

Estructura de Datos y Algoritmos

Resumen hecho por Magdalena Cullen

Algoritmos y Complejidad.....	2
Tiempo de ejecución.....	2
Espacio en RAM.....	3
Complejidad.....	3
Time Complexity of Sorting Algorithms.....	4
Teorema maestro.....	4
Algoritmos para búsqueda en textos.....	4
Data Quality - Matching.....	4
String Matching.....	5
SOUNDEX:.....	5
METAPHONE:.....	5
Q-GRAMS:.....	6
Exact Matching.....	7
EXACT SEARCH:.....	7
KMP (Algoritmo Knuth-Morris-Pratt):.....	7
Lucene.....	8
Programación dinámica.....	9
LEVENSHTEIN DISTANCE:.....	9
Estructuras lineales.....	10
Array (indices).....	10
Array Ordenados.....	11
Stack.....	11
Evaluador de expresiones.....	12
Listas.....	12
Lista lineal simplemente encadenada.....	12
Lista lineal doblemente encadenada.....	13
Listas circulares.....	14
Hashing.....	15
Hashing (hash table).....	15
Factor de carga(current load factor).....	16
Funciones Hash (nbr keys).....	16
Open Addressing or Closed Hashing.....	17
Borrado.....	18
Búsqueda.....	18
Inserción.....	18
Chaining or Open Hashing.....	19
★ Borrado.....	19
★ Búsqueda.....	19
★ Inserción.....	19
Bag.....	20

Hashing de JAVA.....	20
Arboles Binarios.....	20
Árboles Binarios de Expresión.....	21
Arboles Binarios en General.....	21
Arboles Binarios de Busqueda (BST).....	22
Arboles Binarios AVL.....	24
Arboles Binarios Red-Black tree.....	28
Arboles Binarios k-ario.....	29
Arboles Binarios B-tree.....	29
Grafos.....	32
Características (repaso mate discreta).....	32
Formas de recorrer un grafo.....	33
BFS(Breadth-first Search).....	33
DFS(Depth-first Search).....	34
Dijkstra.....	34
Tipos de algoritmos-heuristicas.....	35
Greedy.....	35
Backtracking.....	35
Stack o Queue.....	35
Divide and conquer.....	35
Exhaustiva.....	35

Algoritmos y Complejidad

Tiempo de ejecución

1. **empírico**: la idea de usar la métrica “tiempo de ejecución calculada empíricamente” para rankear algoritmos

dificultades:

- tardan diferente dependiendo de los datos con los que operan
- tiempo también puede cambiar dependiendo de tu computadora

2. **teórico**: consiste en usar una descripción de alto nivel del algoritmo para evaluar su eficiencia independientemente del hardware y software donde ejecuta → fórmula matemática

“contar la cantidad de operaciones primitivas”

- comparaciones
- operaciones aritméticas
- transferencia de control desde una función a otra

OBS: Tamaño del input afecta la performance del algoritmo, entonces la “fórmula” depende de la cantidad de operaciones primitivas y tamaño de dicha entrada

Comportamiento Asintótico, Cota Superior u O Grande

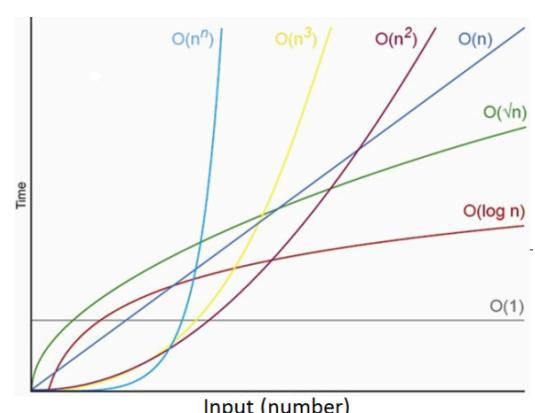
La descripción que buscamos para comparar algoritmos es una “asintota” (cota) expresada en términos de n que nos permita caracterizar la tasa de crecimiento u orden de crecimiento de la fórmula.

Comportamiento Asintótico Superior - O grande (Asymptotic upper bound running time - O-notation):

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 depende de n) y $\exists n_0 > 0$ tal que $\forall n \geq n_0$ se cumple que

$$0 \leq T(n) \leq c * g(n)$$



$n \rightarrow \infty$ tenemos que:

$$\dots < O(\log_2(n)) < O(\sqrt{n}) < O(n) < O(n \log_2(n)) < O(n^2) < O(n^3) < \dots$$

la mejor complejidad es $O(1) \rightarrow$ constante

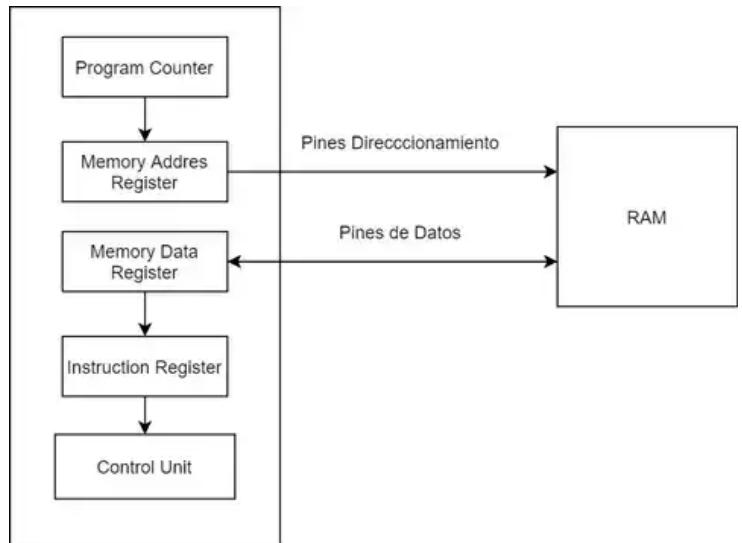
Espacio en RAM

Para que un algoritmo ejecute algo en el procesador, los datos deben estar en RAM. Osea, puede ser que los datos se almacenan en el disco, pero el procesador no va directo al disco.

va a disco → lo carga en RAM → lo ejecuta

Consiste en usar una descripción de alto nivel del algoritmo para evaluar cuánto espacio extra precisa para sus variables (parámetros formales, invocaciones a otra funciones, variables locales). Se lo describe con una fórmula en términos del tamaño de entrada del problema.

