
 & \quad = \mathbf{1.3550219}
 \end{aligned}$$

En DOC2 y DOC3 va a ser cero pues no aparece:

- FormulaGlobal(DOC3,"store")= 0
- FormulaGlobal(DOC2,"store")= 0

Ahora planteamos la suma de los scores para ambos términos:

$$\text{Score}(\text{DOC}_0, "game \text{ OR } store") = 0.8648931 + 1.3550219 = 2,219915$$

$$\text{Score}(\text{DOC}_2, "game \text{ OR } store") = 1.2231436 + 0 = 1.2231436$$

$$\text{Score}(\text{DOC}_3, "game \text{ OR } store") = 0.8648931 + 0 = 0.8648931$$

Y vemos que rankean:

- Doc0
- Doc2
- Doc3

### Consideraciones

Lucene no ofrece escalabilidad. A medida que el conjunto de documentos crece o los clientes que realizan consultas crecen, eso puede ser un problema. Si se precisa un backend que permita escalabilidad, debemos usar Solr o Elasticsearch. Ambos frameworks están construidos sobre Lucene pero permiten escalar ya que coordinan varias instancias de Lucene, las cuales pueden correr en diferentes computadoras (en un cluster de computadoras).

## Unidad 3 - Estructuras Lineales

Hasta ahora buscamos apariciones de elementos en un conjunto. Esto se puede hacer de distintas formas:

- Dejar la colección como está e ingeníárselas para navegar en ella, como hicimos con el KMP.
- Generar una estructura auxiliar, índice, que facilite la búsqueda. Ese fue el caso del archivo invertido.

Siendo el índice la estructura auxiliar que se utiliza para encontrar un elemento, entonces la búsqueda debe ser eficiente.

Caso de uso 1: documentos para search engine. La colección de documentos contiene



6

Caso de uso 2: La colección contiene “alumnos”, pero no sabemos qué información tiene alumnos. Si queremos buscar por legajo:



Vemos que en este caso el índice es “uno a uno” con la información. Si por ejemplo tomamos como índice la edad, esto no se va a respetar. Tenemos una clave repetida manejada sin compactar. Para no repetir, podríamos juntar la información:

20  
19  
...  
Clave de búsqueda  
(key)

<58622, Ana Garcia, 20, [agarcia@gmail.comLeo Nilo, 20, \[lnilo@gmail.com\]\(mailto:lnilo@gmail.com\)>  
<58333, Pablo Conte, 19, \[pconte@gmail.com...\]\(mailto:pconte@gmail.com\)](mailto:agarcia@gmail.com)

Información asociada. Puede estar en RAM o (si es mucha) indicará cómo llegar a la información en disco

## Características de Índices

La clave de búsqueda puede o no tener repetidos. Si es única no tiene sentido hablar de compactación. Si puede repetirse, podremos tener la info asociada compactada o no. La clave de búsqueda debe permitir buscar rápidamente la información adicional.

Ahora bien, el índice no sólo se utiliza para “buscar”. Hay otras operaciones necesarias sobre él.

- Búsqueda.
- Inserción: el índice debe reflejar los datos de la colección. O sea, si inserto un documento en la colección, preciso que el índice lo refleje.
- Borrado: el índice debe reflejar los datos de la colección. O sea, si borro un documento en la colección, preciso que el índice lo refleje.

¿Qué estructura de las que conocemos podría ser buena para representar un índice? Supongamos que los repetidos no se compactan y que hay espacio pre alocado suficiente para las inserciones.

Xta	Búsqueda	Inserción	Borrado
Arreglo (cualquiera, desordenado)	⊗	✓ (los agrego al final)	⊗
Arreglo ordenado por clave de búsqueda	✓	⊗	⊗
Hashing	?	?	?

Si calculamos la complejidad para estas estructuras:

	Búsqueda	Inserción	Borrado
Arreglo (cualquiera, desordenado)	✗ $O(n)$	✓ $O(1)$	✗ $O(n)$
Arreglo ordenado por clave de búsqueda		✗ $O(n)$	✗ $O(n)$
Hashing	✓ ??	✓ ??	✓ ??

(El de búsqueda para arreglo ordenado es  $O(\log_2 n)$ ).

El problema del arreglo es que tiene que garantizar la continuidad de sus elementos.

El problema del hashing es si tiene que resolver colisiones. Si no tiene colisiones es  $O(1)$ , pero es ideal...

Los 3 casos se penalizan si se acaba el espacio pre alocado.

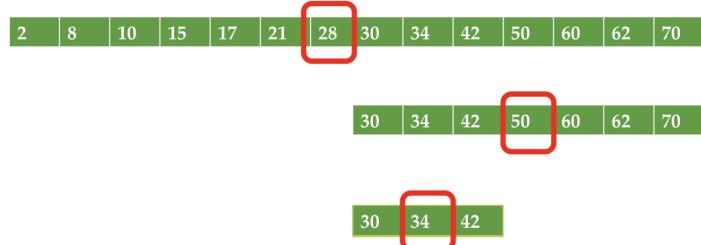
Por esto, pareciera ser que el Hash es ideal para los índices, pero debemos ver qué sucede si necesitamos, por ejemplo, buscar por rangos o devolver el máximo/mínimo elemento.

No hay estructuras de datos perfectas. Dependen de los casos de uso necesarios. En general los objetivos se contraponen y hay que buscar un trade-off. Vamos a comenzar analizando el comportamiento de los arreglos ordenados por clave de búsqueda.

### Arreglos Ordenados

Vimos que la complejidad de búsqueda para este caso es  $O(\log_2 n)$ . El algoritmo que toma la ventaja de que el arreglo esté ordenado es la búsqueda binaria (la que vimos en discreta).

Busco el 34



Vamos dividiendo de a dos, tomando el del medio y comparando. Y así nos vamos moviendo en el arreglo.

¿Cómo calcular complejidades en algoritmos recursivos?

En PI y POO usamos la técnica de programación Divide y Triunfarás (Divide and Conquer): La solución de un problema de tamaño de entrada N se divide en problemas de tamaño menor hasta que la solución es trivial. Finalmente, se combinan los resultados parciales para dar solución al problema original. Este es un “tipo de algoritmo” para resolver problemas.

### Teorema Maestro

Si una fórmula recurrente puede expresarse así:

$$T(N) = a * T\left(\frac{N}{b}\right) + c * N^d$$

Invocación recursiva que divide en subproblemas

Combinación de soluciones parciales

Donde:

N es el tamaño de entrada del problema

$a \in N_{\geq 1}$  (¿Cuántas invocaciones recursivas se realiza ese paso?)

$b \in N_{>1}$  (Mide tasa en que se reduce el tamaño del input)

$c \in R_{>0}$

$d \in R_{\geq 0}$

Entonces la complejidad O grande está dada por los siguientes 3 casos (c no cuenta):

- Si  $a < b^d$  entonces el algoritmo es  $O(N^d)$ .
- Si  $a = b^d$  ba entonces el algoritmo es  $O(N^d * \log N)$ .
- Si  $a > b^d$  entonces el algoritmo es  $O(N^{\log_b a})$ .

Ej. Aplicamos este algoritmo a Fibonacci.

$$\text{Fibo}(N) = \begin{cases} \quad & \text{si } N \leq 1 \\ \text{Fibo}(N-1) + \text{Fibo}(N-2) & \text{si } N \geq 1 \end{cases}$$

```
static public int fibo(int N) {
    if (N <= 1)
        return N;
    return fibo(N-1) + fibo(N-2);
}
```

Entonces:  $\text{Times}(N) = \text{Times}(N - 1) + \text{Times}(N - 2) + 4$

El 4 viene a ser las restas (N-1 y N-2), el return, etc.

Ahora debemos ver: ¿Cuáles son las constantes a, b, c, y d? ¿Qué caso aplica? ¿Cuál es la complejidad O grande?

No tiene “la pinta” de lo que planteamos antes. No hay  $b > 1$  que divida  $N/b$ . **NO podemos aplicar el teorema maestro.** Tenemos que buscar otra forma de calcular su complejidad temporal.

En general, la Técnica Divide y Triunfarás procede de la siguiente forma:

- Divide el problema en subproblemas de un mismo tamaño.
- Resuelve cada subproblema en forma independiente, por recursión
- Combina los resultados parciales para dar solución final.

Cuando esto ocurre, puede aplicarse el Teorema Maestro.

Ej. Apliquemos a Búsqueda Binaria.

Podríamos comenzar garantizando que todo llega bien al algoritmo de búsqueda binaria, pero el cálculo lo tenemos que aplicar al algoritmo recurrente (no acá):

```
static public int indexOf(int[] arreglo, int cantElementos, int elemento) {
    if (cantElementos <= 0)
        throw new IllegalArgumentException("cantidad de elementos debe ser positiva");

    // chequear si esta ordenado y sino ordenarlo
    // bla bla bla

    return indexOf(arreglo, 0, cantElementos-1, elemento);
}
```

La parte recurrente del programa es:

```

static private int indexOf(int[] arreglo, int izq, int der, int elemento) {
    if (izq > der)
        return -1; // no estaba

    // hay intervalo [izq, der]
    int mid= (der + izq) / 2;

    if (elemento == arreglo[mid] ) // lo encontre
        return mid;

    if (elemento < arreglo[mid] )
        return indexOf( arreglo, izq, mid-1, elemento);

    return indexOf( arreglo, mid+1, der, elemento);
}

```

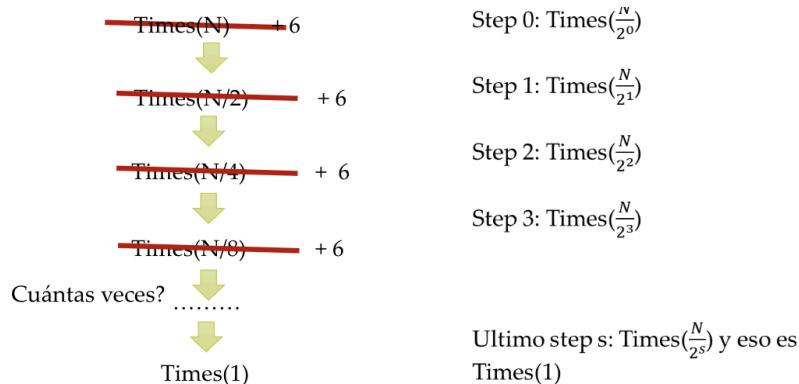
Vemos que tenemos  $Times(N) = Times(\frac{N}{2}) + 6$

El 6 viene de la suma de las operaciones de todo lo que no es llamada recursiva. La suma, división, asignación, acceso de arreglo, etc. Como es 6, sabemos que c es 6 y d es 0, por lo que no depende de N.

Vemos en el último if que llamamos a una o la otra, pero es una sola llamada. Notemos que el input (int mid) lo estamos dividiendo siempre sobre 2. Entonces va a ser  $Times(N/2)$ . Vemos que a = 1 (llamamos una vez) y b = 2. Como tenemos esto, vamos a poder aplicar el teorema maestro.

Como  $1 = 2^0$ , tenemos el segundo caso de los que planteamos. Entonces es  $O(N^0 * \log N)$ , es decir  $O(\log N)$ .

Otra forma de encontrar la complejidad del algoritmo recursivo para búsqueda binaria:



Entonces, como  $\frac{N}{2^s} = 1$   
Entonces,  $N = 2^s$

La cantidad de steps realizados s es  $\log_2 N$

$$Times(N) = \sum_{i=1}^{\log_2 N} 6$$

$$Times(N) = 6 * \log_2 N$$

..

El algoritmo es  $O(\log_2 N)$

Cuando el código es no-recursivo miramos las invocaciones, ciclos (paralelos vs anidados), etc. Si el código es recursivo hay que considerar la cantidad de invocaciones realizadas también.

Ej. Código No Recursivo.

```
static public int surprise(int[] arreglo, int dim) {
    int vble= 0;

    for (int rec = 0; rec < dim; rec++) {
        for (int j = rec+1; j < dim; j++) {
            if (arreglo[rec] * arreglo[j] == 0)
                vble++;
        }
    }

    return vble;
}
```

$$\text{Times(dim)} = \sum_{rec=0}^{dim-1} (3 + \sum_{j=rec+1}^{dim-1} 5)$$

$$\text{Times(dim)} = \sum_{rec=0}^{dim-1} (3 + 5(dim - 1 - (rec + 1) + 1))$$

$$\text{Times(dim)} = \sum_{rec=0}^{dim-1} (3 + 5(dim - rec - 1))$$

$$\text{Times(dim)} = \sum_{rec=0}^{dim-1} (3 + 5(dim - rec - 1))$$

$$\text{Times(dim)} = \sum_{rec=0}^{dim-1} (5 \cdot dim - 5 \cdot rec - 2)$$

$$\text{Times(dim)} = \sum_{rec=0}^{dim-1} (5 \cdot dim - 2) - \sum_{rec=0}^{dim-1} 5 \cdot rec$$

$$\text{Times(dim)} = (5 \cdot dim - 2) \cdot dim + 5 \sum_{rec=0}^{dim-1} rec$$

$$\sum_{i=1}^n i = \frac{n(n+1)}{2}$$

$$\text{Times(dim)} = (5 \cdot dim - 2) \cdot dim + 5 \frac{(dim-1)(dim-1+1)}{2}$$

Entonces es  $O(dim^2)$ .

Ej. Tomemos el siguiente código:

```
public static int surprise(int N) {
    if (N < 4)
        return 16;

    for (int i = 0; i < N; i++) {
        System.out.println(i);
    }

    int auxil= surprise( N / 3);
    int auxi2= surprise( N / 3);

    return auxil + auxi2;
}
```

$$\text{Times}(N) = \begin{cases} 2 * \text{Times}(N/3) + 4 & \text{si } N \geq 4 \\ 1 & \text{si } N < 4 \end{cases}$$

Vemos que podemos aplicar el Teorema Maestro, pues  $a=2$ ,  $b=3$ ,  $c=4$  y  $d=1$ . Tenemos el primer caso de los que planteamos,  $a < b^d$ , por lo que es  $O(N)$ .

### Búsqueda Binaria Iterativa vs Recursiva

En búsqueda binaria hablamos del cálculo de complejidad temporal para algoritmos recurrentes. Ahora bien, la búsqueda ya puede implementarse en forma recursiva (la que vimos) o iterativa.

```

static private int indexOf(int[] arreglo, int izq, int der, int elemento) {
    if (izq > der)
        return -1; // no estaba
    // hay intervalo [iza, der]
    int mid= (der + izq) / 2;
    if (elemento == arreglo[mid] ) // lo encontre
        return mid;
    if (elemento < arreglo[mid] )
        return indexOf( arreglo, izq, mid-1, elemento);
    return indexOf( arreglo, mid+1, der, elemento);
}

static private int indexOf(int[] arreglo, int izq, int der, int elemento) {
    while (izq <= der) {
        // hay intervalo [iza, der]
        int mid= (der + izq) / 2;
        if (elemento == arreglo[mid] ) // lo encontre
            return mid;
        if (elemento < arreglo[mid] )
            der= mid-1;
        else
            izq=mid+1;
    }
    return -1;
}

```

Versión recursiva

Versión iterativa

Vemos que la complejidad temporal es la misma para ambos algoritmos,  $\log_2$ . Esto se da por la cantidad de veces que vamos a hacer el ciclo. Vamos siempre dividiendo a la mitad.

Ahora bien, debemos mirar la complejidad espacial para determinar cuál conviene usar. Vemos que en el caso iterativo es de orden 1 mientras que en la recursiva se realizan  $\log_2$  steps, por lo que  $O(\log_2)$ .

### Ordenación de Arreglos

En nuestra implementación del índice precisamos de un arreglo ordenado. Invocamos el `Arrays.sort()` de java. Pero hay otras maneras de ordenar:

**Quicksort:** Opera in-place, es decir que opera sobre el mismo arreglo (no necesita de otro). Aplica la técnica Divide & Conquer y puede implementarse recursivamente o iterativamente. Lo que hace es particionar en sub arreglos.

Elije un elemento que funciona como pivote (puede ser el primero). Va a modificar el arreglo para saber en qué lado va el pivote y garantizar que a la izquierda van a estar los menores y a la derecha los mayores.

Dicho formalmente: En cada subarreglo elige un pivot y ordena para que todos los elementos a la izquierda del pivot sean menores que él y los de la derecha sean mayores que él => el pivot está en la posición correcta. Si un sub-arreglo tiene 0 o 1 elemento, está ya ordenado (no continua) => fin de la recursividad.

Ej. Tomemos el siguiente arreglo. El pivote “inicial” es el primer elemento, el 34.

1	2	3	4	5	6	7	8	9	10	11	12	13	
34	10	8	60	21	17	28	30	2	70	50	15	62	42

Particiona la lista  $\leq 34$  y  $> 34$  (o bien  $< 34$  y  $\geq 34$ )

Cuando realiza la ordenación y partitiona, nos va a quedar:

10	8	21	17	28	30	2	15	34	60	70	50	62	42
↑ Pos 8													

Vemos que el único elemento que estamos seguros que está bien ubicado es el 34 en la posición 8, porque tiene todos los elementos más chicos a su izquierda y los más grandes a la derecha.

Ahora se repite el proceso con los sub-arreglos. Por ejemplo en el derecho, se toma el 60 y lo mismo.

● Para la lista derecha: pivot 60

10	8	21	17	28	30	2	15	34	60	70	50	62	40
----	---	----	----	----	----	---	----	----	----	----	----	----	----

Particiona la lista derecha  $\leq 60$  y  $> 60$  (o bien  $< 60$  y  $\geq 60$ )

10	8	21	17	28	30	2	15	34	50	40	60	62	70
----	---	----	----	----	----	---	----	----	----	----	----	----	----

Cuando nos queda 0 o 1 elemento, termina el algoritmo porque ya está ordenado.

La implementación queda:

```
static private void swap(int[] unsorted, int pos1, int pos2) {
    int auxi= unsorted[pos1];
    unsorted[pos1]= unsorted[pos2];
    unsorted[pos2]= auxi;
}
```

```

private static void quicksortHelper (int[] unsorted, int leftPos, int rightPos) {
    if (rightPos <= leftPos )
        return;

    // tomamos como pivot el primero. Podria ser otro elemento
    int pivotValue= unsorted[leftPos];

    // excluimos el pivot del cjto.
    swap(unsorted, leftPos, rightPos); |
```

// particionar el cjto sin el pivot  
int pivotPosCalculated= partition(unsorted, leftPos, rightPos-1, pivotValue);

// el pivot en el lugar correcto  
swap(unsorted, pivotPosCalculated, rightPos);

// salvo unsorted[middle] todo puede estar mal  
// pero cada particion es autonoma  
quicksortHelper(unsorted, leftPos, pivotPosCalculated - 1);  
quicksortHelper(unsorted, pivotPosCalculated + 1, rightPos );

}

La función partition lo que va a hacer es devolvernos la posición donde va el pivote. Esto funciona de la siguiente manera:

Recordemos que tenemos el pivote en la última posición. Entonces en el arreglo que va hasta el anterior, vamos a tener dos índices al principio y final (que NO incluye el pivote). Vamos a empezar a comparar desde ambos lados con el pivote, y cuando encontremos uno en izq y otro en der que están “mal”, los swapeamos.

Posibles implementaciones:

```

static private int partition(int[] unsorted, int leftPos, int rightPos, int pivotValue) {
    while (leftPos <= rightPos) {

        while (leftPos <= rightPos && unsorted[leftPos] < pivotValue)
            leftPos++;

        while (leftPos <= rightPos && unsorted[rightPos] > pivotValue)
            rightPos--;

        if (leftPos <= rightPos)
            swap(unsorted, leftPos++, rightPos--);
    }
    return leftPos;
}
```

(De la cátedra)

```

static private int partition(int[] unsorted, int leftVal, int rightVal, int pivotValue){
    int l = leftVal;
    int r = rightVal;

    while(l <= r){
        if(unsorted[l] < pivotValue){
            l++;
        }
        else{
            if(unsorted[r] > pivotValue){
                r--;
            }
            else{
                swap(unsorted, l, r);
            }
        }
    }

    return l;
}

```

(Mía)

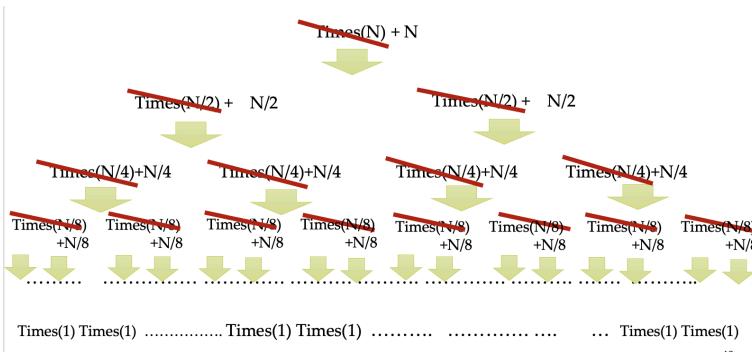
Notemos que el peor caso va a ser cuando esté todo ordenado, y no vamos a poder aplicar el Teorema Maestro porque hay dos invocaciones recursivas pero no en partes iguales.