No siempre un algoritmo es mejor que otro tanto en la parte temporal como en la espacial. Muchas veces sucede que es mejor en lo temporal y peor en lo espacial o viceversa. Es decir, evaluar si un algoritmo es mejor que otro puede ser un tradeoff entre espacio vs tiempo.

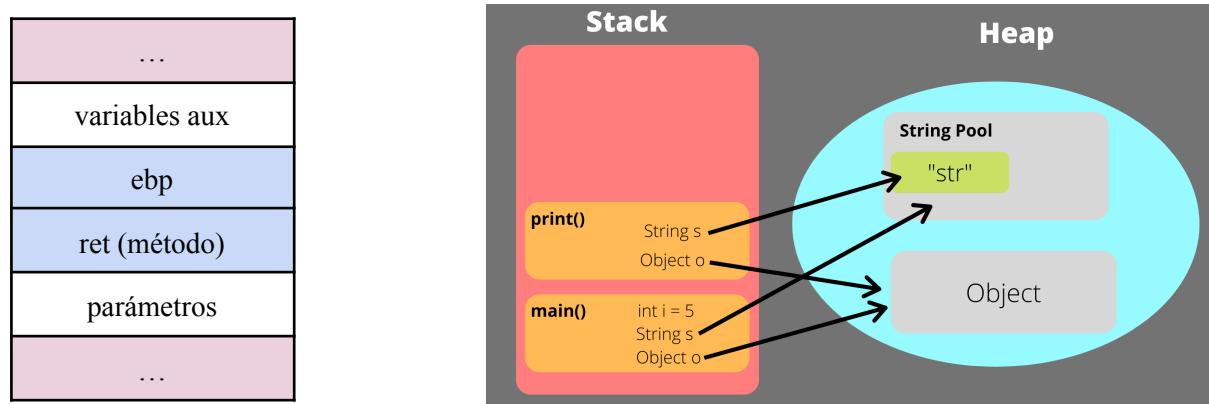


HEAP:

En java cada vez que llamamos a “new” reservamos lugar en esta zona. El **GARBAGE COLLECTOR** 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:



Complejidad

temporal	espacial
se refiere al tiempo que tarda un algoritmo en completar su ejecución en función del tamaño de la entrada	se refiere al espacio en memoria que utiliza el algoritmo en función del tamaño de la entrada

Time Complexity of Sorting Algorithms

Name of Sorting Algorithm	Best Time Complexity	Worst Time Complexity
Bubble Sort	$O(n)$	$O(n^2)$
Merge Sort	$O(n \log(n))$	$O(n \log(n))$
Quick Sort	$O(n \log(n))$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$

Teorema maestro

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

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

Algoritmos para búsqueda en textos

Data Quality - Matching

La coincidencia de datos es un proceso crítico para garantizar la calidad y la integridad de los datos en una organización.

Ayuda a identificar relaciones entre datos dispersos y garantiza que los datos se utilicen de manera precisa y eficaz en análisis y toma de decisiones.

Las mínimas reglas que deberían aplicarse son:

- Sacar blancos del comienzo y final (trim). Pero no es suficiente. Si la palabra es compuesta habría que sacar blancos internos.
 - Ej: ‘ yogurt bebble ’
- 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: John Peter Doe = John P. Doe = J. D. Doe = Doe, John Peter = Doe, John P. = Doe, J. P.

String Matching

SOUNDEX:

Es una técnica fonética utilizada en la indexación y búsqueda de texto para encontrar palabras o nombres que suenan de manera similar.

Soundex Code	Letters
1	B, F, P, V
2	C, G, J, K, Q, S, X, Z
3	D, T
4	L
5	M,N
6	R
No Code	A, E, I, O, U, H, W, Y

- **objetivo:** es encontrar palabras o nombres que tengan una pronunciación similar, incluso si se escriben de manera diferente.
- **Reducción a Código:** Soundex reduce cada palabra o nombre a un código alfanumérico corto basado en su pronunciación.
 - ejemplo: Soundex("hold") = H430

palabra	tabla	soundex
h	H	H430
o	-	
l	4	
d	3	

METAPHONE:

Es un algoritmo de indexación fonética más avanzado que Soundex, diseñado para encontrar palabras o nombres que suenan de manera similar. Ofrece una mayor capacidad de manejo de variaciones de pronunciación y es utilizado en una variedad de aplicaciones relacionadas con el procesamiento de texto y la búsqueda de información.

reglas:

1. Drop duplicate adjacent letters, except for C.
2. If the word begins with 'KN', 'GN', 'PN', 'AE', 'WR', drop the first letter.
3. Drop 'B' if after 'M' at the end of the word.
4. 'C' transforms to 'X' if followed by 'IA' or 'H' (unless in latter case, it is part of '-SCH-', in which case it transforms to 'K'). 'C' transforms to 'S' if followed by 'T', 'E', or 'Y'. Otherwise, 'C' transforms to 'K'.
5. 'D' transforms to 'J' if followed by 'GE', 'GY', or 'GI'. Otherwise, 'D' transforms to 'T'.
6. Drop 'G' if followed by 'H' and 'H' is not at the end or before a vowel. Drop 'G' if followed by 'N' or 'NED' and is at the end.
7. 'G' transforms to 'J' if before 'T', 'E', or 'Y', and it is not in 'GG'. Otherwise, 'G' transforms to 'K'.
8. Drop 'H' if after a vowel and not before a vowel.
9. 'CK' transforms to 'K'.
10. 'PH' transforms to 'F'.
11. 'Q' transforms to 'K'.
12. 'S' transforms to 'X' if followed by 'H', 'IO', or 'IA'.
13. 'T' transforms to 'X' if followed by 'IA' or 'IO'. 'TH' transforms to 'O'. Drop 'T' if followed by 'CH'.
14. 'V' transforms to 'F'.
15. 'WH' transforms to 'W' at the beginning. Drop 'W' if not followed by a vowel.
16. 'X' transforms to 'S' at the beginning. Otherwise, 'X' transforms to 'KS'.
17. Drop 'Y' if not followed by a vowel.
18. 'Z' transforms to 'S'.
19. Drop all vowels unless it is the beginning.

ejemplo:

palabra	regla		Metaphone
h	H	Drop 'H' if after a vowel and not before a vowel.	HLT
o	-		
l	L		
d	T		

palabra	Regla		Metaphone
PH	F	'PH' transforms to 'F'.	FN
o	-		
n	n		
e	-		

Q-GRAMS:

Consiste en generar los pedazos que componen una palabra.

División en q-gramas: El primer paso del algoritmo q-gram consiste en dividir una cadena de texto en fragmentos de longitud fija llamados "q-gramas". El valor de "q" es un parámetro que se elige previamente y determina la longitud de los fragmentos.

La distancia entre 2 strings estará dada por la cantidad componentes que tengan en común:

$$\text{similitud}(s1, s2) = \frac{s1.length + s2.length - \text{notAMatch}}{s1.length + s2.length}$$

ejemplo:

Q-gram(John)= {‘##J’, ‘#JO’, ‘JOE’, ‘OE#’, ‘E##’}

0 1 2 3 4

Q-gram(Joe)= {‘##J’, ‘#JO’, ‘JOH’, ‘OHN’, ‘HN#’, ‘N##’}

0 1 2 3 4 5

comparten = { ‘##J’, ‘#JO’ }

no comparten = { {‘JOE’, ‘OE#’, ‘E##’} , {‘JOH’, ‘OHN’, ‘HN#’, ‘N##’} } → 3+4

$$\text{similitud}(\text{John}, \text{Joe}) = \frac{6+5-(3+4)}{6+5} = 0.3636$$

Exact Matching

EXACT SEARCH:

Un enfoque básico y sencillo para resolver un problema sin utilizar técnicas más avanzadas u optimizadas.

ejemplos:

- naive string matching
- bubble sort
- linear search

n = target	Complejidad temporal: O(n m)
m = source	Complejidad espacial: O(1)

KMP (Algoritmo Knuth-Morris-Pratt):

Es un algoritmo de búsqueda de subcadenas simple. Por lo tanto su objetivo es buscar la existencia de una subcadena dentro de otra cadena. La búsqueda se lleva a cabo utilizando información basada en los fallos previos

como crear el Next:

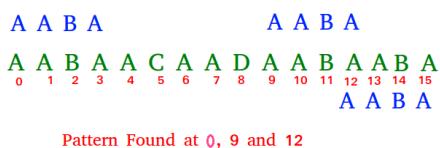
query	B	C	B	C	B	R
Next	0	0	1	2	3	0

La tabla de Next se construye de manera que para **cada posición en la subcadena**, se determine **cuántos caracteres se pueden saltar hacia atrás en caso de que haya una discrepancia**

cómo funciona la búsqueda:

Text : A A B A A C A A D A A B A A B A

Pattern : A A B A

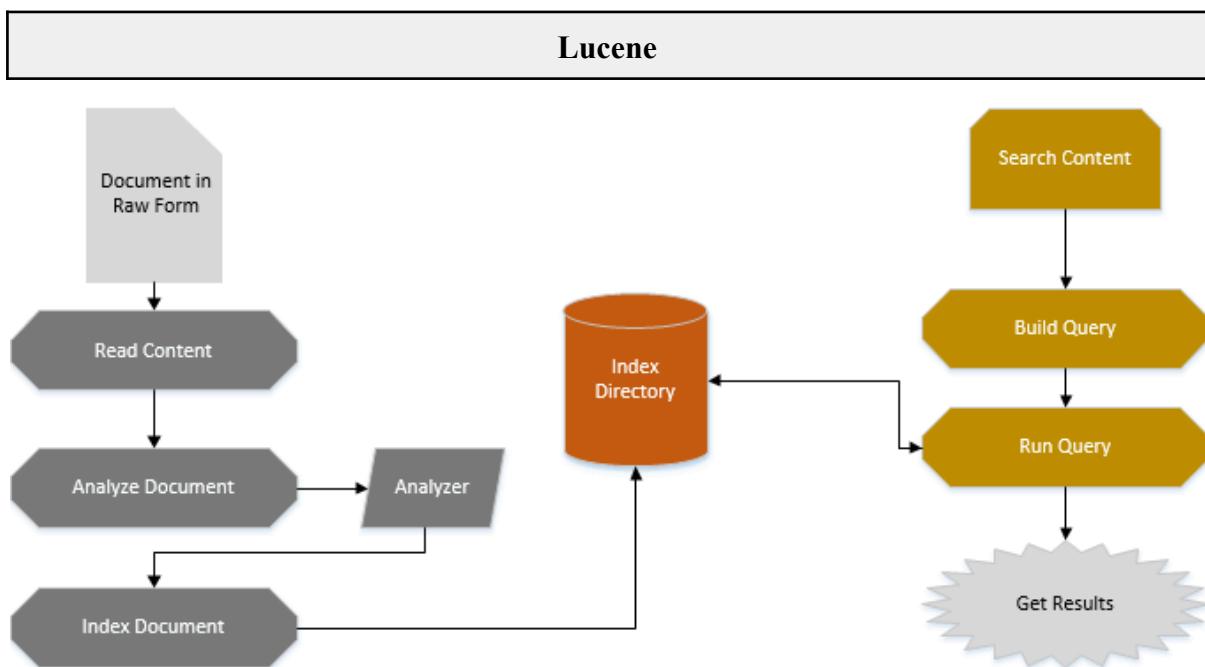


Una vez que se ha construido la tabla de Next, el algoritmo realiza una búsqueda en la cadena principal.