Si calculamos la complejidad temporal para el peor caso:

$$\begin{aligned}
\text{Times}(N) &= \\
&= N + \text{Times}(N-1) \\
&= N + (N-1) + \text{Times}(N-2) \\
&= N + (N-1) + (N-2) + \text{Times}(N-3) \\
&\dots \\
&= N + (N-1) + (N-2) + \dots + 3 + \text{Times}(2) \\
&= N + (N-1) + (N-2) + \dots + 3 + 2 + \text{Times}(1) \\
&= N + (N-1) + (N-2) + \dots + 3 + 2 + 1
\end{aligned}$$

$$\text{Times}(N) = \sum_{i=1}^N i = N^2 \text{ o sea } O(N^2)$$

Quicksort está diseñado para, en el mejor de los casos, tener listas de tamaño mitad en cada iteración. Si eso se lograra, entonces



Cada vez sepáramos en la mitad. Termina en el paso s donde  $\text{Times}(1)$  es  $\text{Times}(\frac{N}{2^s})$  o sea  $1 = \frac{N}{2^s}$ . La cantidad de steps realizados s es  $\log_2 N$ .

En este caso vamos a poder aplicar Teorema Maestro:

$$\text{Times}(N) = 2 * \text{Times}(N/2) + O(N)$$

O sea:  $a=2$ ,  $b=2$  y  $d=1$ . Finalmente, es el caso dos, o sea:

$$O(N^d * \log N)$$

$$O(N * \log N)$$

Para mejorar el algoritmo en el caso que viene casi ordenado, podríamos cambiar el Pivot, por ejemplo tomando el elemento del medio, uno random, la mediana de 3 elementos candidatos predeterminados, etc.

En cuanto a la complejidad espacial:

- En el peor caso, debido a los stackframes tenemos  $O(N)$ .
- En el mejor caso, debido a los stackframes tenemos  $O(\log_2 N)$ .

**Mergesort**: es un algoritmo de ordenamiento eficiente que sigue el enfoque de "divide y conquista".

1. Divide: Divide la lista no ordenada en dos sublistas de tamaño aproximadamente igual.
2. Conquista: Ordena recursivamente cada sublista.
3. Combina: Combina las sublistas ordenadas para producir una única lista ordenada.

La complejidad temporal del mergesort es  $O(n \log n)$ .

## IMPLEMENTAR

### Generics

Java es un lenguaje estáticamente tipado, hay que declarar el tipo de una variable antes de usarla. Sin Generics, los casteos son una posibilidad de errores que se detectan en tiempo de ejecución.

Ej:

```
List v = new ArrayList();
v.add("test");
Integer i = (Integer)v.get(0); // Runtime Exception!
```

Ej: sin casteos, también podemos tener RuntimeException. Los arreglos en Java sin Generics son covariantes => puedo poner elementos de un subtipo.

```
Object[] elems = new String[2];
elems[0] = "hi";
elems[1] = 100; // RuntimeException!
```

Como dijimos, en Java debemos declarar el tipo de variable antes de usarla. Con la introducción de Java Generics ese “tipo” puede parametrizarse. Generics está pensado para parametrizar y minimizar errores. Técnicamente hablando, Generics fue implementado usando la **Técnica de Erasure**, que consiste en reemplazar todo tipo de parámetro con su “bound/restricción” y si no lo hay lo reemplaza por Object. De ser necesario realiza casteos.

Ej:  
public class P<T> {  
 public void method(T p) {  
 ...  
 }  
}

<T> is unbound => Object

Ej:  
public class P<T extends Comparable<T>> {  
  
 public void method(T p)  
 {  
 ...  
 }  
}

<T> is bound => Comparable

En Java los Generics son invariantes, pues no se puede asignar un subtipo generics a un supertipo generics.

Ej: ni compila

```
ArrayList<Integer> ints = new ArrayList<Integer>();
List<Number> numbers = ints;
```

Ej: ni compila

```
public class P<T> {  
    ..  
  
    public static void main(String[] args) {  
        P<Integer> myi = new P<Integer>();  
        P<Number> myp = myi;  
    }  
}
```

Hay muchas restricciones que se establecieron al diseñar en Java Generics y Erasure (leer <https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html>), pero las que más nos van a importar para la materia son:

- No puede un built-in sustituir un tipo paramétrico.
- No puedo crear dinámicamente un arreglo de tipo paramétrico (en tiempo de ejecución) porque su tipo no se conoce ya que en compilación se hizo erasure.

Ej: Probar

```
public class P<T> {
    private T[] arreglo= new T[10];
}
```

Ej: Probar

```
public class P<T> {
    private T[] arreglo;

    public void initialize(int dim) {
        arreglo= new T[dim];
    }
}
```

Tenemos algunas soluciones para estas limitaciones:

**Opción 1:** guardar un arreglo de Objects (no T). Castear cuando sea necesario.

Escribamos entre todos la clase P<E>

```
public class P<E> {

    private Object[] arreglo;

    public void initialize(int dim) {
        ...
    }

    public void setElement(int pos, Eelement) {
        ...
    }

    public E getElement(int pos)
    {
        ...
    }
}
```

Caso de Uso:  
`P<Number> auxi = new P<>();  
auxi.initialize(5);  
auxi.setElement(3, 10);  
auxi.setElement(2, 20.8);  
for (int i= 0; i < 5; i++) {  
System.out.println( auxi.getElement(i) );  
}`

Esto nos va a quedar de la forma:

```

public class P1<E> {

    private Object[] arreglo;

    public void initialize(int dim) {
        arreglo= new Object[dim];
    }

    public void setElement(int pos, E element) {
        arreglo[pos]= element;
    }

    @SuppressWarnings("unchecked")
    public E getElement(int pos)
    {
        return (E) arreglo[pos];
    }
}

```

## Opción 2: Usar reflection. Escribamos la clase P<T>

```

public class P<T> {

    private T[] arreglo;

    public void initialize(int dim, Class<T> theClass) {
        ...
    }

    public void setElement(int pos, T element) {
        ...
    }

    public T getElement(int pos)
    {
        ...
    }
}

Caso de Uso:
P<Number> auxi = new P<()>();
auxi.initialize(5, Number.class);
auxi.setElement(3, 10);
auxi.setElement(2, 20.8);
for (int i= 0; i < 5; i++) {
    System.out.println( auxi.getElement(i));
}

```

La implementación nos queda:

```

package test;

import java.lang.reflect.Array;

public class P2<E> {

    private E[] arreglo;

    @SuppressWarnings("unchecked")
    public void initialize(int dim, Class<E> theClass) {
        arreglo= (E[]) Array.newInstance(theClass, dim);
    }

    public void setElement(int pos, E element) {
        arreglo[pos]= element;
    }

    public E getElement(int pos)
    {
        return arreglo[pos];
    }

    public static void main(String[] args) {
        P2<Number> auxi = new P2<>();
        auxi.initialize(5, Number.class);
        auxi.setElement(3, 10);
        auxi.setElement(2, 20.8);
        for (int i= 0; i < 5; i++) {
            System.out.println( auxi.getElement(i) );
        }
    }
}

```

Notemos que ahora el initialize recibe la clase como parametro para poder hacer el arreglo, utilizando la API reflect.

### Opción 3: Combinamos un poco de las dos anteriores.

```

public class PObjectToT<E>{

    private E[] arreglo;

    @SuppressWarnings("unchecked")
    public void initialize(int dim){

    }

    public void setElement(int pos, E element){

    }

    public E getElement(int pos){
    }
}

```

Caso de Uso:

```

P<Number> auxi = new P<>();
auxi.initialize(5);
auxi.setElement(3, 10);
auxi.setElement(2, 20.8);
for (int i= 0; i < 5; i++) {
    System.out.println( auxi.getElement(i) );
}

```

El código nos queda:

```

package test;

public class P3<E> {

    private E[] arreglo;

    @SuppressWarnings("unchecked")
    public void initialize(int dim) {
        arreglo= (E[]) new Object[dim];
    }

    public void setElement(int pos, E element) {
        arreglo[pos]= element;
    }

    public E getElement(int pos)
    {
        return arreglo[pos];
    }

    public static void main(String[] args) {
        P3<Number> auxi = new P3<>();
        auxi.initialize(5);
        auxi.setElement(3, 10);
        auxi.setElement(2, 20.8);
        for (int i= 0; i < 5; i++) {
            System.out.println( auxi.getElement(i) );
        }
    }
}

```

Pero qué pasa en las 3 opciones si tenemos que el generic E extiende Comparable?

- Opción 1: Funciona perfecto, no cambia nada. (La más recomendada por la cátedra, pero depende del caso).
- Opción 2: Funciona perfecto también.
- Opción 3: No funciona. Es recomendable no usar esta.

### Otros Problemas que requieren Estructuras Lineales

Más allá de la problemática de los “índices”, existen otros problemas que requieren de estructuras de datos sencillas (que podrían implementarse con un arreglo o una lista).

El concepto de búsqueda de un elemento nos llevó a la idea de precisar un índice para facilitar la búsqueda. Pero si no precisamos “buscar” elementos? Es más, si ni siquiera precisamos compararlos entre sí? Existe otro “orden” de elementos que tienen que ver con el momento en que se generan los datos o su orden de llegada. Por ejemplo, los editores de texto permiten realizar operaciones: copy, paste, move, etc. y afortunadamente los avanzados permiten “deshacer” las últimas acciones realizadas. Permiten “arrepentirse” e inspeccionar la herramienta. La estructura auxiliar que nos permite implementar esta característica es la pila (Stack o LIFO).

Otro caso es cuando en tiempo de ejecución se invoca a un método se utiliza el stack del runtime para almacenar: parámetros, variables locales y dirección de retorno. Si un lenguaje de programación no dispone de recursión, y el algoritmo es de naturaleza recursiva, debemos usar un Stack para solucionar esa dificultad.

## Stack

Colección de datos ordenada por orden de llegada. La única forma de acceso es por medio de un elemento distinguido: TOPE que es el último elemento que llegó.

Las operaciones que debe ofrecer son:

- **push**: agrega un elemento a la colección y se convierte en el nuevo tope.
- **pop**: quita un elemento de la colección y cambia el tope de la pila. Es una operación destructiva y solo puede usarse si la colección no está vacía.
- **isEmpty**: devuelve true/false según la colección tenga o no elementos
- **peek**: devuelve el elemento tope pero sin removerlo. Es una operación de lectura y solo puede usarse si la colección no está vacía.

Para desarrollar un stack, podríamos utilizar un arreglo que tiene la ventaja de la *continuidad*, que hace más rápida la búsqueda. Pero en este caso no necesitamos buscar, necesitamos siempre el último elemento. Además, si se acaba el espacio sí debemos buscar espacio contiguo en otro lugar y copiar componentes.

Si se lo implementara con una lista lineal simplemente encadenada no tendríamos el problema de navegación. Los elementos en el Stack solo se acceden por el tope. Es solo cuestión de “apuntar” el tope al elemento conveniente. Es decir, el primer elemento de la lista. Así, jamás tenemos que recorrer para push/pop.

Es importante ver que en una lista, siempre guardamos en la primera posición, por lo que NO necesitamos puntero al último elemento, es decir, NO debe estar doblemente encadenada.

Vemos que la clase Stack de java, lo que hace es extender un vector, lo que está “mal” desde el punto de vista de la POO. Debería encapsular (tener) un vector o a los sumo, no extenderlo. Si solo vamos a usar los 4 métodos, podemos usar Stack de java.

## Caso de Uso: Evaluador de expresiones

Una expresión es una combinación de operadores y operandos. En esta discusión vamos a considerar un subconjunto de operadores: sólo binarios.

Las expresiones se pueden clasificar según la notación que utilizan:

- **prefija**: el operador se encuentra antes de los operandos sobre los que aplica
- **infija**: el operador se encuentra entre los operandos sobre los que aplica
- **postfija**: el operador se encuentra detrás de los operandos sobre los que aplica.

Dados 2 operandos A y B, el operador binario \* puede tener las siguientes posibilidades:

Prefija	Infija	Postfija	Prefija Inversa	Infija Inversa	Postfija Inversa
* A B	A * B	A B *	* B A	B * A	B A *

El problema con la evaluación de una expresión infija es que existen ambigüedades cuando dos operadores tienen la misma precedencia:  $10 - 2 - 3$  ¿Cómo se evalúa? Debe resolverse con asociatividad. Pero se complica más aún cuando aparecen paréntesis que cambian las prioridades.

En las notaciones prefija y postfija el uso de paréntesis es innecesario debido a que el orden de los operadores determina el orden real de las operaciones en la evaluación de expresiones.

Algoritmo para evaluar una expresión que ya esté en notación *postfija*:

- Cada operador en una expresión postfija se refiere a los operandos previos en la misma.
- Cuando aparece un operando hay que postergarlo porque no se puede hacer nada con él hasta que no llegue el operador, y como la notación es postfija el operador va a llegar después. Por lo tanto cada vez que se encuentre un operando la acción a tomar es “pushearlo” en una pila
- Cuando aparezca un operador en la expresión implica que llegó el momento de aplicárselo a los operandos que lo preceden, por lo tanto se deben “popear” los dos elementos más recientes de la pila , aplicarles el operador y volver a dejar el resultado en la pila porque dicho valor puede ser operando para otra subexpresión (al resultado habrá que aplicársele el próximo operador que aparezca).
- Cuando se termine de analizar la expresión de entrada el resultado de su evaluación es el único valor que quedó en la pila.

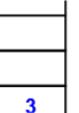
Para implementar esto, vemos que no se necesitó navegar por dentro de la estructura en busca de otras componentes. Siempre se respetó el orden de llegada de los elementos. Por lo tanto vamos a usar un stack.

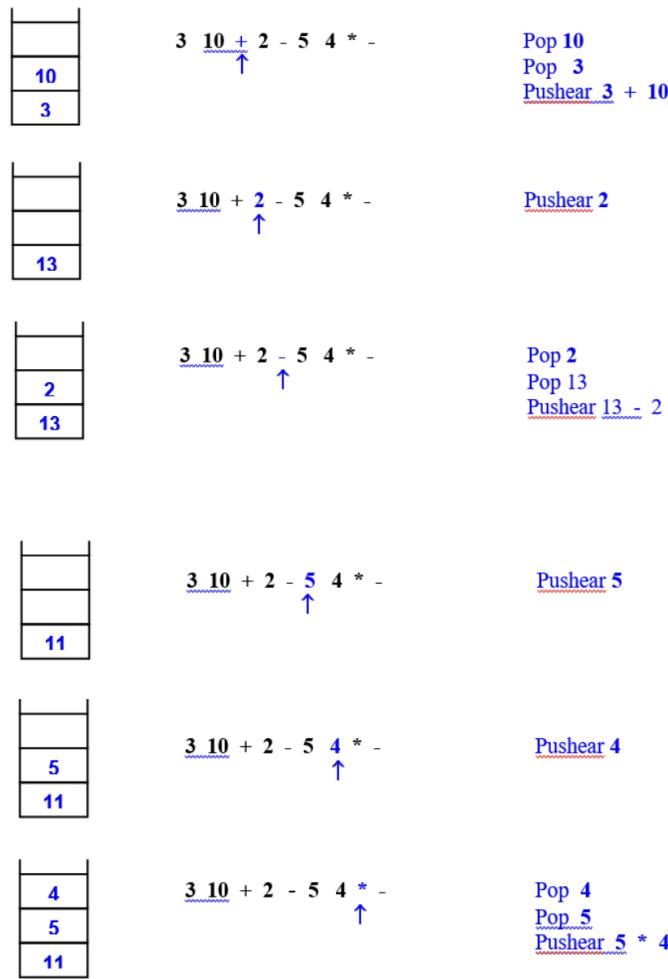
Ejemplo: Supongamos que tenemos la expresión postfija

$3 \ 10 \ + \ 2 \ - \ 5 \ 4 \ * \ -$

(que corresponde a la infija:  $(3 + 10) - 2 - 5 * 4$  ).

Nosotros no vamos a manejar strings acá, porque estaríamos asumiendo demasiadas cosas, como que sabemos la longitud, etc. Nosotros necesitamos analizar de a un token de izquierda a derecha. El algoritmo queda:

Pila	Entrada a analizar	Acción a tomar
	$3 \ 10 \ + \ 2 \ - \ 5 \ 4 \ * \ -$	Pushear $3$
	$3 \ 10 \ + \ 2 \ ^ \ 5 \ 4 \ * \ -$	Pushear $10$



Vamos a tener 2 problemas:

- Cómo parsear un string de entrada para separarlo en tokens válidos (dónde terminan los números y los operadores).
- La evaluación en sí de la expresión postfija. Eso incluye el manejo de errores.

Si nuestros operadores admitidos son + - \* /:

3 - 4 es inválida porque el - espera un operando previo

0.2 3 ? Es inválida porque el ? No es operador válido

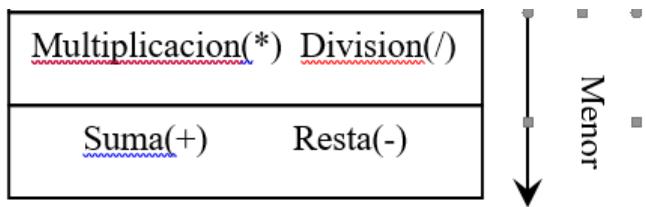
4 3 1 - 4 Es inválida porque faltan operadores

Para leer de la entrada estándar, vamos a usar la clase Scanner. Permite leer de estándar input/archive/string información, indicarle cuáles son los separadores y navegar por sus tokens (iterador).

### Parser de Precedencia de Operadores

Algoritmo para transformar expresión en notación infija a expresión en notación postfija. Idea: cada vez que aparezcan varios operadores se consultará un tabla que

indique cuál se evalúa primero. Si dos operadores tienen la misma precedencia, se utiliza la regla de asociatividad para saber cuál se evalúa primero.



La asociatividad de esas 4 operaciones es de izquierda a derecha. Por lo tanto si tenemos  $A - B + C$  en la expresión postfija primero tiene que aparecer el  $-$  (ya que se evaluará primero  $A - B$ ). Eso es debido a que  $+-$  tienen la misma precedencia, pero como es asociativo a izquierda si llega un  $“+”$  y había previo un  $“-”$ , entonces el previo  $-$  se evalúa antes que el  $+$ .

Si tenemos  $A + B * C / D$  entre el  $“+”$  y el  $“*”$ , el segundo debe aparecer antes que el  $“+”$  en la expresión postfija. Pero entre el  $“*”$  y el  $“/”$  el  $“*”$  también tiene que aparecer antes porque aunque tiene igual precedencia, la asociatividad a izquierda indica que se evaluará primero el  $“*”$  y luego el  $“/”$ . Resumiendo, deberían aparecer en la expresión postfija: primero el  $*$ , luego el  $/$  y finalmente el  $+$ . La expresión original es equivalente a :  $A + (B * C) / D$  .

Salvo que se quisiera cambiar este comportamiento, no hace falta colocar paréntesis en la expresión infija.

Algoritmo: Cada operando de la expresión infija se copia en la expresión postfija. Cuando aparece un operador hay que analizar precedencia respecto del resto de los previos operadores, por lo tanto los casos se reducen a chequear la precedencia entre el tope de la pila y el operador current:

- Si la pila está vacía, se “pushea” el operador current ya que no se lo puede comparar con nada porque es el primero de la subexpresión.
- Si la pila no está vacía:
  - Si el tope de la pila tiene mayor precedencia que el operador current, entonces se realizar “pop” del operador en la pila y se lo copia en la expresión postfija hasta que se acabe la pila o quede en ella uno de menor precedencia que el operador current. Se pushea al operador current, ya que hay que postergar su acción hasta que aparezca otro operador.
  - Si el tope de la pila tiene menor precedencia que el operador current no se puede ir todavía... Se pushea al operador current, ya que hay que postergar su acción hasta que aparezca otro operador.
- Cuando se terminó de analizar la expresión infija, se “popean” todos los operadores de la pila y se copian en la expresión postfija.