Supongamos que rec apunta al carácter en target y que pquery que apunta a un carácter en query.
Mientras haya coincidencia, avanzó en ambos.

Cuando no la haya, se “shiftea” query a next[pquery-1], salvo que pquery sea cero, en cuyo caso hay que avanzar rec en target.

$m = \text{longitude de query}$ $n = \text{longitud de target}$	Complejidad especial: $O(m)$ Complejidad temporal: $O(m)$
--	--



Lucene funciona de la siguiente manera:

1. **Configuración:**
 - Descarga e incorpora las bibliotecas de Lucene en tu proyecto.
2. **Indexación:**
 - Define la estructura de los documentos y crea un índice utilizando un analizador.
3. **Análisis de Consultas:**
 - Convierte las consultas del usuario en términos individuales mediante el mismo analizador.
4. **Realización de Consultas:**
 - Utiliza el IndexSearcher para buscar en el índice utilizando la consulta.

- **Query:**
 - i. **Term Query:**
 - Busca documentos que contengan un término específico.
 - ii. **Range Query:**
 - Encuentra documentos con valores dentro de un rango especificado (por ejemplo, fechas o números).
 - iii. **Wildcard Query:**
 - Busca documentos que contienen términos que coinciden con un patrón que incluye comodines.
 - iv. **Fuzzy Query:**
 - Encuentra documentos que contienen términos similares a un término dado, permitiendo cierta flexibilidad en la coincidencia.
 - v. **Phrase Query:**
 - Busca documentos que contengan una secuencia específica de términos en el orden dado.
 - vi. **Boolean Query:**
 - Combina consultas utilizando operadores lógicos como AND, OR y NOT para refinar los resultados.
 - vii. **Proximity Query:**
 - Encuentra documentos donde los términos están dentro de una distancia específica entre sí.
 - viii. **Boosting Query:**
 - Asigna un peso a ciertos términos o consultas para resaltar su importancia en los resultados

5. Recuperación de Resultados:

- Procesa y recupera la información relevante de los resultados de la búsqueda.

6. Manejo de Cierre:

- Cierra adecuadamente los recursos (**IndexReaders**, **IndexWriters**) después de su uso.

Programación dinámica

LEVENSHTEIN DISTANCE:

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

$$lev_{s_1, s_2}(i, j) = \begin{cases} max(i, j) \\ min \begin{cases} lev_{s_1, s_2}(i - 1, j) + 1 & (\text{insert}) \\ lev_{s_1, s_2}(i, j - 1) + 1 & (\text{delete}) \\ lev_{s_1, s_2}(i - 1, j - 1) + 1 & (\text{if } s_1 \neq s_2) \text{ (replace)} \end{cases} \end{cases}$$

A diferencia de Soundex, que se adoptó para proponer una medida de similitud, a partir de su cálculo, Levenshtein **ES UNA MÉTRICA DE DISTANCIA**.

Por lo tanto, cumple con propiedades:

1. Simetría.

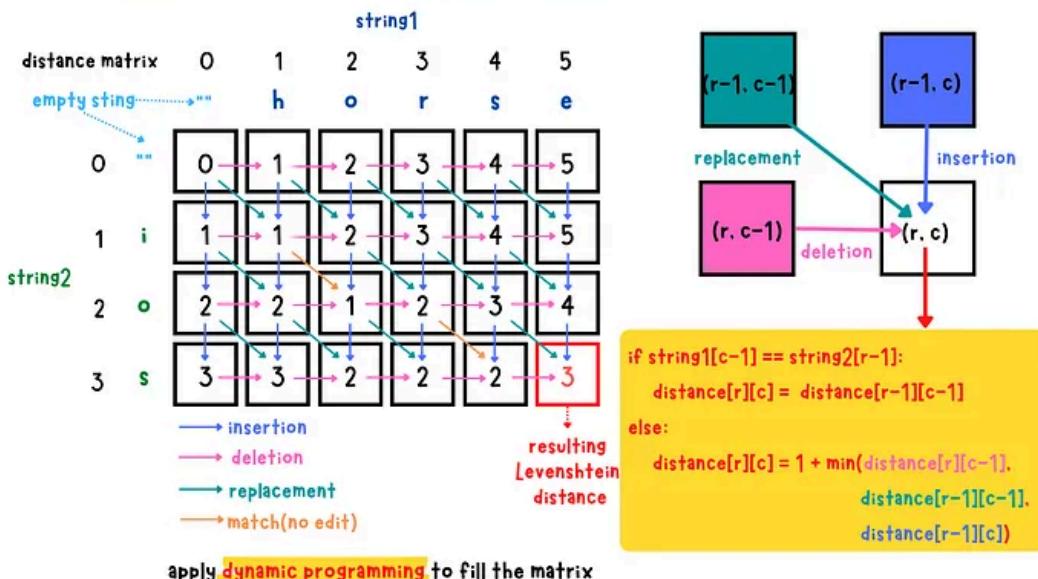
$$\text{Levenshtein}(\text{str1}, \text{str2}) = \text{Levenshtein}(\text{str2}, \text{str1}).$$

2. Desigualdad.

$$\text{Levenshtein}(\text{str1}, \text{str2}) = \text{Levenshtein}(\text{str2}, \text{str3}) \geq \text{Levenshtein}(\text{str1}, \text{str3})$$

$$\text{normalized}(s1, s2) = 1 - \frac{\text{lev}(s1, s2)}{\max(s1.length, s2.length)}$$