Ejemplo: Supongamos que tenemos la expresión infija  $3 + 10 * 2 / 1$   
El algoritmo funciona así:

<i>Pila</i>	<i>Entrada a analizar</i>	<i>Acción a tomar</i>
	$3 + 10 * 2 / 1$ ↑	Concatenar 3 <i>Posfija= “3”</i>
	$3 + 10 * 2 / 1$ ↑	<u>Pushear</u> + <i>Posfija= “3”</i>
	$3 + 10 * 2 / 1$ ↑	Concatenar 10 <i>Posfija= “3_10”</i>
	$3 + 10 * 2 / 1$ ↑	<u>Pushear</u> * <i>Posfija= “3_10”</i>
	$3 + 10 * 2 / 1$ ↑	Concatenar 2 <i>Posfija= “3_10_2”</i>
	$3 + 10 * 2 / 1$ ↑	Pop * (concatenarlo) <u>Pushear</u> / <i>Posfija= “3_10_2_*”</i>
	$3 + 10 * 2 / 1$ ↑	Concatenar 1 <i>Posfija= “3_10_2_*_1”</i>
	Fin de la entrada	Pop / (concatenarlo) Pop + (concatenarlo) <i>Posfija= “3_10_2_*_1/_+”</i>

O sea,

$$3 + 10 * 2 / 1 \rightarrow 3 \ 10 \ 2 \ * \ 1 \ / \ +$$

La tabla de precedencia es:

Elemento que está en el tope de la pila (previo)

Elemento que está siendo analizado (actual)

	+	-	*	/
+	true	true	False	false
-	true	true	False	false
*	true	true	True	True
/	true	true	True	true

Para desarrollarla en java, tenemos dos opciones:

Opción 1:

```
// opcion 1
private static Map<String, Integer> mapping = new HashMap<String, Integer>()
{ { put("+", 0); put("-", 1); put("*", 2); put("/", 3); } };

private static boolean[][] precedenceMatrix=
{ { true, true, false, false},
  { true, true, false, false},
  { true, true, true, true},
  { true, true, true, true},
};

private boolean getPrecedence(String tope, String current)
{
    Integer topeIndex;
    Integer currentIndex;

    if ((topeIndex= mapping.get(tope))== null)
        throw new RuntimeException(String.format("tope operator %s not found", tope));

    if ((currentIndex= mapping.get(current)) == null)
        throw new RuntimeException(String.format("current operator %s not found", current));

    return precedenceMatrix[topeIndex][currentIndex];
}
```

Opción 2:

```
// opcion 2: asumo que - no es parte de ningun operador posible
private static Map<String, Boolean> precedenceMap= new HashMap<String, Boolean>()
{ {
    put("+_+", true); put("+_-", true); put("+_*", false); put("+/_", false);
    put("-_+", true); put("-_-", true); put("-_*", false); put("-/_", false);
    put("*_+", true); put("*_-", true); put("*_*", true); put("*/_", true);
    put("/_+", true); put("/_-", true); put("/_*", true); put("//_", true);
} };

private final static String extraSymbol= "_";

private boolean getPrecedence(String tope, String current)
{
    Boolean rta= precedenceMap.get(String.format("%s%s%s", tope, extraSymbol, current));
    if (rta == null)
        throw new RuntimeException(String.format("operator %s or %s not found", tope, current));

    return rta;
}
```

Implementamos el algoritmo para pasar de infija a postfija:

```

private String infijaToPostfija()
{
    String postfija= "";
    Stack<String> theStack= new Stack<String>();

    while( scannerLine.hasNext() ) {
        String currentToken = scannerLine.next();

        if ( isOperand(currentToken) ) {
            postfija+= String.format("%s ", currentToken);
        }
        else {
            while ( !theStack.empty() && getPrecedence(theStack.peek(), currentToken) ) {
                postfija+= String.format("%s ", theStack.pop());
            }

            theStack.push(currentToken);
        }
    }

    while ( !theStack.empty() ) {
        postfija+= String.format("%s ", theStack.pop());
    }

    return postfija;
}

```

Vemos que este método auxiliar es private, pero es muy importante por lo que lo podríamos testear. Al estar private, no podemos hacer testeos con JUNIT. Una opción sería pasarlo a public, testear y volver a private, pero no es práctico. Vamos a usar reflection para “skipear” la visibilidad en java. Ejemplo:

```

public class Sorpresa {
    private double f()
    {
        return 35;
    }

    private double f(double param)
    {
        return param;
    }
}

@Test
void test2() throws NoSuchMethodException, SecurityException,
IllegalAccessException, IllegalArgumentException,
InvocationTargetException {
    Sorpresa sorpresaInstance = new Sorpresa();

    Method myMethod
        = Sorpresa.class.getDeclaredMethod( "f");

    myMethod.setAccessible(true);

    double result
        = (Double) myMethod.invoke(sorpresaInstance);

    assertEquals(35, result);
}

```

Lo que hacemos en la línea resaltada es decirle que es accesible y luego lo invocamos. Además, podemos inyectar entrada estándar en los testeos:

```

// inyecto entrada estandard
String input = "-9 -1 - 10 2 * / 1 5 - 2 -3 / / *";
InputStream inputstream = new ByteArrayInputStream(input.getBytes());
System.setIn(inputstream);

```

Si ahora queremos agregar la potencia, solamente vamos a cambiar la tabla. Pero la potencia es asociativa de derecha a izquierda:

$$a^{b^c} = a^{(b^c)}$$

La tabla nos va a quedar:

Elemento que está en el tope de la pila (previo)	Elemento que está siendo analizado (actual)				
	+	-	*	/	^
+	true	true	false	false	False
-	true	true	false	false	false
*	true	true	true	true	False
/	true	true	true	true	false
^	True	True	True	true	false

En el último cuadro, vemos que la potencia “no se puede ir” cuando llega otra potencia, porque sino sería asociativa de izquierda a derecha.

Ahora, vamos a incluir los **paréntesis** en las expresiones. Notemos que los mismos no deben aparecer en la salida ya que no son necesarios en las expresiones postfijas (ni en prefijas). Lo más sencillo será tomarlos como operadores:

- Si el operador current es un “(“ el mismo debe postergarse hasta que aparezca ”)”. Es decir, completar la tabla para que se lo pushee siempre.
- Si el operador current es un ”)” el mismo debe sacar todos los operadores de la pila y concatenarlos en el string de salida hasta encontrar el “(“ que aparea con él. Cuando en el tope aparezca el “(“ debe sacarlo del tope de la pila pero no concatenarlo (ya que los paréntesis no van a la expresión postfija).

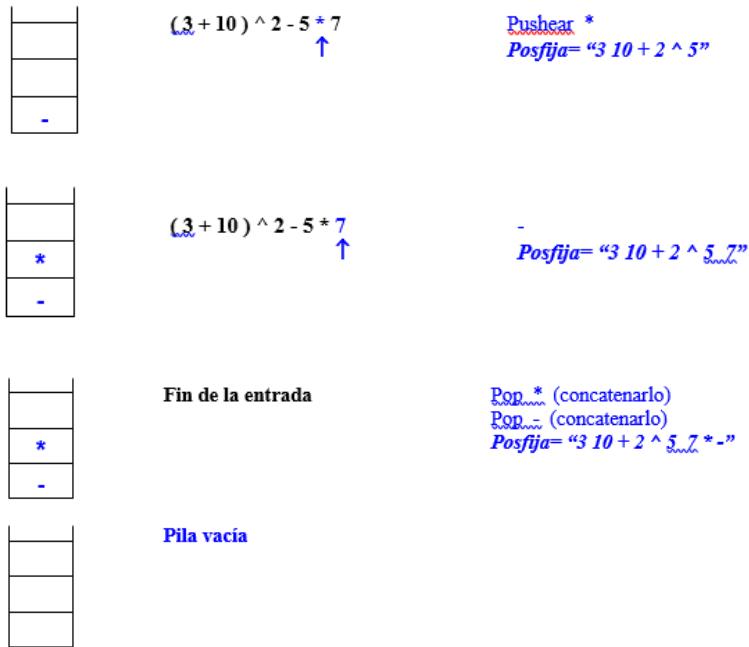
Al completar la tabla, debemos establecer que la precedencia entre “(“ y ”)” sea false para manejarla como un caso especial, pues sino se vacía la pila.

Por lo tanto, la tabla nos queda:

Elemento que está en el tope de la pila (previo)	Elemento que está siendo analizado (actual)						
	+	-	*	/	^	(	)
+	True	True	False	False	False	False	True
-	True	True	False	False	False	False	True
*	True	True	True	True	False	False	True
/	True	True	True	True	False	False	True
^	True	True	True	True	False	False	True
(	False	False	False	False	False	False	false

Ejemplo: Supongamos que tenemos la expresión infija  $(3 + 10)^2 - 5 * 7$

<i>Pila</i>	<i>Entrada a analizar</i>	<i>Acción a tomar</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>Pushear (</i> <i>Posfija= " "</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>-</i> <i>Posfija= "3"</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>Pushear +</i> <i>Posfija= "3"</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>-</i> <i>Posfija= "3 10"</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>Pop + (concatenar)</i> <i>Pop (</i> <i>Posfija= "3 10 +"</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>Pushear ^</i> <i>Posfija= "3 10 + 2"</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>-</i> <i>Posfija= "3 10 + 2 ^"</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>Pop ^ (concatenar)</i> <i>Pushear -</i> <i>Posfija= "3 10 + 2 ^ 5"</i>
	$(3 + 10) ^ 2 - 5 * 7$ ↑	<i>-</i> <i>Posfija= "3 10 + 2 ^ 5"</i>



Ahora, podríamos considerar una nueva extensión del algoritmo, donde los operandos no sean solo constantes sino variables previamente definidas.

### Problemas

*Recursos compartidos:* Hay un único recurso (impresora) y múltiples clientes que llegan asincrónicamente y precisan usarlo. A medida que el único recurso se libera, puede tomar otro pedido.

*Múltiples Recursos compartidos:* Similar al anterior, pero con múltiples recursos compartidos administrados centralizadamente. Ej: una sola lugar para acceder a las múltiples cajas para cobrar.

*Algoritmos en Grafos:* Algoritmos como BFS precisan estructuras auxiliares para posponer los momentos de procesamiento mientras se visitan los elementos del grafos.

*Pipes:* Un Pipe es un mecanismo para comunicar 2 procesos donde un proceso escribe en el pipe y otro proceso lee la información en el orden en que fue escrita, en forma secuencial. Ambos procesos abren su canal de comunicación en ambos extremos simultáneamente. Son procesos independientes (asincrónicos) con su propia velocidad (de producción o lectura) por eso se precisa de una estructura auxiliar.

Todos estos problemas se pueden solucionar con una misma estructura, una **cola**.

### Queue

Es una colección de datos ordenada por orden de llegada. La única forma de acceso es por medio de dos elementos distinguidos: FIRST indica cuál es el más antiguo de los elementos de la colección y tiene prioridad para salir, y LAST marca el elemento más reciente que ha llegado a la colección. Las operaciones que debe ofrecer son:

- `queue(element)`: agrega un elemento a la colección convirtiéndolo en el más reciente o sea, se convierte en el nuevo LAST.
- `deque()`: quita el elemento más antiguo de la colección (FIRST) y cambia el FIRST. Es una operación destructiva y solo puede usarse si la colección no está vacía.
- `peek()`: devuelve el elemento más antiguo de la colección (FIRST) sin removerlo (sin cambiar el FIRST). No es destructiva. Solo puede usarse si la colección no está vacía.
- `isEmpty()`: devuelve true/false según la colección tenga o no elementos.
- `size()`: (opcional) devuelve la cantidad de elementos de la colección y es ideal para estimar cuánto hay que esperar por ser atendido.

Vemos que en java es una interfaz que extiende Collection, y los métodos que no quiere los “inhibe”.

Para implementar el queue, tenemos dos opciones:

**Lista:** Como cuando encolamos lo hacemos a través del último, podemos apuntarlo y cambiar el last. No tenemos problema de navegación. porque los elementos en la lista solo se acceden por el FIRST o LAST. Es solo cuestión de “apuntarlos” convenientemente.

**Arreglo:** Misma lógica. En un arreglo es un problema tener espacio libre y no usarlo. Si se le acaba el espacio, hay que reallocar (se vuelve O(N)). Eso ocurre en unbounded Queue mientras que no en bounded Queue (el usuario conoce la capacidad máxima del queue).

Si se tiene una unbounded Queue (no hay límite en la cantidad de elementos que puede manejar), LinkedList es superior a ArrayList, pero si se trata de un bounded Queue (hay límite y podría arrancar con ese tamaño pre alocado porque nunca crecerá) se puede realizar una implementación de las operaciones de encolar y desencolar en O(1) también. Hay que hacer un tratamiento “circular” de un arreglo para aprovechar al máximo ese espacio pre alocado. Agregar el método private `isFull()` para chequear si se puede o no seguir encolando.

Java no viene equipada con una clase Queue sino que es una interface que limita las operaciones de la clase LinkedList. Las operaciones ofrecidas son similares a las discutidas y son las esperadas para una Queue.

## Índices y Arreglos Ordenadas

Ventajas:

- El arreglo ordenado es una buena estrategia para implementar un índice (búsqueda rápida de la info). Permite algoritmo de “búsqueda binaria”.
- Excelente para algunas operaciones que requieren acceso a una componente en particular, ya que se accede por “posición”. Ej: getMaximo(), getMínimo().

Desventajas:

- Los datos tienen que estar contiguos. Para garantizarlo, el insertado/borrado de un elemento requiere “mover” otras componentes.
- Además, cuando se acaba el espacio pre-alocado, generar más espacio (aún de a “chunks”) implica generar otro espacio contiguo y llevar todas las componentes. Por eso conviene hacerlo de a “chunks”.

Tratando de superar el problema de la contigüidad y re-alocación del espacio, podemos pensar en estructuras de datos que permitan que los elementos estén “físicamente aislados y lógicamente conectados”. Opción: “Lista lineal simplemente encadenada” con su root (Primer elemento): null. Si insertamos un elemento y se genera su espacio en \$AAFF.

El problema es que no vamos a poder acceder a un elemento “del medio” sin recorrer como hacíamos en las listas.

*Lista Lineal Simplemente Encadenada:* Es una estructura de datos que está compuesta por 0 o más nodos. Cada nodo (elemento) almacena 2 cosas: su info y la referencia al elemento siguiente.

*Lista Lineal Simplemente Encadenada Ordenada:* Es una lista lineal simplemente encadenada que además mantiene los elementos ordenados con algún criterio de ordenación.

La inserción puede implementarse de diferente maneras:

- Resuelto totalmente en SortedLinkedList (la provista), en forma iterativa.
- Resuelto totalmente en SortedLinkedList, en forma recursiva.
- Delegando a la clase Node la inserción.

Análogamente, el remove() se puede implementar en cualquiera de estas 3 maneras.

Notemos que algunas operaciones como getMax(), se puede reducir su complejidad ( $O(n)$ ) guardando información extra. Para eso, tenemos:

*Lista Lineal Simplemente Encadenada con Header:* Es una estructura de datos compuesta por:

- Un elemento distinguido llamado “header” que tiene la referencia del primer elemento de la lista y además información global de la lista.
- Cada nodo/elemento (común) almacena 2 cosas: su info y la referencia al elemento siguiente.

Es una lista lineal simplemente encadenada con header que además mantiene los elementos ordenados con algún criterio de ordenación. Hay 2 tipos de nodos: header y comunes. El nodo header no tiene que ser comparable, pues hay uno solo de ese tipo. Los nodos comunes tienen que poder compararse entre sí.

Análisis de posibilidades para “lista lineal simplemente encadenada con header para dar soporte a un índice” (análisis del peor caso, complejidad temporal)

	Búsqueda	Inserción	Borrado
Arreglo ordenado por clave de búsqueda	✓ $O(\log n)$	✗ $O(n)$	✗ $O(n)$
Lista Lineal Simplemente Encadenada Ordenada con Header	✗ $O(n)$	✗ $O(n)$	✗ $O(n)$

Para esto, vemos que no parecería haber una ventaja de la lista (que sí era clara en el caso que no hacía falta la búsqueda, como queue y stack).

Para borrar/insertar un elemento hay 2 situaciones que se dan:

- a) borrar/insertar un elemento recorriendo desde el header
- b) borrar/insertar un elemento sin buscarlo (estamos apuntando al elemento que queremos borrar, por ejemplo, haciendo uso del iterador)

En el arreglo ordenado, esa operación es  $O(n)$ , porque en el “caso a” tenemos que la búsqueda se hace en  $O(\log n)$ , pero todo empeora debido al movimiento de datos para garantizar contigüidad lo que se hace en  $O(n)$ . Análogamente, en el “caso b” accederlo es  $O(1)$ , pero debido al movimiento de datos para garantizar contigüidad, termina siendo  $O(n)$ . O sea: siempre es  $O(n)$ .

En la lista lineal simplemente encadenada ordenada, la operación dependen si es “caso a” o “caso b”. En el “caso a” tenemos que la búsqueda se hace en  $O(n)$ . Aunque no hay movimiento de datos porque no se precisa contigüidad, la operación es  $O(n)$ . En cambio, en el “caso b” estamos parados en el elemento y, como no hay movimiento de datos para garantizar contigüidad, entonces sería  $O(1)$ . Para estar apuntando al elemento a borrar” vamos a colocar el `remove()` en el iterador.

Por eso Java tiene como “opcional” el método `remove()` en la interface “Iterator”. Si en vez de `remove()` de lista se usa `remove()` de operador, la complejidad es  $O(1)$ . Tiene algunas especificaciones:

- Remove de iterador tiene que invocarse luego de un `next()`.
- No se pueden invocar 2 `remove()` seguidos (tiene que haber un `next()` en el medio).
- Si no se satisfacen esas condiciones se lanza la excepción “`IllegalStateException`”.

Pero el remove() de tiene cosas que no pueden chequearse y pueden producir un problema en tiempo de ejecución: uso de cursores anidados, donde uno de ellos elimina un elemento (el otro que había chequeado que había elementos obtiene un error porque el elemento ya no está). Por ejemplo:

```
public static void main(String[] args) {

    SortedListService<Integer> a= new SortedLinkedListWithHeaderAllowsRemoves<>();
    a.add(10);

    for (Iterator<Integer> iter1 = a.iterator(); iter1.hasNext();) {
        Integer nro1;
        nro1= iter1.next();
        Iterator<Integer> iter2 = a.iterator();
        if ( iter2.hasNext() )
        {
            iter1.remove();
            Integer nro2 = iter2.next();
        }
    }
}
```

Vemos que hay dos iteradores donde uno borra, y el otro queda apuntando a un elemento que ya no existe. Otro caso:

```
public static void main(String[] args) {

    SortedListService<Integer> a= new SortedLinkedListWithHeaderAllowsRemoves<>();
    a.add(10);

    for (Iterator<Integer> iter1 = a.iterator(); iter1.hasNext();) {
        Integer nro1;
        nro1= iter1.next();
        Iterator<Integer> iter2 = a.iterator();
        if ( iter2.hasNext() )
        {
            iter1.remove();
            Integer nro2 = iter2.next();
            iter2.remove();
        }
    }
}
```

Es importante además notar que el remove del iterador no puede utilizar el remove de la lista, porque ahí debería volver a recorrer y no sería O(1).

Entonces, ahora tenemos:

	Búsqueda	Inserción desde el header	Inserción desde iterador	Borrado desde el header	Borrado desde iterador
Arreglo ordenado por clave de búsqueda	✓ O(log n)	✗ O(n)	✗ O(n)	✗ O(n)	✗ O(n)
Lista lineal simplemente encadenada ordenada por clave de búsqueda	✗ O(n)	✗ O(n)	✓ O(1)	✗ O(n)	O(1)

*Lista Lineal Dblemente Encadenada con Header:* Es una estructura de datos compuesta por un elemento distinguido llamado “header” que tiene la referencia del primer elemento y además información global de la lista. Cada nodo/elemento (común) almacena 3 cosas: su info y la referencia a los elementos previo y siguiente.

*Lista Lineal Dblemente Encadenada Ordenada con Header:* Es una lista lineal doblemente encadenada con header que además mantiene los elementos ordenados con algún criterio de ordenación.

- Hay 2 tipos de nodos: header y comunes.
- El nodo header no tiene que ser comparable. Hay uno solo de ese tipo de nodo!
- Los nodos comunes tienen que poder compararse entre sí.

## Unidad 4 - Hashing (Dispersión)

Cada vez que había que hacer un lookup rápido de alguna componente, surgió la idea de que una tabla de Hashing podía ser una estrategia superadora frente a usar un Arreglo ordenado o Lista ordenada.

Ej: problemas de informática o matemática

- Representar la clase Bag
- Representar la clase Set

Para cualquiera de esos 2 escenarios un hashing parece ser una buena elección. Otro ejemplo es shazam, que divide por ciertas características partes del sonido para luego buscarlos.

Veamos en qué casos sería malo:

```
public interface IndexParametricService <T extends Comparable<? super T>>{  
    void initialize(T [] elements);  
    boolean search(T key);  
    void insert(T key);  
    void delete(T key);  
    int occurrences(T key);  
    T[] range(T leftKey, T rightKey, boolean leftIncluded, boolean rightIncluded);  
    void sortedPrint();  
    T getMax();  
    T getMin();  
}
```

Para los últimos 4 va a ser malo, es decir, todos los que impliquen orden.