Levenshtein Distance: the minimum number of edits to transform a string to another.
edit operations: insertion, deletion and replacement



Estructuras lineales

Array (indices)

MUY EFICIENTE

Características de los índices:

- la clave de búsqueda puede o no, tener repetidos.
 - si es única no tiene sentido hablar de comparación
 - en caso contrario, podremos tener información adicional complementaria
- EL INDICE SOLO SE UTILIZA PARA BUSCAR
 - 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: si borro un documento en la colección, preciso que el índice lo refleje

	Búsqueda	Inserción	Borrado
Array	O(n)	O(1)	O(n)
Array ordenado	$O(\log_2(n))$	O(n)	O(n)
Hashing	??	??	??
obs: el problema de hashing es que debe resolver colisiones, si no tiene es O(1)			

Array Ordenados

Usa “búsqueda binaria” → O($\log_2(n)$)

```
public int busquedaIter(int[] arr, int x){
    int left = 0, right = arr.length -1;
    while( left <= right){
        int middle = left + (right - left)/2;
        if(arr[middle] < x){ // belongs to right side of array, then, change left value
            left = middle+1;
        }
        else if(arr[middle] > x){ // belongs to right side of array, then, change right value
            right = middle-1;
        }else{ // arr[middle] == x therefore index is middle
            return middle;
        }
    }
    return -1;
}

public int busquedaRec(int[] arr, int x, int left, int right){
    if(left <= right){
        int middle = left + (right - left)/2;
        if(arr[middle] == x){
            return middle; // recursion ends
        }

        if(arr[middle] < x){ // belongs to right side of array, then, change left value
            busquedaRec(arr, x, middle+1, right);
        }
        else { // belongs to right side of array, then, change right value
            busquedaRec(arr, x, left, middle-1);
        }
    }
    return -1;
}
```

Stack

⚠ DATOS ORDENADOS POR ORDEN DE LLEGADA

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.

OBS: Si se lo implementa con una **lista lineal simplemente encadenada**, como los **elementos sólo se acceden por el tope**, es sólo cuestión de apuntar

el tope al elemento conveniente. Es decir, **el primer elemento de la lista**.
 Así, jamás tenemos que recorrer el push/pop.
 → **Resulta más conveniente la lista lineal.**

Evaluador de expresiones

$(2 + (3 * 4)) = x$		
= + 2 * 3 4 x	2 + 3 * 4 = x	2 3 4 * + x =
prefija	infija	postfija
El operador se encuentra antes de los operandos sobre los que aplica.	El operador se encuentra entre los operadores sobre los que aplica.	El operador se encuentra detrás de los operadores sobre los que aplica.

algoritmo pasos: (**entrada postfija**)

1. si es operador
 - a. si es vacio pushear al stack
 - b. si es mayor que el operador peek, pushear al stack
 $\wedge > /, * > +, -$
 - c. si son de igual “valor”, popear el peek y luego pushear el operador siendo evaluado
 $+, - \quad *, /$
 - d. si es menor agregar al vector de salida
2. si es un numero agregar al vector de salida

Listas

Lista lineal simplemente encadenada

Es una estructura de datos compuesta por 0 o más nodos. Cada nodo (elemento) almacena 2 datos: Su información y la referencia al siguiente elemento.

ORDENADA: elementos ordenados con algún criterio de ordenación

CON HEADER:

- elemento distinguido → header (primer elemento de la lista)
- Cada nodo/elemento (común) almacena 2 datos: Su información y la referencia al siguiente elemento

Uso de Iteradores:

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

- a. **Borrar/insertar un elemento recorriendo desde el header:**
 - i. Tenemos que la búsqueda se hace en $O(n)$.
 - ii. Aunque no hay movimiento de datos porque no se precisa contigüidad, la operación es entonces $O(n)$.
- b. **Borrar/insertar un elemento sin buscarlo:**
 - i. Estamos parados en el elemento y como no hay movimiento de datos para garantizar contigüidad, entonces sería $O(1)$.
 - ii. Para estar “apuntando al elemento a borrar” vamos a colocar el delete() en el iterador.

	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 ordenadas por clave de búsqueda	O(n)	O(n)	O(1)	O(n)	O(1)

Lista lineal doblemente encadenada

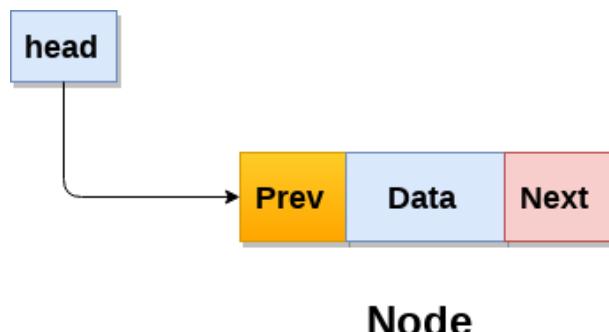
Es una estructura de datos compuesta por:

- Un elemento distinguido llamado header que tiene la referencia del primer elemento y además información global de la lista.
- Cada nodo/elemento almacena 3 datos: Su información, y la referencia a los elementos previo y siguiente.

Una variante es una Lista Lineal Doblemente Encadenada Ordenada con Header, que mantiene los elementos ordenados con algún criterio de ordenación.

```
class Node {
    int data;
    Node prev, next;

    public Node(int data) {
        this.data = data;
        this.prev = null;
        this.next = null;
    }
}
```



Insertar un Nodo:

1. Crear un nuevo nodo con la información deseada.
 - a. Este nodo contendrá la información que deseas almacenar en la lista.
2. Determinar la posición donde se insertará el nuevo nodo.
 - a. Puede ser al principio, al final o en una posición específica de la lista.
3. Actualizar los punteros del nuevo nodo y de los nodos vecinos.
 - a. Si estás insertando al principio, actualiza el puntero next del nuevo nodo para que apunte al nodo actual al principio.
 - b. Si estás insertando al final, actualiza el puntero prev del nuevo nodo para que apunte al último nodo actual.