### Tabla de Hashing

Primero, notemos que el hashing es un manejo de **arreglos**. Es decir, va a ser un arreglo. Además, la estructura (el lugar que ocupa una componente) no depende del orden en que llegaron las demás componentes.

Ej: inserto 10, 20, 30, 40, 50

Ej: inserto 50, 40, 30, 20, 10

Pero la posición del elemento es fuertemente dependiente de los valores de las otras componentes.

Ej: inserto 90, 40, 100, 120, 130

Por eso es que se precisa "buscar" el elemento =>  $O(\log N)$

Tabla de Hashing (hashing a secas) es una estructura de datos que utiliza un **arreglo** (lookup table) para almacenar pares key/value de una forma muy especial. No mantiene ni contigüidad, ni orden de las componentes.

Si  $\Omega$  es el universo de los ítems que queremos almacenar y tenemos un arreglo Lookup subyacente, entonces, utiliza una función de hashing hash:  $\Omega \rightarrow [0, |\text{Lookup}| - 1]$  (donde  $|\text{Lookup}|$  es la cantidad de ranuras). Dado un conjunto de valores posibles, nos da una ranura donde debe insertarse el elemento idealmente (hay que ver si hay lugar). Prioriza la búsqueda, tratando de que el algoritmo tenga complejidad cercana a  $O(1)$  en la búsqueda.

Primer escenario: suficiente espacio, o sea  $|\Omega| \ll |\text{Lookup}|$ . Sabemos que los arreglos tienen el problema de la alocación del espacio y hay que re-alocar cuando el espacio es insuficiente. En una primera aproximación vamos a suponer que hay espacio suficiente para almacenar los pares key/value.

Sea  $\Omega$  el conjunto de claves a hashear, y sea LookUp el arreglo para albergar los pares key/value. Asumimos  $|\Omega| \leq |\text{Lookup}|$ , es decir, hay posibilidad de que a cada key se le asigne una ranura de LookUp. Tenemos definido hash:  $\Omega \rightarrow [0, |\text{Lookup}| - 1]$ . Idealmente encontraríamos al valor asociado a un determinado key en  $\text{LookUp}[\text{hash}(\text{key})]$ .

### Prehash

Como los keys que proporciona el usuario son tipos opacos (TAD, Object, etc.) es bueno solicitarle que proporcione una función de prehash:  $\Omega \rightarrow N$  (números naturales). Nosotros debemos garantizar que hash:  $\Omega \rightarrow [0, |\text{Lookup}| - 1]$ , porque la ranura tiene que ser válida. Entonces hacemos:

$$\text{hash}(\text{key}) = \text{prehash}(\text{key}) \% |\text{Lookup}|.$$

Como hashing no resulta bien para operaciones típicas como sortedPrint(), min(), max(), range(), definimos una nueva interfaz para índices que precisen operaciones solo lookup.

```

public interface IndexParametricService<K, V> {

    // no acepta key ni data null=> lanza exception. Si el key está, realizar un update en el valor.
    // Si no existia lo inserta. Si hace falta crece de a chunks
    void insertOrUpdate(K key, V data);

    // nunca nunca nunca debe tirar exception. Devuelve el valor asociado si lo encuentra o null si no está.
    V find(K data);

    // nunca nunca nunca debe tirar exception.
    // Borra y devuelve true si el elemento estaba. Si no lo encuentra devuelve false .
    boolean remove(K key);

    // nunca nunca nunca debe tirar exception. Devuelve la cantidad de elementos presentes
    int size();

    // imprimir en cualquier orden
    void dump();
}

```

Ejemplo de uso: En un escenario ideal, si los pares key/value a insertar fueran (55, "Ana"), (44, "Juan"), (18, "Paula"), (19, "Lucas"), (21, "Sol") y considerando un arreglo de 10 componentes que es capaz de albergar a dichos pares, testear el código.

0	
1	(21, 'Sol')
2	
3	
4	(44, 'Juan')
5	(55, 'Ana')
6	
7	
8	(18, 'Paula')
9	(19, 'Lucas')

En un escenario no tan ideal, si los pares key/value a insertar fueran (55, "Ana"), (29, "Victor"), (25, "Tomas"), (19, "Lucas"), (21, "Sol") y se re ejecuta el código, se obtiene:

0	
1	(21, 'Sol')
2	
3	
4	
5	(25, 'Tomas') Se perdió (55, 'Ana')
6	
7	
8	
9	(19, 'Lucas') Se perdió (29, 'Victor')

**Colisión:** Se dice que 2 claves key1 <> key2 colisionan si hash(key1) = hash(key2), es decir, se les asigna la misma ranura.

**Función hash perfecta:** Se dice que una fn de hash es Perfecta si no produce colisiones. Es decir, si  $\text{key1} \neq \text{key2} \Rightarrow \text{hash}(\text{key1}) \neq \text{hash}(\text{key2})$ . Sería una función inyectiva.

Una función hash perfecta no es fácil de encontrar. Además se tiene un universo de claves posibles (aunque no se las precise hashear a todas) y por lo tanto, para garantizar que nunca van a colisionar habría que conocer mucho sobre la forma de los keys.

Aunque no podamos tener un hashing perfecto, el objetivo es que sea lo mejor posible, es decir, minimizar la cantidad de colisiones. Más allá de todo esto, en algún momento el Lookup puede quedarse sin lugar y hay que incrementar su tamaño (de a chunks) y rehashear.

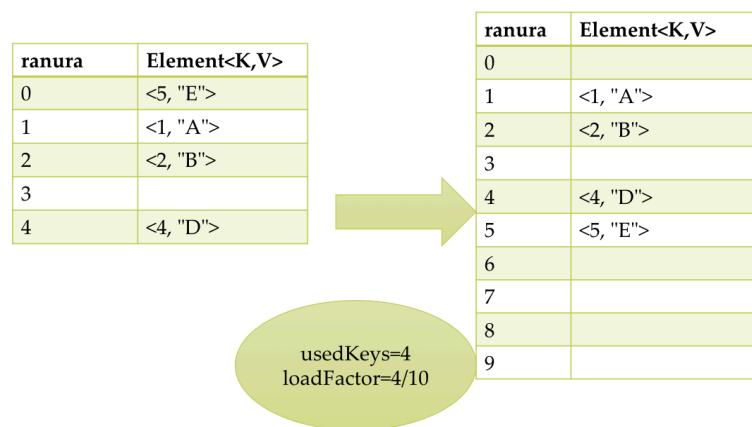
### Factor de Carga

Consiste en  $| \text{Keys usadas} | / | \text{Lookup} |$ , es decir que es nos dice qué “porcentaje” está ocupado.

El algoritmo de inserción provisorio (porque todavía no manejamos colisiones) consiste en ir a la ranura correspondiente y proceder según el caso:

- Si está ocupada por el mismo key => hacemos un update.
- Si está ocupada por otra key => es inserción con colisión. Esta inserción no es exitosa porque todavía no manejamos colisiones, por lo tanto, provisoriamente lanzamos una excepción.
- Si está vacía => es inserción exitosa. Primero inserta allí. Luego, chequea si el factor de carga supera un cierto umbral predefinido (Load Factor Threshold) y si eso ocurre se duplica el espacio y se rehashean todas las claves.

Ejemplo: supongamos que tomamos un threshold de 0,75. En el siguiente caso, ya tenemos 4 de 5, que es superior, por lo que debemos hacer un rehash. La clave 5 que estaba en 0, ahora va a ir a parar a la ranura 5.



Este resize y rehash le agrega mucha complejidad, es una de sus principales debilidades.

## Colisiones

Existen 2 formas de resolver las colisiones:

- Open Addressing or Closed Hashing: dentro de la misma tabla de hashing se guardan los elementos que colisionaron.
- Open Hashing or Closed Addressing or Chaining: fuera del hashing se almacenan los elementos que colisionaron.

**Cuidado**, están como “cruzados” los nombres.

### Open Addressing or Closed Hashing

Cada ranura puede tener null (está vacío o baja física). Aunque la ranura no esté vacía puede ser que el elemento no esté, ya que hay que manejar el concepto de bajas lógicas (además de las físicas). Es decir, una ranura representa 3 estados: tiene un elemento o no tiene un elemento (dado por baja lógica o bien por baja física).

Típicas formas de resolver esa colisión:

*Rehasheo Lineal* (linear probing). Si hay colisión en la ranura  $i$ , entonces intentar con la ranura  $i+1$ , y así siguiendo hasta encontrar que el elemento (se hace update) o encontrar un lugar vacío (baja física) y se inserta allí. Con esta técnica si hay lugar lo encuentra seguro.

Por ejemplo, si cae en ranura 4 y está ocupado va a intentar ranura 4+1, luego ranura 4+1+1, luego ranura 4+1+1+1, etc. Se suele tratar al arreglo como una lista circular. Para volver al principio usamos módulo (sumar uno y dividir por el tamaño del arreglo), NO un if.

*Rehasaheo Cuadrático* (quadratic probing). El intervalo entre ranuras a usar, si hubiera colisión, será cuadrática. Ej: si le toca la ranura 4 y está ocupado, va intentar la ranura  $4+1^2$ , luego  $4+2^2$ , luego  $4+3^2$ , etc. Problema: podría haber lugar y no lo encuentra.

Otra combinación predeterminada (determinística) de  $f_n$ : siempre conviene que la última sea rehasheo lineal.

**Borrado**: No se puede reemplazar al lugar borrado por una ranura vacía porque la búsqueda de alguna clave puede necesitar "pasar sobre ella" si hubo colisión. Se debe manejar dos tipos de borrado: físico (realmente se elimina el elemento) y lógico (se lo marca como que no está, y si más tarde hay que insertar en esa ranura se la puede aprovechar).

- El borrado físico se usa cuando la ranura que le sigue (la que se obtiene al aplicar hash  $i+1$ ) está también borrada físicamente.

- El borrado lógico se lo usa en caso contrario, o sea cuando la ranura que le sigue está ocupada o bien borrada lógicamente.

**Inserción:** Si la ranura calculada está marcada como baja física, el elemento se inserta allí. Caso contrario (está ocupado y no es el elemento a insertar, o bien está marcado como baja lógica) hay que comenzar a navegar con las sucesivas celdas hasta encontrar la primera baja física (o el error porque el elemento ya existía). Atención que no se puede detenerse en la primera baja lógica y pretender insertarlo allí porque justamente puede estar más adelante. Una vez que se encuentra la primera baja física se lo puede insertar allí o en alguna de las bajas lógicas halladas en ese trayecto.

Ejemplo:

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- **myLookUp.insert(15, "Meg");**
- myLookUp.delete(23); //Joe
- myLookUp.delete(15); //Meg
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> notdeleted
5	<4, "Sue"> notdeleted
6	<15, "Meg"> notdeleted
7	
8	
9	

Vemos que para los inserts, sacando el primer elemento todos quedaron “desplazados” un lugar de donde deberían ir realmente.

- myLookUp.insert(3, "Dick");
- myLookUp.insert(23, "Joe");
- myLookUp.insert(4, "Sue");
- myLookUp.insert(15, "Meg");
- myLookUp.delete(23); //Joe
- **myLookUp.delete(15); //Meg**
- myLookUp.insert(4, "Sue");
- myLookUp.insert(43, "Paul");

ranura	Element<K,V>
0	
1	
2	
3	<3, "Dick"> notdeleted
4	<23, "Joe"> deleted
5	<4, "Sue"> notdeleted
6	
7	
8	
9	

Ahora, para los delete vemos que para el 23, el siguiente está ocupado, por lo que hacemos baja lógica (deleted). Para el 15, el siguiente no está ocupado, por lo que hace baja física (borra).

La inserción de Sue, no puede ser en 4, porque hay baja lógica, entonces tiene que ser en el siguiente que es el mismo (update). Para Paul sucede algo similar, pero en la posición 3. Ya está ocupada, por lo que debería ir al siguiente donde hay una baja lógica. Se guarda esta posición y sigue avanzando hasta que encuentra la baja física.

Si se supera el threshold, hay que rehashear. Cuando hacemos esto, NO llevamos las bajas lógicas.

Con todo esto, podemos ver la en la **búsqueda** que se comienza buscando la clave en la ranura calculada. Si el lugar está con baja física seguro que no está en otro lado. Caso contrario si está marcado como ocupado y coincide con el valor esperado, se ha encontrado. Pero si el lugar está ocupado y no es el elemento buscado o bien está como baja lógica, no se sabe si va aparecer más adelante (en la aplicación de las sucesivas funciones de hashing). O sea que en ese caso hay que seguir buscando hasta encontrarlo (hallar una ranura ocupada que coincida con el elemento) o bien hallar una baja física.

Linear Hashing es muy eficiente para implementar la resolución de colisiones (aprovecha la localidad de los componentes => elementos cercanos).

Si hay lugar lo encuentra seguro. La desventaja se presenta cuando el factor de carga es alto.

### Open Hashing o Closed Addressing o Chaining

Las colisiones se resuelven en una estructura auxiliar (lista lineal, etc). Cada ranura puede tener null, o bien una estructura auxiliar con las componentes que colisionaron en dicha ranura (zona overflow). La zona de overflow se administra a demanda (no a priori).

**Remove:** Si la ranura está en null (sin zona de overflow habilitada), el elemento no existía. Si hay zona overflow, se lo navega para borrarlo (si estuviera). Si luego del borrado se obtiene una zona de overflow innecesaria, entonces la ranura vuelve a null. Es una operación destructiva.

**Find:** Si la ranura está en null, el elemento no está. Si hay zona de overflow, se lo navega allí para ver si se lo encuentra. Operación read-only.

**InsertOrUpdate:** Si se le asigna una ranura vacía, se habilita la zona de overflow. La inserción se hace en la zona de overflow correspondiente, si es que el elemento no estaba, caso contrario es un update. Es una operación destructiva.

En un hashing, no hay orden de los elementos. No es razonable pedir que el “key” sea comparable para que pueda ordenarse. No vamos a usar una lista ordenada simplemente encadenada, sino una lista. Pueden usar la que viene con Java: `LinkedList`.

Es importante redefinir `equals()`, pues sino la lista no permitirá updates porque siempre considerará que los elementos son diferentes. Manejar una lista en zona de overflow puede generar una complejidad mucho mayor que 1 en inserción/búsqueda/update/remove.

Para acercar a orden 1, vamos a tomar un factor de carga para el chaining, y así lograr que no se “sobrecargue” una ranura. Vamos a tomar un factor local

Factor de carga global > threshold => duplica tabla y rehashea con la idea de acercarnos a O(1).

## Hashing de Java

Hashing es una colección que viene implementada en Java. Usa, al igual que como hicimos nosotros:

- Arreglo de ranuras con zona de overflow

```
transient Node<K,V>[] table;
```

- Manejo del espacio inicial y factor de carga

```
static final int DEFAULT_INITIAL_CAPACITY = 1 << 4; // aka 16

static final float DEFAULT_LOAD_FACTOR = 0.75f;
```

- Si se acaba el espacio, lo duplica

```
++modCount;
if (++size > threshold)
    resize();
```

- Tiene manejo de zona de overflow. Si no hay mucha ocupación en una ranura, entonces usa una lista:

```
static class Node<K,V> implements Map.Entry<K,V> {
    final int hash;
    final K key;
    V value;
    Node<K,V> next;

    Node(int hash, K key, V value, Node<K,V> next) {
        this.hash = hash;
        this.key = key;
        this.value = value;
        this.next = next;
    }
}
```

Pero si hay demasiados elementos allí, los transforma en un (red black tree) árbol:

```
static final class TreeNode<K,V> extends LinkedHashMap.Entry<K,V> {
    TreeNode<K,V> parent; // red-black tree links
    TreeNode<K,V> left;
    TreeNode<K,V> right;
    TreeNode<K,V> prev; // needed to unlink next upon deletion
    boolean red;
```

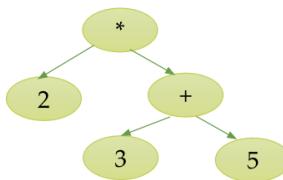
Utiliza el `HashCode()` para ordenar en el árbol.

## Unidad 5 - Árboles

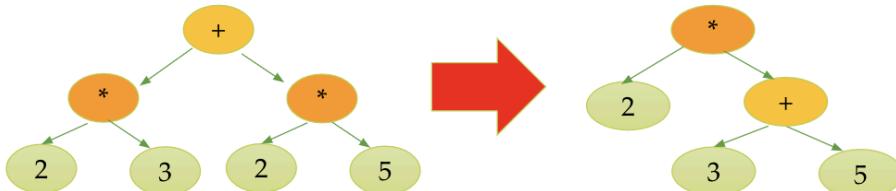
### Árbol Binario

Estructura de datos formada por nodos, donde cada nodo o está vacío o tiene 3 componentes: datos, subárbol izquierdo y subárbol derecho. Existe un nodo distinguido llamado raíz, por el cuál comenzamos la navegación. Es un caso especial de grafo, pero hay que tener en cuenta que los algoritmos de grafo son distintos a los de árbol.

Un caso de uso de árboles son los compiladores. Las expresiones formadas por operadores unarios/binarios pueden representarse con BT, donde las expresiones más anidadas se deberán evaluar primero.  $2 * (3 + 5)$



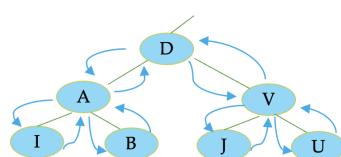
Además, se pueden usar estrategias para optimizar las expresiones a la hora de evaluarlas (re estructurar el BT).



Otra aplicación es cualquier cosa que tenga una representación jerárquica. Ej: jefe de en una organización. También, podrían usarse como soporte para índices.

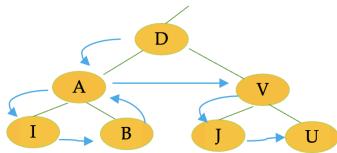
Recordemos:

- In-Order:



LVR

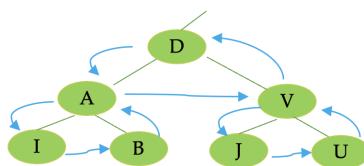
- Pre-Order:



D	A	I	B	V	J	U
---	---	---	---	---	---	---

VLR

- Post-Order:



I	B	A	J	U	V	D
---	---	---	---	---	---	---

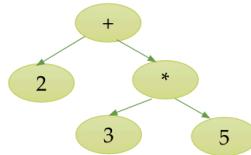
L R V

## Árbol Binario de Expresiones

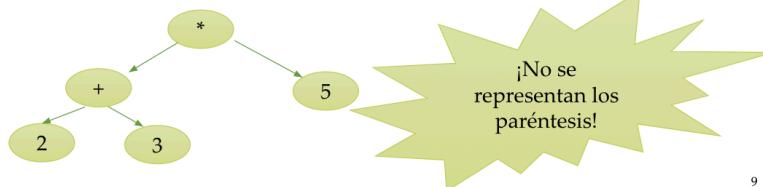
Se utiliza para representar expresiones algebraicas. Los nodos internos representan los operadores, binarios o unarios. Las hojas representan los operandos, es decir, constantes y variables. Según cómo se recorra el árbol (traversal) in-order, pre-order o post-order, se obtiene una expresión infija, prefija o postfija, respectivamente. Por lo tanto, permite representar expresiones en notación infija (no es un árbol ordenado por el contenido).

Así como usábamos una pila y una tabla de precedencia de operadores para pasar de una expresión en notación infija a postfija (para eliminar ambigüedad) y luego con una pila evaluábamos la expresión, ahora también a partir de una expresión, por ejemplo infija, construiremos el árbol de expresiones asociado y lo evaluaremos para devolver el valor de la expresión. Se le pedirá al usuario que use paréntesis con cada operando, aunque parezcan innecesarios, para no tener que utilizar la tabla y armar el árbol directamente. Por ejemplo

Ej: Este árbol representa la expresión infija ( 2 + (3 \* 5) )



Ej: este árbol representa la expresión infija ( (2 + 3) \* 5)



9

El desafío está en “saltar” los paréntesis, dado que no van en el árbol. Para el input vamos a pedir que finalice en \n. Los espacios serán los separadores de tokens.

Una expresión aritmética E está dada por las siguientes reglas de derivación:

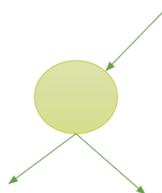
- E -> ( E + E )
- E -> ( E - E )
- E -> ( E \* E )
- E -> ( E / E )
- E -> ( E ^ E )
- E -> cte

Una regla de derivación dice: una variable (en este caso E) va a ser redefinida a partir de una de las anteriores reglas. Vemos que dentro de los paréntesis hay otras E, que son a su vez una expresión que debe expandirse (o también una constante como vemos en la última regla).

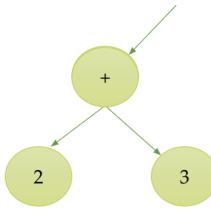
Las reglas se pueden resumir en 2:

- E -> ( E op E )
- E -> cte

Si por ejemplo nos llega “new ExpTree("( 2 + 3 ) \n);”, vemos que el primer token es un paréntesis, por lo que vamos a estar dentro de la primera regla. Cuando descartemos el mismo, vamos a tener que hacer una recursión.

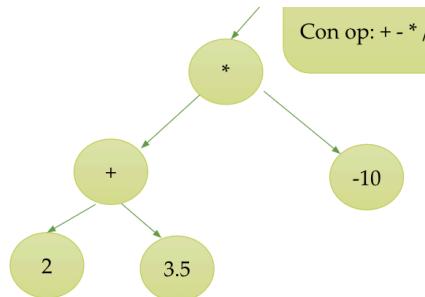


Cuando tenemos el 2, estamos en la segunda regla, por lo que se arma la hoja con la constante. Luego tenemos el token del + que debe ser un operando válido. La recursión debe armar en la rama derecha lo que tenga la expresión luego del +. Como es un 3, directamente se coloca como hoja. Nos queda:



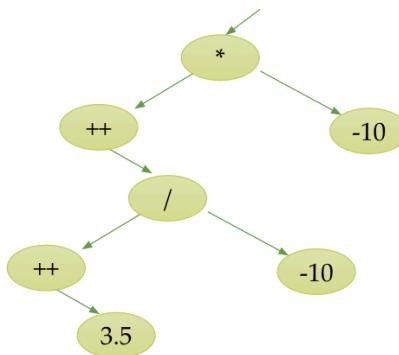
Un caso de error sería “new ExpTree“( ( 2 + 3 ) ) \n”); porque si bien va a armar correctamente el árbol para (2 + 3), cuando vuelva para armar la rama derecha no va a encontrar nada y fallará.

Tomemos ahora el caso “new ExpTree“( ( 2 + 3.5 ) \* ( -5 / -1 ) \n”);. Vamos armando el árbol recursivamente y vemos que finalmente nos queda:



Para resolver, vamos a aplicar la regla de delegación, es decir que todo se resuelve en el nodo.

El árbol de expresiones que vimos es especial porque todos los operadores son binarios. Un árbol de expresiones, pero con operadores unarios/binarios tendría otra forma.



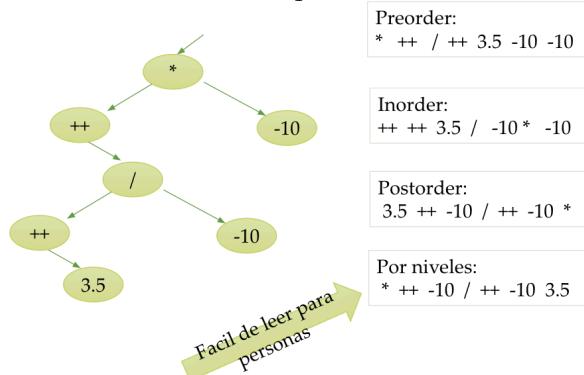
Si quisiéramos guardar sus datos en un archivo de texto, para luego (en otro momento) reconstruirlo desde el archivo de texto, algún recorrido vendría bien. Si quisiéramos hacerlo:

- Reconstruirlo leyéndolo.
- Sin haberlo construido nunca, editar un archivo, escribir la info con su estructura y leerlo leyéndolo

Un archivo de texto es una buena opción.

Se pueden generar convenciones para esto, pero hay una muy utilizada que consiste en guardar los datos “planos”, sin indicar a qué nodo corresponde la info. Claramente esa info está implícita en el archivo.

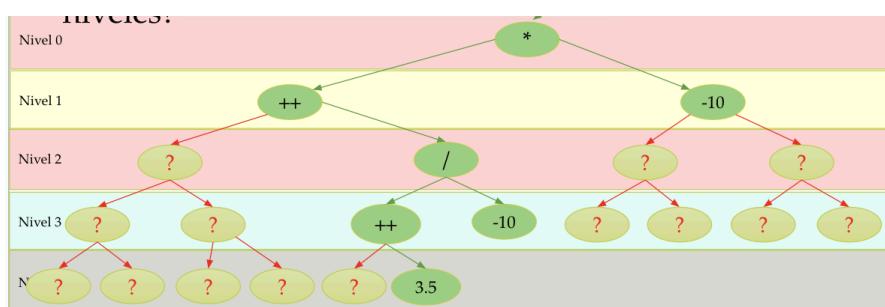
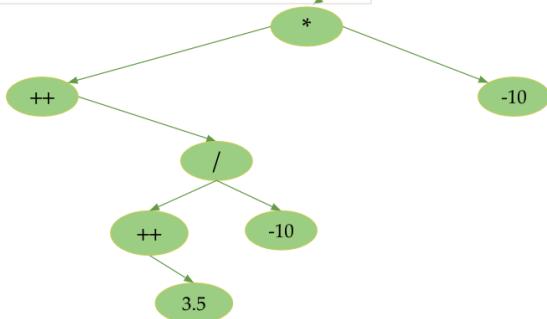
Almacenar su recorrido, podría servir.



Notemos que el por niveles es la manera más fácil de leer para nosotros. Tenemos que colocar “dummy” símbolos para completar los espacios. Debemos elegir algún símbolo “metadato” que no sea parte del lenguaje.

Ejemplo:

¿Cómo armaríamos el árbol si la expresión vienen por niveles? Por niveles: \* ++ -10 / ++ -10 3.5



¿Cómo armaríamos el árbol si la expresión viene por niveles? Generamos la raíz y postergamos qué hay que hacer con ella, hasta que llegue el token. Ni siquiera sé si tendrá cero, uno o dos hijos.



Pendientes:  
\$20 consumir

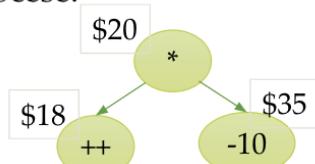
Ahora, estamos en el nivel 0. Debemos consumir la dirección \$20 y sacarlo de pendientes. Luego completo el dato. Como no sé si el \$20 tendrá cero, uno o dos hijos (depende del token), pido que cuando llegue el momento se los procese.



Pendientes:  
\$20 izq  
\$20 der

Completamos esto mismo para el nivel 1 y nos queda:

ocese.

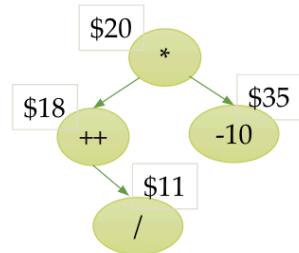


Pendientes:  
\$18 izq  
\$18 der  
\$35 izq  
\$35 der

Por niveles con placeholders:

*	++	-10	?	/	?	?	?	?	?	?	?	?	?	?	?	?	?	?	3.5
Nivel 1	Nivel 2				Nivel 3	Nivel 4 (posiblemente incompleto)													

Notemos que tenemos en pendientes los hijos izquierdo y derecho del ++ y del -10. Repetimos para el segundo nivel:



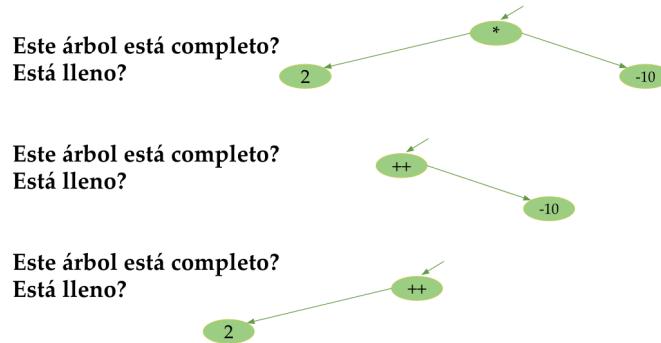
Pendientes:  
Null consumir  
Null consumir  
\$11 izq  
\$11 der  
Null consumir  
Null consumir  
Null consumir  
Null consumir

Vemos que para los dummies, pusimos a consumir null, porque no los incluiremos en nuestro árbol.

### Árbol Binario Completo

Un árbol binario es **completo** (complete) si todos los niveles, excepto posiblemente el último, tiene todos los nodos posibles y el último nivel tiene los nodos lo más a la izquierda posible. Un árbol binario está **lleno** (full) si todos los niveles, tiene todos los nodos posibles.

Por ejemplo:



El primer caso está completo y lleno, el segundo ninguno de los dos y el tercero completo pero no lleno.

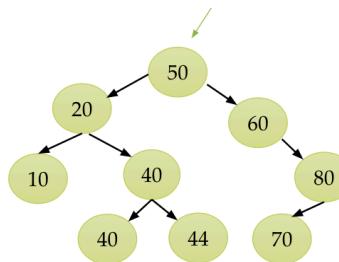
**Altura:** Longitud (cantidad de ejes) del camino más largo desde la raíz hacia las hojas. Un nodo formado solo por una raíz tiene altura 0. La altura del árbol vacío.

### Árbol de Búsqueda Binario

Los usos de los árboles son múltiples. Además de los árboles de expresiones, usar una estructura de árbol ordenada para buscar elementos suena interesante:

- De la lista toma lo mejor: encadenar los elementos con punteros y no tener que alojar zona contigua.
- De los arreglos ordenados toma lo mejor: la posibilidad de aplicar búsqueda binaria (es un árbol binario...)

El **Árbol Binario de Búsqueda** o **Binary Search Tree** (BST) es un árbol binario donde cada nodo no vacío cumple la siguiente condición: todos los datos de su subárbol izquierdo son menores o iguales que su dato, y todos los datos de su subárbol derecho son mayores que su dato.

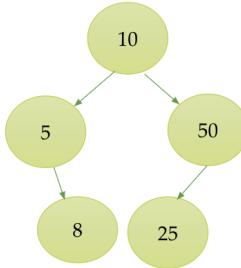


Vemos que siempre el nodo de la izquierda es menor o igual, mientras que el de la derecha es siempre mayor.

*Insertar en un BST:* Crece desde las hojas, es decir que para insertar vamos a crear un nodo que será una hoja. Es decir, que cuando insertemos siempre va a terminar siendo una hoja. Por ejemplo:

`myTree.insert(10); //Es el root`

```
myTree.insert(50);
myTree.insert(25);
myTree.insert(5);
myTree.insert(8);
```



Vemos que cada valor que se insertó, en su momento fue una hoja hasta que siguió la inserción.

*Borrar en un BST:* no basta con eliminar un elemento, se debe mantener la forma del original (no deformarse). El algoritmo es el siguiente:

- R1: Si el nodo a eliminar es hoja, actualizar quien lo apunta a él (puntero a raíz o su antecesor inmediato) para que ya no lo apunte más a él y pase a apuntar a NULL.
- R2: Si el nodo a eliminar tiene un solo hijo, actualizar quien lo apunta a él (puntero a raíz o su antecesor inmediato) para que en vez de apuntarlo a él lo haga al hijo del que se borra.
- R3: Si el nodo a eliminar tiene dos hijos se procede en dos pasos: primero se lo reemplaza por un nodo lexicográficamente adyacente (su predecesor inorder o sea el más grande de los nodos de su subárbol izquierdo, o bien su sucesor inorder o sea el más chico de los nodos de su subárbol derecho), y finalmente se borra al nodo que lo reemplazó (seguro que dicho nodo tiene a lo sumo un solo hijo, sino no sería el lexicográficamente adyacente, y por lo tanto es fácil de borrar).

**Problemas del BST:** El peor caso es cuando está totalmente desbalanceado. En este caso se pierde toda la ventaja de la búsqueda binaria, necesaria inclusive para la inserción/borrado/búsqueda en sí.

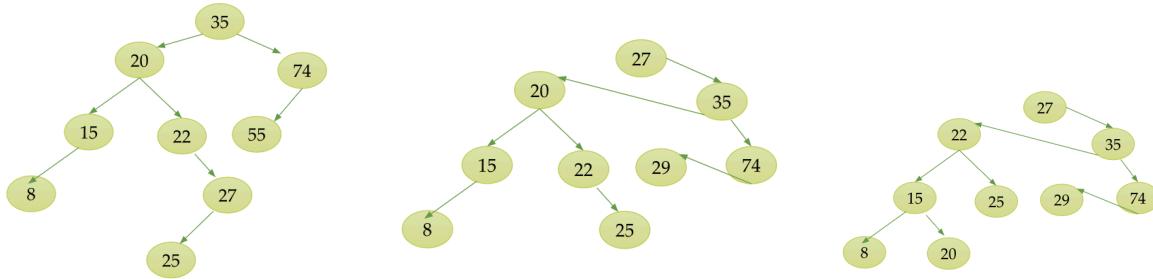
Si los elementos llegan en forma ordenada, se obtiene un árbol degenerado. Se obtiene algo que tiene la desventaja de la lista lineal en cuanto a cómo llegar a un elemento en particular, pero ocupando mucho más espacio.

En general, todas las operaciones son  $O(h)$  siendo  $h$  la altura del árbol, pero cuando está completo es  $O(\log N)$  y sino puede llegar a ser  $O(N)$ .

**Conclusión:** hay que tratar de que la altura del árbol sea la mínima posible. O, por lo menos, tenerla controlada.

AVL: “BST balanceado por altura”

Un AVL es un BST donde en cada nodo la diferencia de alturas entre sus 2 subárboles es a lo sumo 1.



El primer caso no es un AVL, mientras que el segundo y tercero sí. La diferencia es que el del medio no está completo, mientras que el último sí lo está.

El AVL es un árbol con buena forma. Las inserciones y borrados están definidas de manera tal que garantizan que se pueden realizar en  $O(\log N)$  y ser invariantes ante su propiedad (BST donde en cada nodo la diferencia de alturas entre sus subárboles es a lo sumo 1).

**Factor de Balance (fb):** Se calcula restando las alturas de cada subárbol de la forma:  

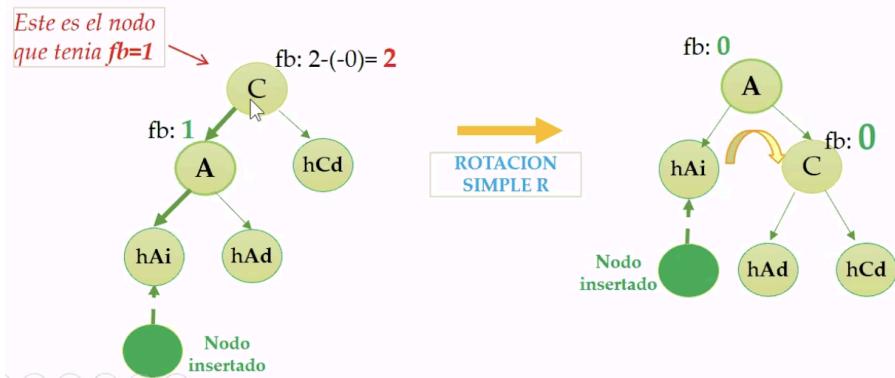
$$fb(\text{nodo}) = \text{altura}(\text{nodo.left}) - \text{altura}(\text{nodo.right})$$

#### *Inserción:* Operación Inserción

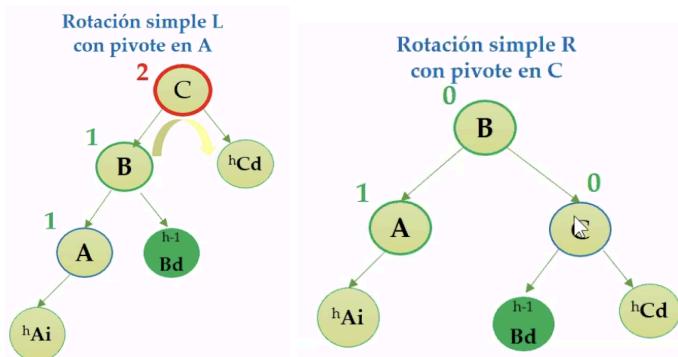
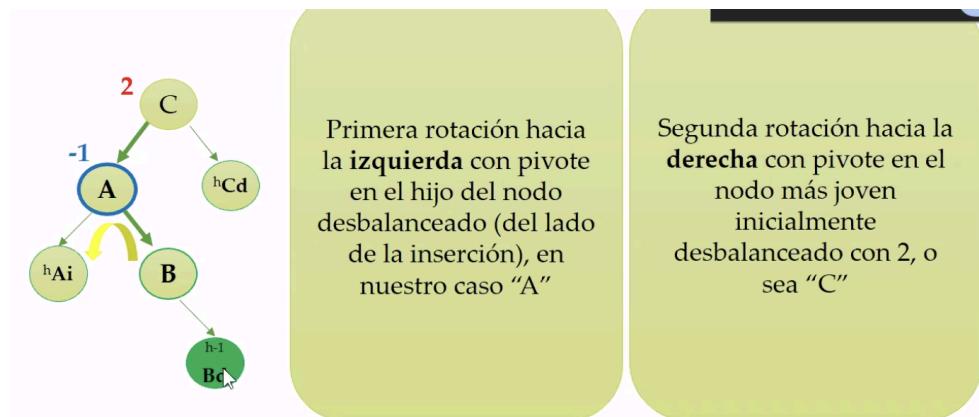
- Se inserta en el BST.
- Si está desbalanceado se aplican rotaciones para que siga siendo AVL => se rota el árbol con pivote más joven desbalanceado (más cercano al nodo insertado). Más precisamente:
  - Caso 1: si se inserta a la izquierda de un nodo con factor de balance 1, ese nodo va a tener factor de balance 2 y deja de ser AVL. Hay que rotar.
  - Caso 2: Si se inserta a la derecha de un nodo con factor de balance -1, ese nodo va a tener factor de balance -2 y deja de ser AVL. Hay que rotar.

**Rotación:** Transforma un árbol binario en otro, de tal manera que preserva el orden de sus elementos al navegar en inorder. Existen rotaciones simples y dobles.

- Caso A: Insertar a izquierda de un nodo con fb 1. Este nodo tendría fb 2 y dejaría de ser AVL. Tiene dos subcasos.
  - Caso A1- **Rotación simple a derecha (R):** cuando se inserta respecto del hijo izquierdo del nodo con factor de balance 1, en el subárbol izquierdo.



- Caso A2- Rotación doble izquierda a derecha (LR): cuando se inserta respecto del hijo izquierdo del nodo con factor de balance 1, en el subárbol derecho.

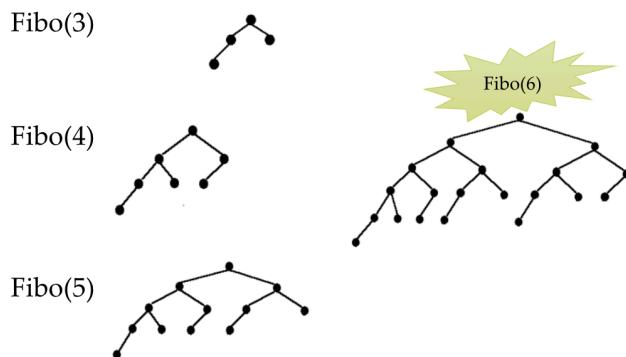


- Caso B: insertar a derecha de un nodo con  $fb = -1$ . Este nodo tendría  $fb = -2$  y dejaría de ser AVL. Tiene dos subcasos (cuyas soluciones son análogas a lo anterior - caso espejado).
  - Caso B1- Rotación simple a izquierda (L): cuando se inserta respecto del hijo derecho del nodo con factor de balance -1, en el subárbol derecho.
  - Caso B2- Rotación doble derecha a izquierda (RL): cuando se inserta respecto del hijo derecho del nodo con factor de balance -1, en el subárbol izquierdo .

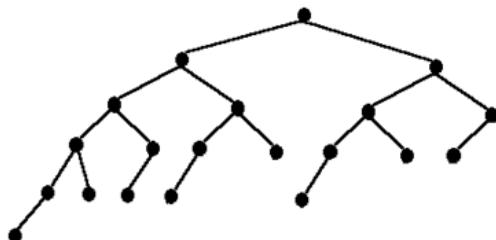
## Árbol de Fibonacci

Es el AVL que presenta el peor desbalanceo posible. Se define como:

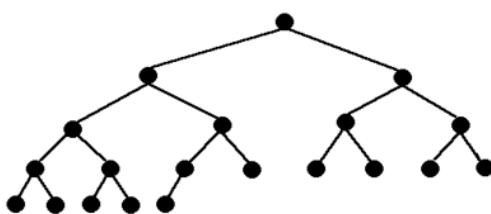
- Fibonacci de orden 0 es el ÁRBOL NULO.
- Fibonacci de orden 1 es un NODO.
- Fibonacci de orden  $h \geq 2$  es un árbol que tiene:
  - como hijo izquierdo un Fibonacci de orden  $h - 1$ .
  - como hijo derecho un Fibonacci de orden  $h - 2$ .