- c. Si estás insertando en una posición específica, actualiza los punteros de los nodos vecinos para incluir el nuevo nodo.

Buscar un Nodo:

1. Comenzar desde el inicio de la lista (head).
- a. Si estás buscando por valor, compara el valor del nodo actual con el valor deseado.
2. Avanzar al siguiente nodo.
- a. Si el valor no coincide, pasa al siguiente nodo.
- b. Si estás buscando por posición, sigue avanzando el número de veces necesario.
3. Repetir el proceso hasta encontrar el nodo deseado o llegar al final de la lista.
- a. Si llegas al final de la lista sin encontrar el nodo, el nodo no está en la lista.
4. Finalizar la búsqueda.
- a. Si encuentras el nodo, puedes devolverlo o realizar la acción necesaria.

Eliminar un Nodo:

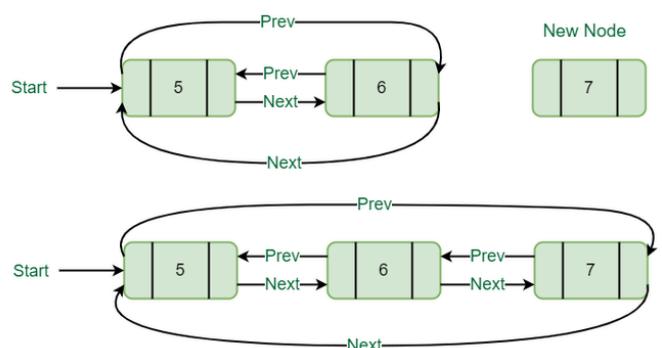
1. Identificar el nodo que se va a eliminar.
- a. Puedes buscar el nodo por su valor o posición en la lista.
2. Actualizar los punteros de los nodos vecinos.
- a. Si el nodo tiene un nodo anterior (prev), actualiza el puntero next de ese nodo para saltar el nodo que se eliminará.
- b. Si el nodo tiene un nodo siguiente (next), actualiza el puntero prev de ese nodo para saltar el nodo que se eliminará.
3. Liberar la memoria del nodo a eliminar.
- a. Esto puede hacerse automáticamente en algunos lenguajes de programación o manualmente si estás trabajando con lenguajes que requieren gestión manual de memoria.

Listas circulares

Pueden ser encadenadas simples o dobles, en las que el último elemento, posee la referencia al primer elemento de la misma

```
class Node {
    int data;
    Node next;

    public Node(int data) {
        this.data = data;
        this.next = null;
    }
}
```



Insertar un Nodo en una Lista Circular:

1. Crear un nuevo nodo con la información deseada.
 - a. Este nodo contendrá la información que deseas almacenar en la lista.
2. Determinar la posición donde se insertará el nuevo nodo.
 - a. Puede ser al principio, al final o en una posición específica de la lista.
3. Actualizar los punteros del nuevo nodo y de los nodos vecinos.

- a. Si estás insertando al principio, actualiza el puntero next del nuevo nodo para que apunte al nodo actual al principio y el puntero next del último nodo para que apunte al nuevo nodo.
- b. Si estás insertando al final, actualiza el puntero next del último nodo para que apunte al nuevo nodo y el puntero next del nuevo nodo para que apunte al nodo actual al principio.
- c. Si estás insertando en una posición específica, actualiza los punteros de los nodos vecinos para incluir el nuevo nodo.

Buscar un Nodo en una Lista Circular:

1. Comenzar desde cualquier nodo de la lista.
 - a. Puedes empezar desde cualquier nodo de la lista, ya que la lista es circular y no tiene un inicio o fin definido.
2. Comparar el valor del nodo actual con el valor deseado.
 - a. Si estás buscando por valor, compara el valor del nodo actual con el valor deseado.
3. Avanzar al siguiente nodo.
 - a. Si el valor no coincide, pasa al siguiente nodo.
4. Repetir el proceso hasta volver al nodo de inicio o encontrar el nodo deseado.
 - a. Si vuelves al nodo de inicio sin encontrar el nodo, significa que el nodo no está en la lista.

Eliminar un Nodo de una Lista Circular:

1. Identificar el nodo que se va a eliminar.
 - a. Puedes buscar el nodo por su valor o posición en la lista.
2. Actualizar los punteros de los nodos vecinos.
 - a. Si el nodo tiene un nodo anterior (prev), actualiza el puntero next de ese nodo para saltar el nodo que se eliminará.
 - b. Si el nodo tiene un nodo siguiente (next), actualiza el puntero next del nodo anterior al nodo que se eliminará para saltar el nodo que se eliminará.
3. Liberar la memoria del nodo a eliminar.
 - a. Esto puede hacerse automáticamente en algunos lenguajes de programación o manualmente si estás trabajando con lenguajes que requieren gestión manual de memoria.

Hashing

Hashing (hash table)

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

 OBS: Prioriza la búsqueda, tratando de que el algoritmo tiene complejidad cercana a $O(1)$.