Para Fibo(6), vemos que:



Tiene 20 nodos y una altura de 5. Si ahora miramos la altura de uno perfectamente balanceado con la misma cantidad de nodos:



Vemos que tiene altura 4, sólo 1 menos que el Fibonacci. Entonces, podemos formalizar:

Sea un AVL de altura  $h$  con la menor cantidad de nodos posibles en esa altura.

AVL de altura 0 tiene 1 nodo



AVL de altura 1 tiene 2 nodos

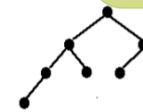


AVL de altura 2 tiene 4 nodos

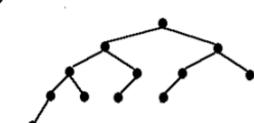


AVL de altura H tendrá  
1 nodo + cantnodos avl altura h-1 +  
cantnodos avl altura h-2

AVL de altura 3 tiene 7 nodos



AVL de altura 4 tiene 12 nodos



Entonces: AVL de altura h tendrá 1 nodo + cantidad nodos avl altura h-1 + cantidad nodos avl altura h-2.

¿Qué relación tiene con los números de Fibonacci?

AVL altura h con menor cant de nodos	Cant nodos en relación a los números de fibo?
Altura 0 CantNodos=1	nrofibo(altura+3)-1 = 1
Altura 1 CantNodos=2	nrofibo(altura+3)-1= 2
Altura 2 CantNodos=1 + 1 + 2 = 4	nrofibo(altura+3)-1= 4
Altura 3 CantNodos=1 + 4 + 2 = 7	nrofibo(altura+3)-1= 7
Altura 4 CantNodos=1 + 7 + 4 = 12	nrofibo(altura+3)-1= 12
Altura 5 CantNodos=1 + 12 + 7 = 20	
...	
Altura h CantNodos=?	nrofibo(h+3)-1

Números de fibonacci :

nrofibo(0)=0, nrofibo(1)=1, nrofibo(2)=1, nrofibo(3)=2, nrofibo(4)=3, nrofibo(5)=5, nrofibo(6)=8, nrofibo(7)=13, nrofibo(8)=21, etc

El árbol AVL que tiene la mínima cantidad de nodos para cierta altura es el Árbol de Fibonacci (es más esparcido posible por construcción). Si ese árbol con mínima cantidad de nodos y altura h sabemos tiene CantNodos= nrosFibo(h+3)-1, entonces, otros árboles AVL con misma altura tendrán posiblemente más nodos.

CantNodos  $\geq$  nrosFibo(h+3)-1

Pero los números de Fibonacci tienen una propiedad  $nroFibo(w) \geq \frac{a^w}{\sqrt{5}}$ , donde

$a = \frac{1+\sqrt{5}}{2}$  y se denomina *golden number*.

Entonces, sea n la cantidad de nodos de un AVL de altura h donde sabemos que  $n \geq \frac{a^{h+3}}{\sqrt{5}} - 1$ . Si despejamos h, obtenemos que es  $O(\log n)$ . Es decir que para un AVL de n nodos, la altura está acotada por  $O(\log n)$ .

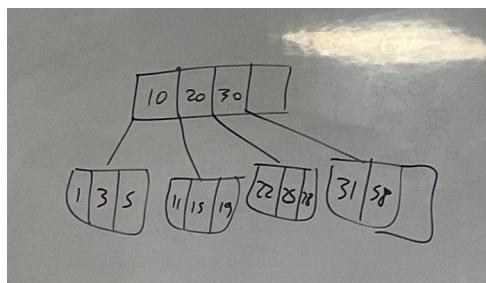
En conclusión, vimos que hay una diferencia poco significativa en altura, y como mantener AVL es menos costoso que un completo (perfectamente balanceado), va a ser preferible un árbol AVL.

Hay diferentes algoritmos para garantizar que un árbol sea balanceado y esta propiedad sea invariante ante inserciones/borrados. Algunos son más baratos que otros. Las transformaciones que hay que hacer para mantenerlo apropiado llegan en general  $O(\log n)$ , o sea, vale la pena hacerlas.

Tarea: Estudiar inserciones en Red-Black-Tree - IMPORTANTE - Entra al final

### Árbol Multicamino M-ario (orden M)

Los nodos guardan hasta  $M-1$  claves de información, con un máximo de  $M$  hijos. Cada clave  $C_i$  de un cierto nodo será tal que las claves almacenadas en su subárbol izquierdo serán menores y las almacenadas en su subárbol derecho serán mayores que él.



La clase sería, por ejemplo:

```
class Node{  
    int[] ...;  
    Node[] ... ;  
}
```

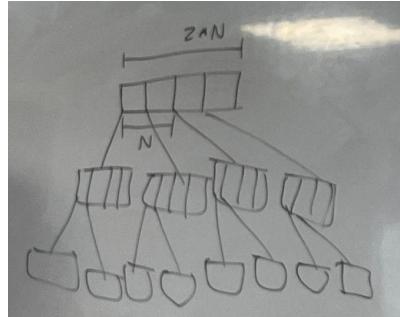
### Árboles Multicamino Balanceados

El equilibrio perfecto resulta muy costoso de mantener y es poco práctico. En 1970, R. Bayer y E. M. Mc Creight postularon un criterio razonable que permite implementar algoritmos relativamente sencillos para búsquedas, inserciones y eliminaciones. Para mantener éstos árboles multicaminos balanceados se utiliza una estructura subyacente de la familia de los árboles B.

Un árbol B de orden N es un árbol de búsqueda (ordenado) que cumple con los siguientes axiomas:

- Cada nodo contiene a lo sumo  $2 * N$  claves.
- Cada nodo, excepto la raíz, contiene por lo menos  $N$  claves. Entonces todos los nodos tienen entre  $2N$  y  $N$  claves.
- Cada nodo o es hoja o tiene  $M+1$  descendientes donde  $M$  es el número de claves que posee realmente ese nodo.

- Todas las hojas están al mismo nivel.
- En cuanto al orden: si un nodo tiene  $c_1 c_2 \dots c_m$  elementos  $c_1 < c_2 < \dots < c_m$ , pero además para cada  $c_i$  ( $1 \leq i \leq m$ ) los elementos del subárbol izquierdo de  $c_i$  son menores que  $c_i$  y los elementos del subárbol derecho de  $c_i$  son mayores que  $c_i$ .



*Algoritmo de Búsqueda:* Buscamos la clave  $X$  en un nodo. Para ello lo recorremos secuencialmente desde  $C_1$  hasta  $C_k$ , siendo  $k$  el número de claves que realmente posee dicho nodo, hasta que se den alguno de estos casos:

- Si  $X < C_1$ , como en el nodo las claves están ordenadas no tiene sentido seguir buscando en ese nodo, luego sigo buscando en el subárbol apuntado por  $P_0$
- Si  $X = C_i$  para algún  $i \leq k$  entonces lo encontré
- Si  $C_i < X < C_{i+1}$  para algún  $i$  prosigo en la búsqueda en el subárbol apuntado por  $P_i$
- Si  $C_k < X$  siendo  $k$  la cantidad de claves que posee, entonces sigo la búsqueda en el subárbol apuntado por  $C_k$

Si en algún caso el puntero por donde hay que seguir la búsqueda fuera null, entonces el elemento buscado no está.

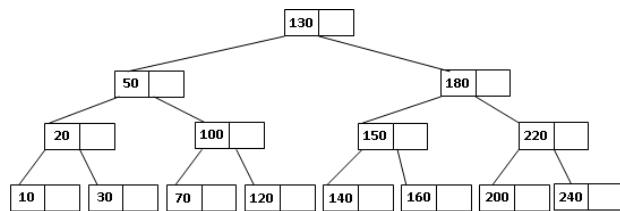
*Algoritmo de Inserción:* Si se quiere insertar una clave  $X$  en un árbol  $B$  de orden  $N$ , se procede de la siguiente manera:

- La inserción siempre se hace **en las hojas** (para poder detectar si el nodo a insertar ya está presente o no)
- Para insertar se coloca el elemento  $X$  en la hoja que corresponda (el nodo debe estar ordenado)
- Si el elemento nuevo hace que la cantidad nueva  $k$  sea mayor que el  $2^*n$  permitido, el nodo se abre en dos, subiendo la clave del medio al nodo antecesor de dicho nodo. Este algoritmo es recursivo hasta la raíz, o sea si al ubicar la clave del medio en el nodo antecesor ocasiona que el nodo viole la condición de árbol  $B$  de orden  $n$  ( $k > 2^*n$ ) el procedimiento de repite.

*Algoritmo de Borrado:* Si no encuentra en un nodo hoja, se lo reemplaza por una clave lexicográficamente adyacente, por ejemplo el sucesor in order y se lo elimina de dicha hoja. Si fuera hoja se lo elimina directamente.

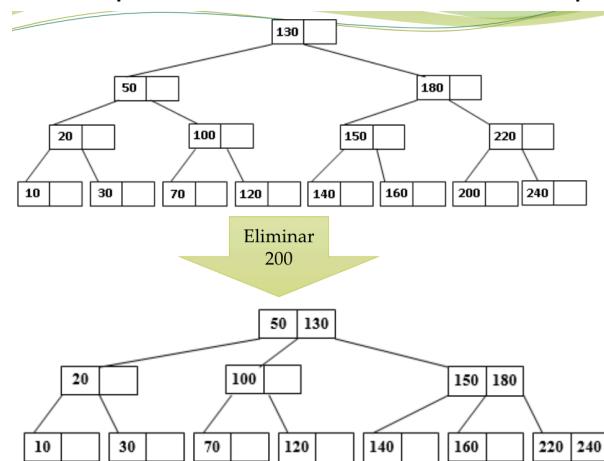
Luego, para la hoja que colaboró el borrado se analiza si cumple las condiciones de árbol B de orden N. Si ha quedado en rojo (tiene menos elementos que los permitidos), se une dicho nodo con su hermano y medio antecesor (el cual es eliminado del nodo al cual pertenece, porque acude en ayuda de su hijo) armando un sólo nodo. Se verifica si cumple las condiciones de árbol B de orden N, y sino se lo partitiona subiendo el elemento del medio. Después se analiza qué sucede con el nodo donde estaba su medio antecesor, y se sigue el proceso recurrentemente hasta llegar a la raíz.

Ejemplo: Dado el ultimo árbol de Orden 1 obtenido, eliminar 200, 220, 50.

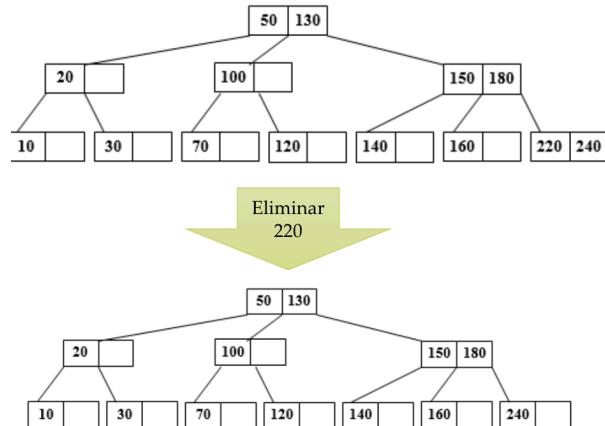


Eliminar 200: Como el nodo que contenía al 200 queda “en rojo”, se une con su hermano y baja su medio antecesor formando un solo nodo (220, 240), que se transforma en un nodo de árbol B de orden 1.

Pero el nodo donde estaba el medio antecesor 220 queda en rojo, entonces se une con su hermano y baja el medio antecesor formando un solo nodo (150, 180). ¿Es éste un nodo de un árbol B de orden 1? Sí, pero ¿qué pasa con el nodo donde estaba el medio antecesor? Queda en rojo. Entonces se une con su hermano y baja el medio antecesor formando un solo nodo (50, 130). Es éste un nodo de árbol B de orden 1? Sí. El árbol B, después de la eliminación del 200, queda así:

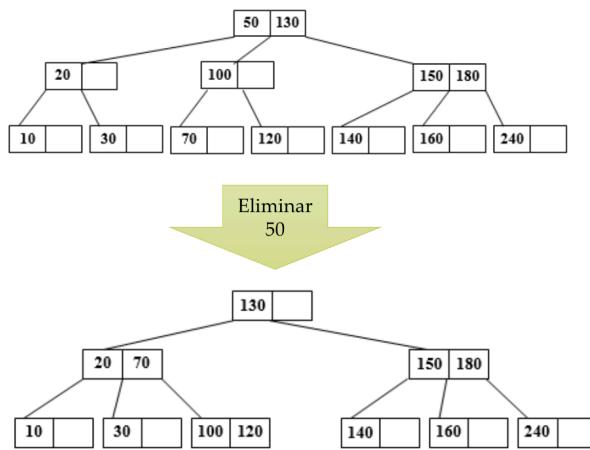


Eliminar 220:



Eliminar 50: Como no es un nodo hoja se lo reemplaza por una clave lexicográficamente adyacente, por ejemplo su sucesor in order: 70. Pero el nodo donde se encontraba la clave 70 es una hoja que queda en rojo, luego se une con su hermano y baja el medio antecesor formando un nodo (100, 120). Si bien ese nodo es correcto, el nodo donde estaba el 100 queda en rojo, luego, se une con su hermano y su medio antecesor.

Como tiene dos medio hermanos (20 y 150) hay que elegir una convención, por ejemplo, si tiene medio hermano izquierdo se toma ése, sino el de la derecha. No importa la convención utilizada, siempre que se la respete durante todo el algoritmo. Tomaremos esta convención, por lo cual queda el nodo (20, 70). ¿Es éste un nodo de un árbol B de orden 1? Sí. ¿Qué pasa con el nodo donde estaba el medio antecesor? Sigue siendo un nodo de un árbol B de orden.



## Eficiencia de los árboles

Aunque el código que analizamos para B orden N solo guarda claves, típicamente se define:

```
public class BTree<T extends Comparable<T>, V>
implements ITree<T, V> {...
```

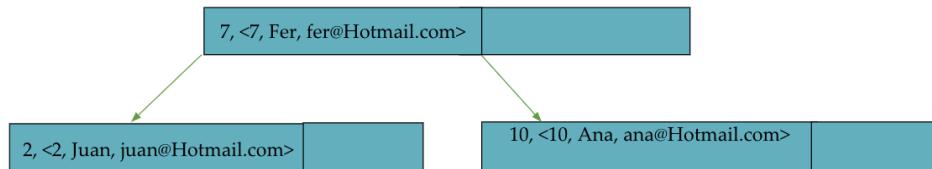
Donde V es para asociar al Key un Record. Ej: para record student: legajo, nombre, email.

<10, Ana, ana@Hotmail.com>

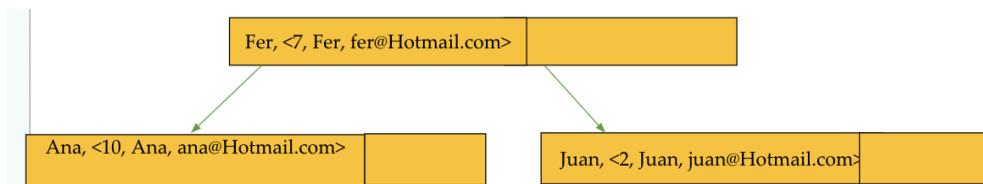
<2, Juan, juan@Hotmail.com>

<7, Fer, fer@Hotmail.com>

Si B ordena por legajo:



Si B ordena por nombre:



Si la colección tiene mucha información el Record se deja en disco. Se accede al archivo con RandomAccessFile (fixed length records) por su offset. El árbol quedaría:

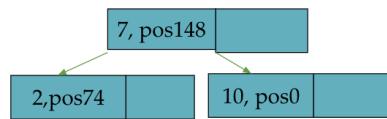
```

public class BTree<T extends Comparable<T>, V>
implements ITree<T, V> ...
    
```

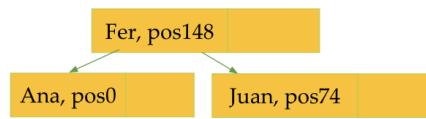
Donde V es el “index u offset” del archivo en disco. Ej: para record student: legajo, nombre, email. Todos los registros ocupan, por ejemplo, 4+50+20 bytes.



Si B ordena por legajo



Si B ordena por nombre



## Unidad 6 - Grafos

### Casos de Uso

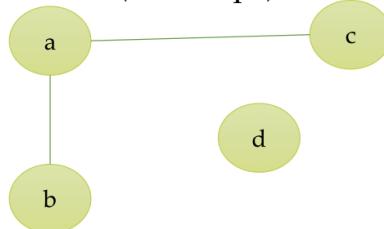
Los grafos tienen varios casos de uso:

- Flujo/Transporte: Para representar rutas, conexiones y tráfico. Ej: rutas áreas, formas alternativas para ir de un lugar a otro, etc.
- Redes sociales, y el análisis de la comunidad digital. Ej: 2500 usuarios de twitter y su interconexión. Dada la complejidad de las interrelaciones entre participantes, existe mucha información que puede extraerse al analizar este tipo de redes: el más influyente, las comunidades, recomendaciones, entre otros.
- Instaladores/Compiladores/Optimizadores: Precisan saber el orden conveniente en que hay que instalar/configurar/ejecutar paquetes. Ej: manejador de proyectos de software como maven. Con tantas dependencias, ¿En qué orden debe instalarse un paquete y todas sus dependencias? ¿Quién depende de quién?
- Al fabricar un elemento (ej: un auto). ¿Por dónde se empieza?
- Al armar un cronograma para cierta actividad, como ser ¿cuál es el plan para recibirse de Ing. en Informática en ITBA?

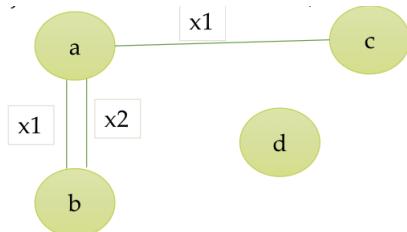
### Tipos de Grafos

Los distintos tipos de grafos con los que vamos a trabajar son:

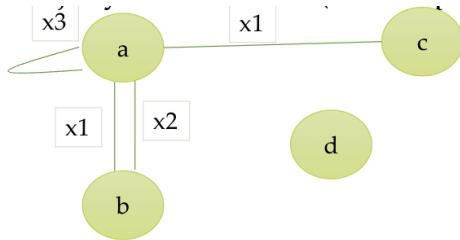
- Con ejes no dirigidos
  - Simple: entre cada par de nodos hay a lo sumo un eje. No admite lazos (self-loops).



- Multigrafo: entre cada par de nodos puede haber varios ejes. No admite lazos (self-loops). Notemos que se incluyen rótulos para caracterizar los ejes.



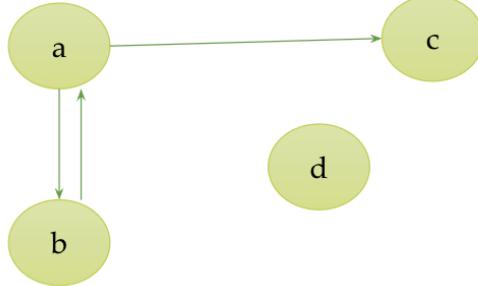
- Pseudografo: entre cada par de nodos puede haber varios ejes y admite lazos (self-loops).



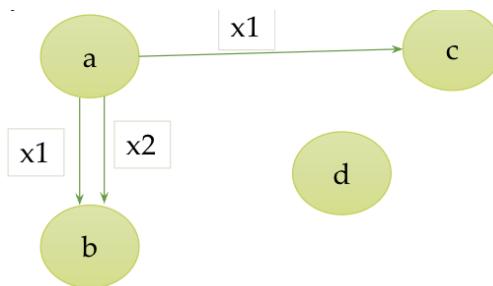
Notemos que no hay una categoría con los grafos simples pero que admiten lazos. Los matemáticos no lo nombraron.

- Con ejes dirigidos (digrafos)

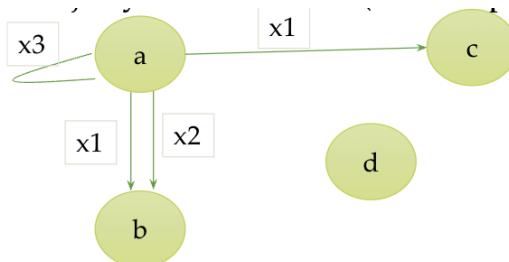
- Simple digrafo: entre cada par de nodos hay a lo sumo un eje. No admite lazos (self-loops). Vemos que hay dos ejes entre a y b, pero como son dirigidos y van en direcciones opuestas, cuentan como uno cada uno.



- Multi digrafo: entre cada par de nodos puede haber varios ejes. No admite lazos.



- Pseudo digrafo: entre cada par de nodos puede haber varios ejes y admite lazos (self-loops).