Para minimizar la búsqueda de elementos, el LookUp será usado de la siguiente manera:

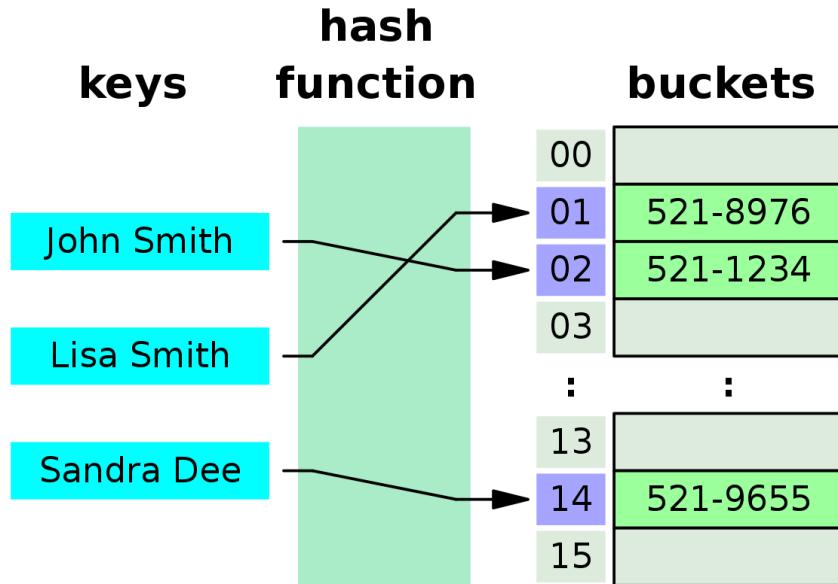
- El lugar que ocupa el elemento sea calculado por alguna fórmula:

hash: Keys → LookUp , donde LookUp[hash(key)] contiene el valor.

- Siempre debe garantizarse que el hash(key) devuelve una ranura válida. Por lo tanto, hash(key) se define como:

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

Así, permitimos que el usuario pueda proporcionar la función prehash(), sin conocer el tamaño adecuado para el LookUp.



Definiciones:

- Se dice que **dos claves** $\text{key1} \neq \text{key2}$ **colisionan** si $\text{hash}(\text{key1}) = \text{hash}(\text{key2})$, es decir, se les asigna la misma ranura.
- Se dice que una función de Hash es **perfecta** si no produce colisiones. Es decir, $\text{key1} \neq \text{key2} \Rightarrow \text{hash}(\text{key1}) \neq \text{hash}(\text{key2})$. Sería una función **inyectiva**.

Una función 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, se debe garantizar que nunca van a colisionar.

🕒 **OBJETIVO:** Una función de hash buena (aunque no perfecta) será aquella que **minimiza la cantidad de colisiones**.

Factor de carga(current load factor)

$$\frac{|\text{keys used}|}{|\text{look-up}|} = \text{current load factor}$$

cuando el factor de carga supera un umbral predefinido (Load Factor Threshold) hay que duplicar espacio y rehashear

Funciones Hash (nbr keys)

- **División ó Módulo:**
 - $\text{hash}(X) = X \bmod m$. Donde m debe ser número primo .
- **Mid-Square:**
 - Se calcula el cuadrado de un número y se toman los bits centrales como lugar donde hashear.
 - Ejemplo:
 - si $X = 14$, $X * X = 23420$ y hay que tomar los bits centrales para poder hashear.
 - Si la tabla tuviera 11 elementos usaría el número $34 \% 11$ (o $42 \% 11$).
- **Folding o Plegado:**
 - Se divide el número en zonas de la misma longitud. Se las suma y se toman los bytes necesarios.
 - Ejemplo:
 - si el numero es 20112241203123 y la tabla tiene 101 elementos, se arman grupos de a 3 dígitos,
 - se obtiene $020 + 112 + 241 + 203 + 123 = 699$ y se lo hashea a $699 \% 101$.
- **Análisis del Dígito:**
 - implica un conocimiento de **antemano**, de las características de la población.
 - Se analizan los patrones de las claves, en busca de la información de la clave que menos se repite.
 - Ejemplo:
 - Si las claves para hashear los alumnos de una facultad en cierto año fuera su DNI, no convendría elegir sus primeros dígitos porque todos los alumnos de un mismo año comienzan con las mismas cifras de DNI.

 **TIP :** a los efectos de mejorar la dispersión conviene que el tamaño de la tabla sea un número PRIMO

COLISIONES

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

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 2 estados:

1. tiene un elemento
2. no tiene un elemento (dado por baja lógica o bien por baja física).

formas de resolver esa colisión:

- lineal:
 - 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.
- cuadrático:
 - El intervalo entre ranuras a usar, si hubiera colisión, será cuadrática.
- otra combinación predeterminada:
 - siempre conviene que la última sea rehashing 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)
- lógico (se lo marca como que no está, y si más tarde hay que insertar en esa ranura se la puede aprovechar)

Búsqueda

1. Comienza buscando la clave en la ranura calculada.
2.
 - a. Si el lugar está con **baja física seguro que no está en otro lado**.
 - b. Caso contrario: si está marcado como **ocupado**
 - i. **coincide** con el valor esperado, se ha **encontrado!!!**.
 - ii. **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). Osea 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.

Inserción

- A. Si la ranura calculada está marcada como baja física, el elemento se inserta allí.
- B. Caso contrario (está ocupado y no es el elemento a insertar, o bien está marcado como baja lógica)
 1. 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.