Nuevamente, notemos que no hay un nombre para los digrafos simples con lazos.

Depende de lo que se quiera modelar, vamos a tomar el tipo de grafo que más nos convenga.

Ej. Para rutas áreas, ¿hay algún vuelo que salga de una ciudad y llegue a la misma ciudad?

Rta: No. Entonces, no elegiría ninguno de los casos que acepten lazos.

Ej: Para correlatividades entre materias. ¿Puede una materia ser correlativa a ella misma? ¿Serviría no poner quien va antes que quien?

Rta: No. Entonces, no elegiría ninguno de los casos que acepten lazos. Tampoco elegiría uno no dirigido.

Además de todos los casos que expusimos tenemos las variantes donde los ejes aceptan pesos. Ej: en el caso de rutas áreas esto puede ser muy útil porque podría colocar en dichos pesos la duración del vuelo, etc.

Por lo tanto, tenemos los siguientes tipos de grafo:

Dirigido?	Multiplicidad?	Lazos?	Nombre
⊗	⊗	⊗	Simple
⊗	⊗	✓	Simple con lazos
⊗	✓	⊗	Multi Grafo
⊗	✓	✓	Pseudo Grafo
✓	⊗	⊗	(Simple) Digrafo
✓	⊗	✓	Digrafo con lazos
✓	✓	⊗	Multi Digrafo
✓	✓	✓	Pseudo digrafo

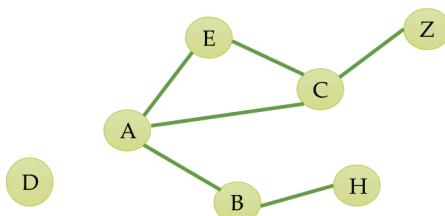
Al simple con lazos lo nombramos nosotros por lo que ya vimos antes. También se lo conoce como grafo default.

Si además de estas 8 combinaciones les permitimos manejar peso en los ejes => tenemos 16 TIPOS.

A veces, algunas definiciones/algoritmos dependen del tipo.

Ej: indegree y outdegree solo aplica a grafos dirigidos. Calcular el camino mínimo solo aplica a grafos con peso en los ejes, etc.

Veamos un caso de uso. Tomemos el árbol:



```

g.dump();

System.out.println(String.format("#vertices: %d", g.getVertices().size() ));

System.out.println(String.format("#edges: %d", g.getEdges().size() ));

// degree de cada nodo
for(Character aV: g.getVertices()) {
    System.out.println(
        String.format("vertex %s has degree %d", aV, g.degree(aV)));
}

```

Al imprimir obtenemos:

```

Vertexes:
(A) (B) (C) (D) (E) (H) (Z)
Edges:
(A) -- (B)
(A) -- (C)
(A) -- (E)
(B) -- (A)
(B) -- (H)
(C) -- (A)
(C) -- (E)
(C) -- (z)
(E) -- (A)
(E) -- (C)
(H) -- (B)
(Z) -- (C)

#vertices: 7
#edges: 12
vertex A has degree 3
vertex B has degree 2
vertex C has degree 3
vertex D has degree 0
vertex E has degree 2
vertex H has degree 1
vertex z has degree 1

```

## Grafo Simple

Sea  $G=(V, E)$ . Representamos a ambos conjuntos:

- 1) Vértices  $V$

E	B	A	F	D	G	U	T	C
---	---	---	---	---	---	---	---	---

- 2) Y hay cuatro propuestas típicas para representar a los ejes  $E$ :

- a) **Matriz de adyacencia.** Ideal grafos densos y estáticos, porque sino la tabla queda casi vacía.

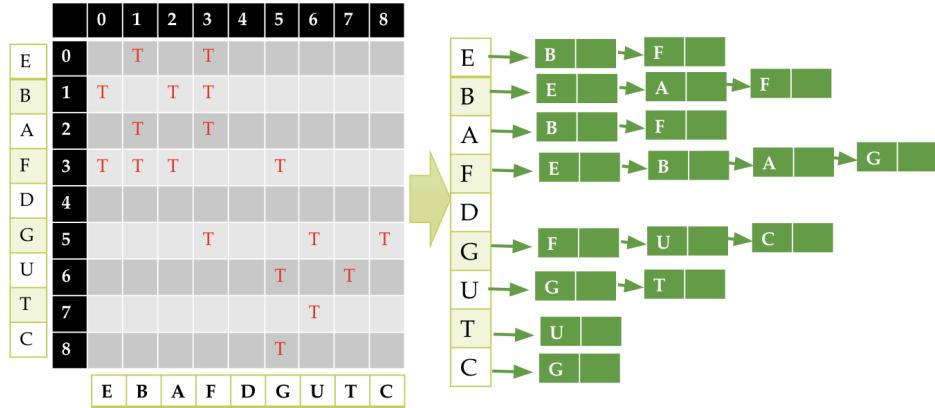
	0	1	2	3	4	5	6	7	8
E	0	F	T	F	T	F	F	F	F
B	1	T	F	T	T	F	F	F	F
A	2	F	T	F	T	F	F	F	F
F	3	T	T	T	F	F	T	F	F
D	4	F	F	F	F	F	F	F	F
G	5	F	F	F	T	F	F	T	F
U	6	F	F	F	F	F	T	F	F
T	7	F	F	F	F	F	T	F	F
C	8	F	F	F	F	F	T	F	F

E	B	A	F	D	G	U	T	C
---	---	---	---	---	---	---	---	---

Si me piden el degree de F, vamos al cuadro y recorremos toda la fila contando. Si quiero saber si tiene conexión con algún otro, vamos a la fila de F y la columna del otro valor.

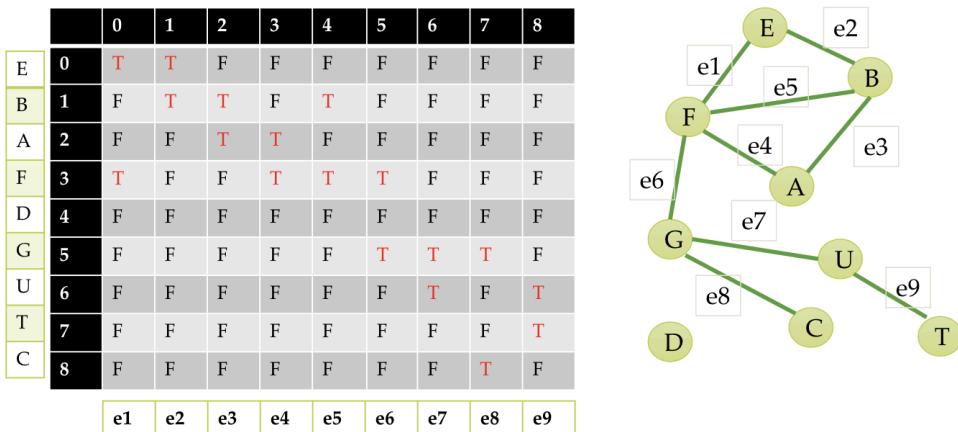
Con la potencia de la matriz, puedo buscar caminos de la longitud a la que esté elevada la misma.

- b) **Lista de adyacencia.** Ideal grafos esparcidos (sparse)



Vemos que se complica ver si hay dos conectados. Lo bueno es la versatilidad de agregar conexiones. Es la que nosotros vamos a implementar.

- c) **Matriz de incidencia.** Se colocan vértices y ejes en filas y columnas.



- d) **Lista de incidencia.** Misma lógica que la matriz. Se representa una lista asociada que compacta

## Generación de Grafos

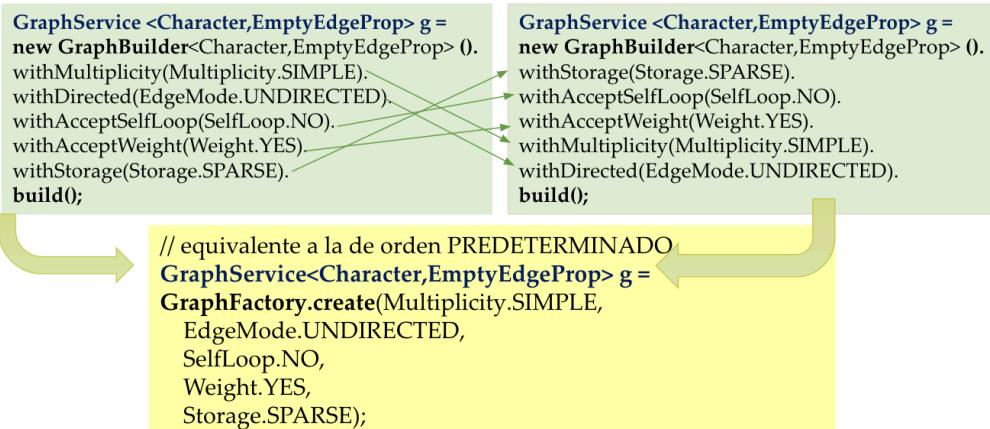
Queremos que el usuario genere 12 tipos de grafos posibles \* 2 (por el peso en los ejes) \* 4 (por las implementaciones posibles), pero no vamos a esperar que el usuario conozca el nombre de un montón de clases, para los casos. Una buena idea es escribir un Factory. Esta idea es la que implementa la clase JGraphT, una biblioteca de código abierto sobre grafos muy importante.

## Modo de Uso:

```
GraphService<Character,EmptyEdgeProp> g =
    GraphFactory.create(Multiplicity.SIMPLE,
        EdgeMode.UNDIRECTED,
        SelfLoop.NO,
        Weight.NO,
        Storage.SPARSE);
```

La clase GraphFractory es abstracta. Create es método static. No me da una instancia de GraphFactory. Me da una instancia de la clase que corresponda a través de un servicio. Vemos a su vez que se parametrizan las clases que representan las propiedades de los vértices y las propiedades de los ejes.

Otra buena idea es ofrecer una clase Builder que permita que en cualquier orden se seteen parámetros en forma aislada (si no se proporcionan asumen algún default) y cuando lo decida invoque build() que finalmente invoca al GraphFactory. Es un caso de postergación.



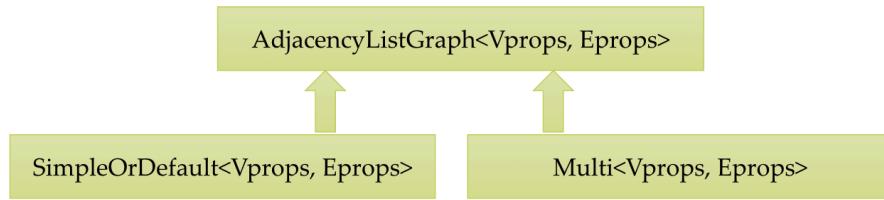
Vemos que lo puede crear con cualquier orden, y podemos setear los parámetros por default en caso de no ser definidos por el usuario. La clase builder no va a ser abstracta, va a ser transitoria.

GraphFactory (sea que lo invoca directamente el usuario o sea que se invoca a partir del GraphBuilder) genera una instancia de SimpleOrDefault y Multi. Ambas clases tienen mucho en común. Sin embargo, difieren en algo:

- addEdge: en el caso de SimpleOrDefault, si se vuelve a crear otro eje entre el mismo par de vértices, se ignora. En Multi no se ignora, se crea.
- removeEdge: en el caso de SimpleOrDefault, si se indica un par de vértices, con sus propiedades y se lo encuentra, se borra el único eje encontrado. En el caso de Multi se borran todas las apariciones de ese eje con mismas propiedades entre esos vértices.

Tip: si no se especifican properties se borra en el caso de SimpleOrDefault el único eje que pudiera existir entre dichos vértices. Si es Multi lanza exception (para evitar ambigüedad). Es responsabilidad del usuario definir equals/hash en la clase que represente las propiedades de los vértices y las propiedades de los ejes.

Como ambas clases tienen mucho en común, podemos hacerlas especializar de la clase abstracta `AdjacencyListGraph`:



Sea `SimpleOrDefault` o `Multi`, como cada vértice tiene una lista de adyacencia asociada, podemos armar un Map de vértice a su lista de adyacencia. Claro que esa “lista de adyacencia” será diferente si estamos con un `SimpleOrDefault` o `Multi`.

## Recorridos en Grafos

Recorridos (traversal) en grafos se usan para ver los nodos alcanzables a partir de un nodo y se muestra dicho camino. Aclaración: Si fuera un multigrafo (para que no haya ambigüedades) podría indicar cuál eje tomar en cada caso para la navegación. Igual, los vértices los sigo visitando una sola vez. Los métodos son:

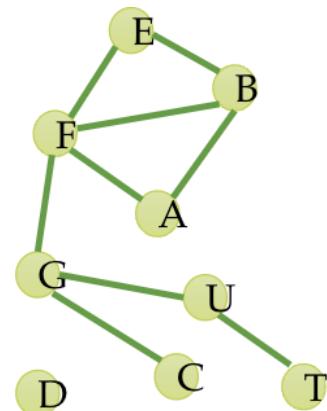
- Breadth-First Search (BFS)
- Depth-First Search (DFS)

BFS: Se visitan los vecinos de cada nodo. Hay que guardar los que ya se visitaron, porque se puede llegar a un mismo nodo por distintos caminos. Por ejemplo:

	0	1	2	3	4	5	6	7	8
E	F	T	F	T	F	F	F	F	F
B	T	F	T	T	F	F	F	F	F
A	F	T	F	T	F	F	F	F	F
F	T	T	T	F	F	T	F	F	F
D	F	F	F	F	F	F	F	F	F
G	F	F	F	T	F	F	T	F	T
U	F	F	F	F	F	T	F	T	F
T	F	F	F	F	F	F	T	F	F
C	F	F	F	F	F	T	F	F	F

E	B	A	F	D	G	U	T	C
---	---	---	---	---	---	---	---	---



Nos paramos en E:



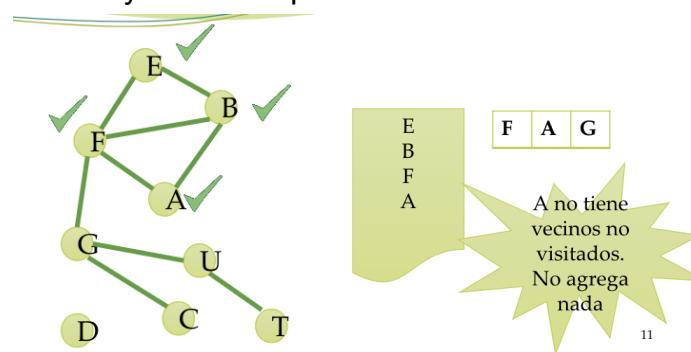
Visitamos B:



Ahora, nos movemos a F:



Ahora, nos movemos a A y notemos que no tenemos vecinos “no visitados”:



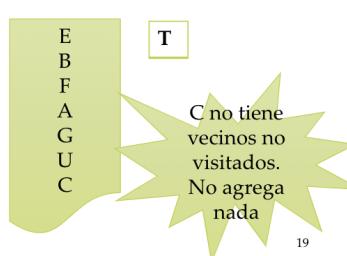
Entonces, F y A se ignoran porque ya fueron procesados. Vamos a G:



Luego, vamos a U:



Vamos a C, pero como no tiene vecinos no visitados, no agrega nada:



19

Lo mismo va a ocurrir con T. Por lo que se va a agregar a la lista a imprimir pero no va a agregar nada. Entonces, termina el algoritmo. Es similar al printByLevels que hicimos en árboles.

Este es printBFS(V) clásico. Tiene que servir para todo tipo de grafo. Notemos que como estructura auxiliar utiliza una queue. Como precisa marcar los vértices por los que pasó:

Opción 1: estructura paralela a los vértices booleana.

Opción 2: representar el vértice con un tag booleano.

DFS: Es similar al anterior, pero en vez de un queue se utiliza un stack. El algoritmo es muy similar.

### Algoritmo de Dijkstra

Retomamos el algoritmo visto en discreta. Tenemos el siguiente código:

```

While (! costosConocidos.isEmpty() ) {
    current= Sacar el de menor costo de costosConocidos.
    if ( Visited.contains( current.vertex)
        continue;

    Visisted.add( current.vertex);
    Foreach ( e in ejes incidentes de current )
        if ( e.target ya estaba en visitado)
            saltar;
        else
            si (suma costo de current + e.weight < costo e.target) {
                actualizar el costo de e.target;
                agregarloCostosConocidos(e.target, new Costo)
            }
}
}

```

Vamos a guardar cada nodo y el camino de menor peso a él. Para realizar el algoritmo, vamos a ir guardando también los visitados y los costos conocidos en cada paso, para así ir comparando y encontrar el camino de menor peso.

Inicialmente:

Dijkstra de C a los demás?

	A	B	C	D	E
Costo?	Inf	Inf	0	Inf	inf

```

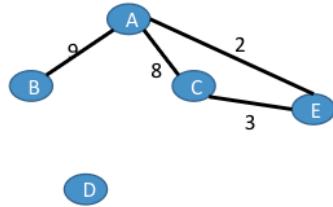
While (! costosConocidos.isEmpty() ) {
    current= Sacar el de menor costo de costosConocidos.
    if ( Visited.contains( current.vertex)
        continue;

    Visisted.add( current.vertex);
    Foreach ( e in ejes incidentes de current )
        if ( e.target ya estaba en visitado)
            saltar;
        else
            si (suma costo de current + e.weight < costo e.target) {
                actualizar el costo de e.target;
                agregarloCostosConocidos(e.target, new Costo)
            }
}
}

```

Visited = { }

costosConocidos= { (C,0) }



No tenemos ningún visitado y únicamente conocemos el peso de ir a C (donde comenzamos), que es cero. Ahora, vamos a tomar:

Dijkstra de C a los demás?

	A	B	C	D	E
Costo?	8	Inf	0	Inf	3

```

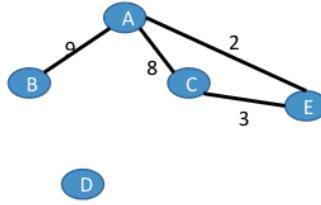
While (! costosConocidos.isEmpty() ) {
    current= Sacar el de menor costo de costosConocidos.
    if ( Visited.contains( current.vertex)
        continue;

    Visisted.add( current.vertex);
    Foreach ( e in ejes incidentes de current )
        if ( e.target ya estaba en visitado)
            saltar;
        else
            si (suma costo de current + e.weight < costo e.target) {
                actualizar el costo de e.target;
                agregarloCostosConocidos(e.target, new Costo)
            }
}
}

```

Visited = { C }

costosConocidos= { (E, 3), (A, 8) }



Como en los costos conocidos vemos que el menor es (E, 3), nos vamos a mover a él y lo vamos a marcar en visitados. Luego repetimos el proceso anterior.

Dijkstra de C a los demás?

	A	B	C	D	E
Costo?	5	Inf	0	Inf	3

```

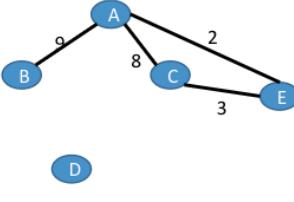
While (! costosConocidos.isEmpty() ) {
    current= Sacar el de menor costo de costosConocidos.
    if ( Visited.contains( current.vertex)
        continue;

    Visisted.add( current.vertex);
    Foreach ( e in ejes incidentes de current )
        if ( e.target ya estaba en visitado)
            saltar;
        else
            si (suma costo de current + e.weight < costo e.target) {
                actualizar el costo de e.target;
                agregarloCostosConocidos(e.target, new Costo)
            }
}
}

```

Visited = { C, E }

costosConocidos= { (A, 5), (A, 8) }



En E, como C está marcado como visitado lo vamos a descartar, por lo que únicamente vamos a tomar el camino de E a A. Vemos que la suma del anterior y el nuevo dan un camino de 5, que agregamos en los costos conocidos. Como 5 es menor a 8, vamos a pasar de E a A y lo vamos a marcar como visitado para repetir

el proceso. En A, solamente podremos ir a B, pues C y E ya están marcados como visitados. Por lo tanto, finalmente vamos a obtener:

Dijkstra de C a los demás?

	A	B	C	D	E
Costo?	5	14	0	Inf	3

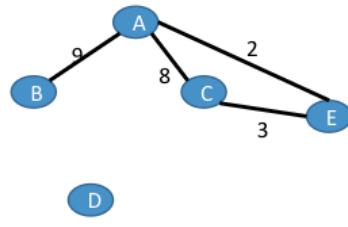
```

While (! costosConocidos.isEmpty() ) {
    current= Sacar el de menor costo de costosConocidos.
    if ( Visited.contains( current.vertex) )
        continue;
    Visisted.add( current.vertex);
    Foreach ( e in ejes incidentes de current )
        if ( e.target ya estaba en visitado)
            saltar;
        else
            si (suma costo de current + e.weight < costo e.target) {
                actualizar el costo de e.target;
                agregarloCostosConocidos(e.target, new Costo)
            }
}

```

Visited = { C, E, A, B }

costosConocidos= {}

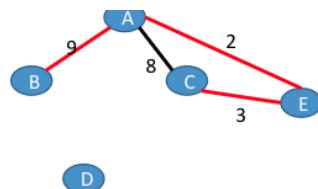


A este algoritmo, le podríamos agregar que se guarde además del costo (o peso), el nodo previo desde el que se llegó de la siguiente manera:

```

...
Foreach ( e in ejes incidentes de current )
    if ( e.target ya estaba en visitado)
        saltar;
    else
        si (suma costo de current + e.weight < costo e.target) {
            actualizar el costo de e.target;
            agregarloCostosConocidos(e.target, new Costo)
            prev.put( e.target, current.vertex); // hashing
        }
}

```



5: [C, E, A]
14: [C, E, A, B]
0: [C]
INF: []
3: [C, E]

Como sabemos el previo a cada nodo, podemos reconstruir el camino mediante el cuál se llegó.

**Importante:** Dijkstra tiene una precondition, y es que los ejes no pueden tener peso negativo. Esto rompe la suma y comparación para el algoritmo.

Otra forma de implementarlo correctamente (hay varias) consiste en:

En vez de agregar nodos posiblemente repetidos a la estructura: si el costo mejora **agregoOActualizo** (si el elemento no estaba en la estructura lo agrego, sino saco a ese y agrego). Así, la estructura nunca tiene más de  $|V|$  elementos.

Con esto, la complejidad sería:

Times

=

$$\sum_{u \in V} (Times(sacarMin) + \sum_{v \text{ vecino de } u} Times(agregoOactualizo))$$

=

$$\sum_{u \in V} Times(sacarmin) + \sum_{u \in V} \sum_{v \text{ vecino de } u} Times(agregoOactualizo)$$

$$= |V| * Times(sacarMin) + |E| * Times(agregoOActualizo)$$

Vemos que Times =  $|V| * Times(sacarMin) + |E| * Times(agregoOActualizo)$ . Si como estructura elegimos un AVL o Red Black Tree, ¿cuánto es O?

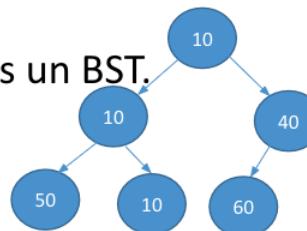
Rta  $O(|V| * \log_2 |V| + |E| * c * \log_2 |V|)$   
o sea  $O((|V| + |E|) * \log_2 |V|)$

Hay otra estructura que se usa mucho y java tiene implementada: PriorityQueue que implementa un Binary Heap. El código que nosotros vamos a usar está basado en esta estructura.

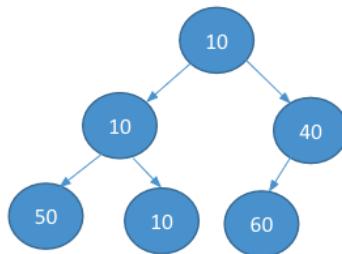
Un **Binary Heap** es un BT completo tal que cada nodo es menor o igual que todos los elementos de sus subárboles.

Claramente no es un BST.

Ej:



Se pueden representar muy eficientemente con arreglos (si me pongo a cubierto para no quedarme corto con el espacio). El arreglo surge del recorrido “por niveles” del Binary Heap.



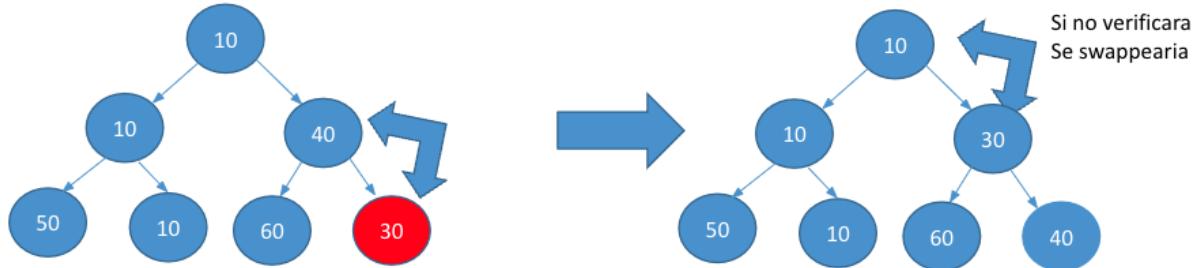
10	10	40	50	10	60		
0	1	2	3	4	5	6	7

Vemos que los nodos están en el arreglo por nivel. Esto nos ayuda a qué, si tomamos un nodo, podemos saber mediante una fórmula quién lo apunta y quienes son sus dos hijos. Cada elemento en el arreglo tiene la siguiente característica de indización: para pos=i su antecesor está en pos $\lfloor (i-1) / 2 \rfloor$ , su hijo izq está en pos  $2*i+1$  y su hijo derecho está en pos  $2*i+1+1$ .

¿Qué hay de las operaciones que precisamos?

- Consultar Min => O(1)
- Add => lo insertamos al final (trivial) pero tenemos que sólo swappear para garantizar la propiedad.
- SacarMin => lo reemplazamos por el último y luego swappeamos para garantizar la propiedad de completo.

Por ejemplo, insertamos el 30. Como viola, empezamos a swappear (en el arreglo) con el antecesor (que se donde esta...) hasta que se verifique la propiedad.



A lo sumo, ¿cuántos swappeos hacemos? La altura del árbol, o sea  $O(\log n)$ . El borrado del min es análogo: se reemplaza la raíz por la última hoja y se swappea hasta garantizar propiedad. Es  $O(\log n)$ .

Como  $\text{Times} = |V| * \text{Times}(\text{sacarMin}) + |E| * \text{Times}(\text{agregoOActualizo})$ , la complejidad para PriorityQueue es:  $O(|V| * \log_2 |V| + |E| * c * \log_2 |V|)$

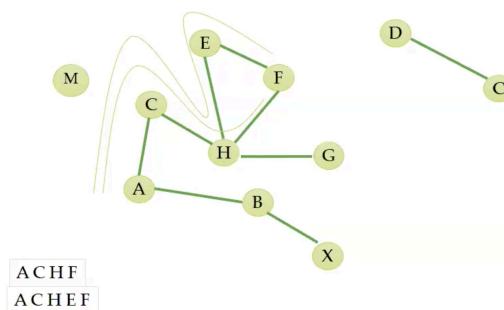
Por lo tanto, tenemos que:

- Si como estructura elegimos un AVL o RedBlackTree,  $O(|V| + |E|) * \log_2 |V|$ )
- Para PriorityQueue la complejidad es:  $O(|V| + |E|) * \log_2 |V|$ )

### Todos los Caminos y Bipartitos

Tomemos ahora el caso en que queremos poder conocer todos los caminos posibles entre dos nodos.

**g.printAllPaths('A', 'F');**

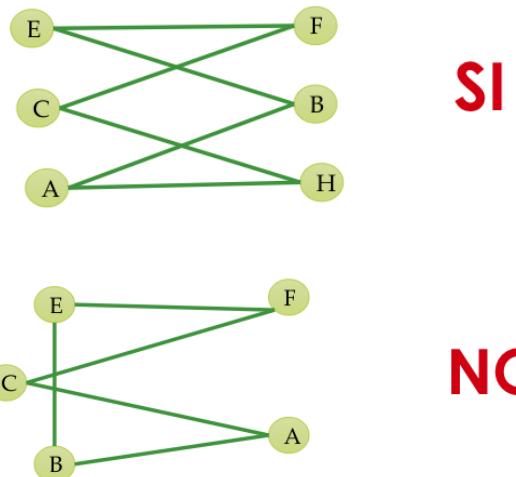


Vamos a utilizar una técnica recursiva donde vamos a ir recorriendo todos los adyacentes al nodo actual (mediante ciclos for) hasta encontrar (o no) el nodo deseado.

En la figura anterior, vemos que una opción para el camino entre A y F es ACHF. Una vez que llegamos a F, debemos volver atrás en la recursión para seguir buscando otros caminos. Para poder volver atrás, tenemos que deshacer lo que hicimos previamente (por ejemplo los visitados). Así, logramos “volver” al nodo H para ir a E y encontrar el segundo camino ACHEF.

Por lo tanto, vamos a tener una invocación recursiva dentro de un ciclo for.

Recordemos que un **grafo bipartito** es un grafo cuyos vértices se pueden separar en dos conjuntos disjuntos, de manera que las aristas no pueden relacionar vértices de un mismo conjunto.

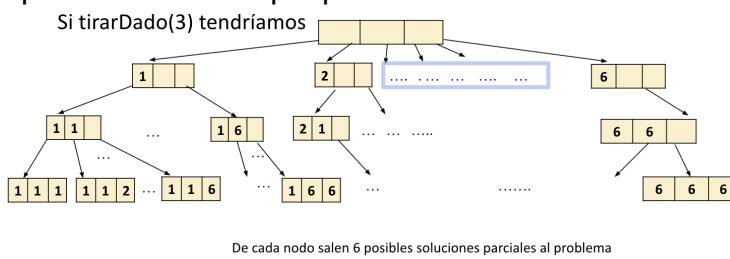


Se debe cumplir que todo nodo A tenga solo B como adyacentes y que todo nodo B tenga solo nodos A como adyacentes

## Unidad 7 - Heurísticas

Vamos a ver distintos tipos de problemas y sus soluciones. Muchos juegos se pueden resolver explorando un grafo implícito. Es decir se explora un espacio de soluciones generadas implícitamente por un grafo. Ese “grafo” representa en sus nodos una solución parcial del problema. El nodo raíz representa la configuración inicial del problema. Hay un eje desde un nodo  $n_1$  a  $n_2$  ( $n_1 \rightarrow n_2$ ) si desde  $n_1$  se puede ir a  $n_2$  aplicando una regla de juego (que lleve a un estado válido).

Ejemplo 1: Dado un número que indica la cantidad de veces que voy a tirar un dado, indicar todas las posibles valores que pueden obtenerse. Si tirarDado(3) tendríamos



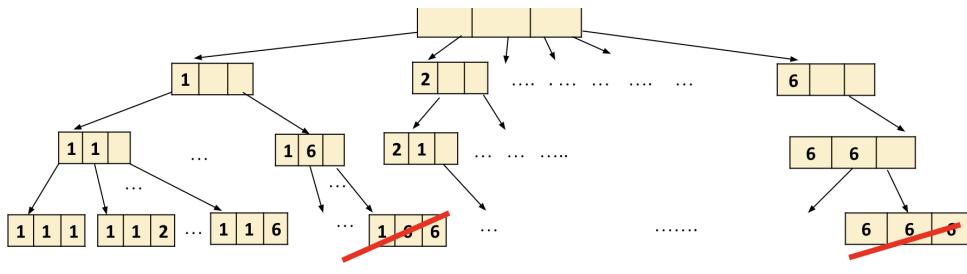
### Técnica de Búsqueda Exhaustiva

Es una técnica para buscar todas las posibles soluciones explorando el espacio de soluciones en forma implícita. ¿Para qué sirve? Un problema tiene muchas soluciones y las quiero todas.

Consideraciones: el grafo es implícito (no lo genero). Típicamente se implementa con recursión (o sea Stack) aunque puede hacerse con una Queue. Ej: DFS y BFS.  
Idea para esta técnica (típicamente recursiva):

1. Si el nodo no puede expandirse más (no hay más opciones a partir de él) => retornar/imprimir el resultado.
2. Sino, por cada posibilidad para ese nodo de expandir un pxmo nivel (ciclo for):
  - a. El nodo puede resolver un caso pendiente
  - b. Explorar nuevos pendientes (soluciones quizás parciales)
  - c. El nodo puede deshacer/quitar el caso pendiente generado

¿Qué pasa si hay restricciones? Tomemos el caso que dado un número que indica la cantidad de veces que voy a tirar un dado, se debe indicar todas las posibles valores que pueden obtenerse, siempre que la suma de los valores no supere un umbral . Si tirarDado(3, 10) tendríamos:



Si lo resuelvo con Búsqueda Exhaustiva el chequeo de las restricciones lo hago al final de la exploración de todas las posibles soluciones. En el ejemplo, analizo las 216 posibles soluciones y me quedo con aquellas que no supera el umbral, pero ya exploré TODO el espacio de posibles soluciones, lo que es poco eficiente.

### Técnicas de Backtracking

Ante la presencia de restricciones, pueden aprovecharlas para no explorar todo el grafo de soluciones y PODAR aquellos nodos que no conducen a la solución. Ahí en donde las técnicas de Backtracking entran en juego: no expande innecesariamente nodos que ya se saben (gracias a la restricción) no conducirán a la solución.

¿Cómo puedo evitar no expandir más un nodo? Un nodo intermedio ya lleva acumulado valores. En el mejor de los escenarios, completará con números bajos, es decir "1" en lo que falta. Si esa sumatoria de valores actuales junto con faltantes \*1 supera el umbral, imposible seguir.

### Backtracking + Programación Dinámica

Y si no recalculamos la suma del nodo cada vez? Por qué calcular tantas veces la suma. Lo podemos agregar como parámetro => ahí estamos introduciendo programación dinámica. En el caso de los datos, agregamos un parámetro de suma acumulada.

Ahora, agreguemos una nueva restricción. Supongamos que queremos que no se repitan, es decir no diferenciar si el valor sale en la primera, segunda o n-ésima posición. Es decir, las siguientes soluciones representan lo mismo:

### Soluciones que pueden ser consideradas equivalentes

```
[1, 1, 2] [1, 2, 1] [2, 1, 1]
[1, 1, 3] [1, 3, 1] [3, 1, 1]
[1, 1, 4] [1, 4, 1] [4, 1, 1]
[1, 2, 2] [2, 1, 2] [2, 2, 1]
[1, 2, 3] [1, 3, 2] [2, 1, 3] [2, 3, 1] [3, 1, 2] [3, 2, 1]
[1, 1, 1]
[2, 2, 2]
```

¿Cuál podría ser la elegida como representativa de cada conjunto? ¿Cuál es fácil de generar? ¿Cuál es la primera que se genera en la exploración?

### Problema de las 8 Reinas

Objetivo: En un tablero de ajedrez de  $N \times N$ , colocar  $N$  reinas de forma tal que no se jaqueen. Las reinas se jaquean: en fila, en columna o en diagonal y contradiagonal. Se quieren todas las soluciones posibles. Para este caso, vamos a ir completando el tablero poniendo una reina por columna y haciendo backtracking. Si no podemos colocar las  $N$ , volvemos a la columna anterior y buscamos otra posición para la reina. Es decir, intentamos poner una reina (hasta completar) de izquierda a derecha, en diferentes columnas. Cada vez que ponemos una nueva reina, si no satisface las restricciones deshacemos lo hecho e intentamos otra opción.

Podemos mejorar los chequeos, dado que no hace falta tener un tablero de  $N \times N$ . Si solo almacenamos un vector de  $N$  posiciones y guardamos en cada posición la fila usada nos ahorraremos de chequear.

```
// row is empty hasta col?
for(int c= 0; c< colCandidate; c++)
    if (tablero[rowCandidate][c])
        return false;
```

```
// row is empty
if (tablero[rowCandidate] != null)
    return false;
```

No hace falta chequear la diagonal. Los valores que se pueden generar para la diagonal son FIJOS. Ej: Para  $4 \times 4$  tengo 7 valores ( $2 * N - 1$ ). Los valores son iguales para toda la diagonal.

```
// diagonal empty hasta col?
for(int c=colCandidate-1, r= rowCandidate+1; c>=0 && r<tablero.length ;
    c--, r++)
    if (tablero[r][c])
        return false;
```

```
// diagonal empty
if (diag[rowCandidate + colCandidate])
    return false;
```

No hace falta chequear la contradiagonal. Los valores que se pueden generar para la contradiagonal son FIJOS. Ej: Para  $4 \times 4$  tengo 7 valores ( $2 * N - 1$ ). Los valores son iguales para toda la contradiagonal.

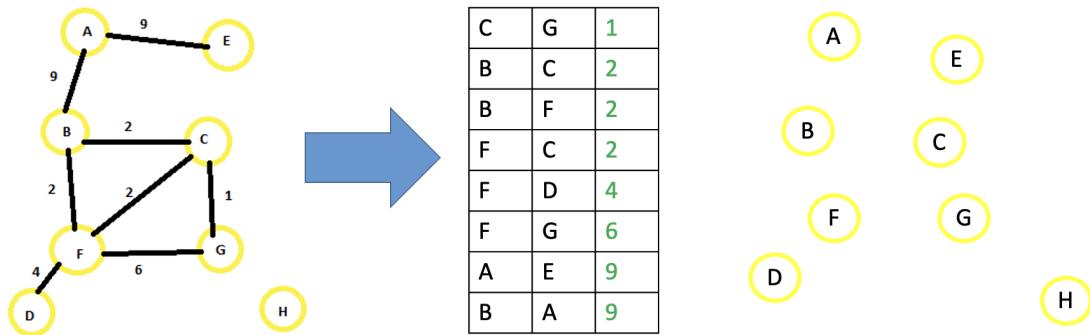
```
// contradiagonal empty hasta col?
for(int c=colCandidate-1, r= rowCandidate-1; c>=0 && r>=0 ;
    c--, r--)
    if (tablero[r][c])
        return false;
```

```
// contradiagonal empty
if (contradiag[rowCandidate - colCandidate + tablero.length - 1])
    return false;
```

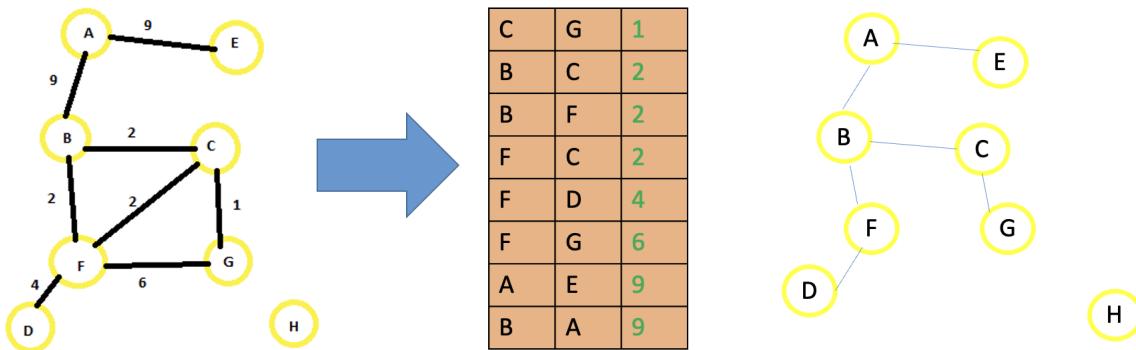
## Técnicas Algorítmicas Estudiadas

**Divide & Conquer =>** técnica que descompone un problema de tamaño N en problemas más pequeños que tengan solución. Finalmente, se debe proponer cómo construir la solución final a partir de las soluciones de los problemas menores. Ej: Mergesort, Quicksort, Búsqueda en un BST, Búsqueda en un arreglo ordenado.

**Algoritmos Avidos (Greedy) =>** técnica que busca en cada etapa un óptimo local con el objetivo de llegar al óptimo global (aunque de esta forma no siempre se consigue el óptimo global). Ejemplo: algoritmo de Kruskal para encontrar el mínimo árbol generador de un grafo.



Finalmente se obtiene:



Problema del ATM: tengo que entregar la menor cantidad de monedas que coincidan con el monto pedido. ¿Es greedy? Ej: tengo monedas de 25, 10, 1. Me piden \$27, ¿qué debo entregar? Se toma la moneda de mayor denominación, se ve cuántas se pueden entregar y luego se pasa a la siguiente de mayor denominación. Ahora me piden \$30, ¿qué debo entregar? Si usamos el algoritmo de antes, vamos a devolver una de 25 y cinco de 5, que no es lo mejor sino dar tres de 10. Por lo tanto **NO** es greedy.

**Fuerza Bruta/Búsqueda Exhaustiva (con Stack o Queue):** calcula y enumera todas las posibles soluciones. Si se agregan restricciones, ej: se pide “la mejor”, recién en el final evalúa cual es la mejor. Es decir, en las hojas se evalúa si se

satisface la restricción pedida. Pero si hubiera una restricción que me permite “podar” el árbol en ramas que son innecesarias (las que no conducen a la solución) => mucho más eficiente! **Backtracking es una opción**. Ante la presencia de restricciones, no exploran todas las posibles soluciones, sino solo las prometedoras. Ej: N-Queens

Y, ¿si no recalcuro innecesariamente cosas que calculé previamente? Entonces uso **Programación Dinámica**. Técnica que permite almacenar valores que se calcularon previamente en soluciones anteriores para reusarlos en vez de recalcularlos reiteradamente. Ej: Levenshtein, Dijkstra, Ackerman, Fibonacci.