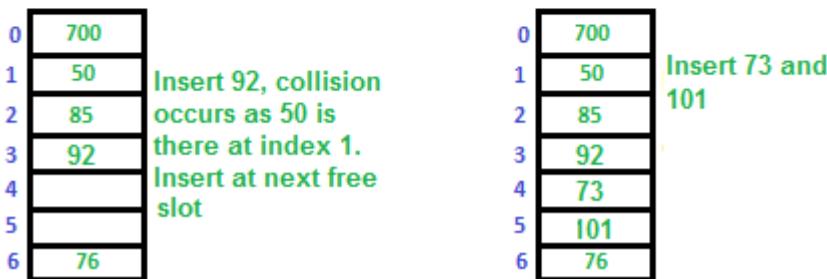
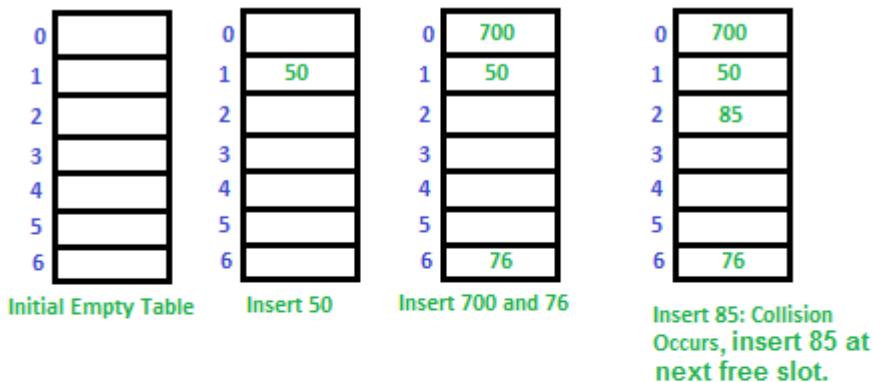
2. Una vez que se encuentra la primera baja física se lo puede insertar allí o en alguna de las bajas lógicas halladas en ese trayecto.

👉 **TIP:** Para no rebotar de más en las colisiones, lo mejor en el algoritmo de inserción es no insertar en el primer espacio libre que se encuentra, sino en la primera baja lógica que se encontró en el camino hasta descubrir que se podía insertar (se halló baja física).

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

Si hay lugar lo encuentra seguro.

La desventaja se presenta cuando el factor de carga es alto ! Buscar qué es lo que se llama “Primary Clustering”



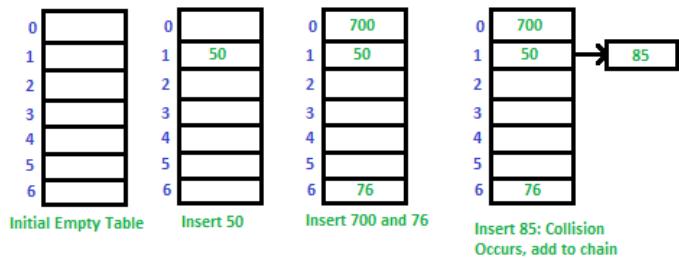
Chaining or Open Hashing

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. Si la lista tiene una única componente, entonces no hubo colisión aún en esa ranura.

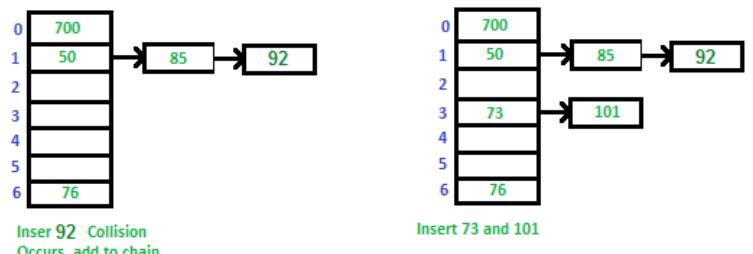
★ Borrado

- Si la ranura está en **null**, el elemento **no existía**.
- Si hay zona de **overflow** se lo **elimina** de la misma.
- Si la zona de overflow queda **vacía**, la ranura **vuelve a valer null**.



★ Búsqueda

- Si la ranura está en **null**, **no está**.
- Si hay zona de **overflow**, se lo navega allí para ver si se **encuentra**.



★ Inserción

- Si se le asigna una ranura vacía, se habilita la zona de overflow.

- Si ya había zona de overflow se lo intenta insertar allí.

⚠ IMPORTANTE EN UN HASHING NO HAY ORDEN DE LOS ELEMENTOS

Bag

Un Bag es una colección que **permite** elementos **repetidos** en ella (no los ignora ni los rechaza). Está diseñada para devolver **rápidamente** cuántas **ocurrencias** tiene cierto elemento en la misma.

Los elementos de un Bag no son key/values sino elementos.

SIEMPRE QUE USEMOS UN HASHING E INVENTEMOS TIPOS PROPIOS:

- redefinir hashCode()
- redefinir equals()

Hashing de JAVA

- a) Arreglo de ranuras con zona de overflow

```
transient Node<K,V>[] table;
```

- b) 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;
```

- c) Si se acaba el espacio

```
++modCount;
if (++size > threshold)
    resize();
```

- d) si NO hay mucha ocupación en una ranura, entonces usa una lista

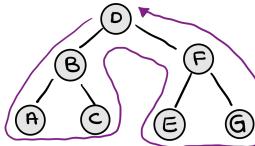
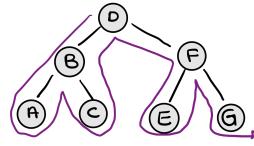
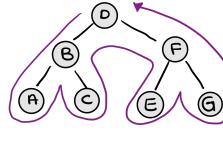
i) PERO si hay demasiado elementos allí, los transforma en un red-black tree

		Closed hashing	Chaining
Complejidad temporal	search	O(n) (linear or quadratic)	O(n) (linear or quadratic)
	insert	O(n) (linear or quadratic)	O(n) (linear or quadratic)
	delete	O(n) (linear or quadratic)	O(n) (linear or quadratic)

Arboles Binarios

Es una 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.

Pre-order: root → left → right	in-order: Left → root → right	Post-order: left → right → root
 D B A C F E G PRE-ORDEN 1º vez que pasa nodo	 A [B] C [D] E [F] G IN-ORDEN izquierda → raíz → derecha	 A C B E G F D POST-ORDEN ultima vez que se pasa por el nodo

Árboles Binarios de Expresión

Expresiones: los compiladores interpretan expresiones formadas por constantes, variables y sub-expresiones.

Árboles binarios de expresión: Los nodos internos representan operadores binarios y unarios. Las hojas representan los operandos, es decir, constantes y variables.

 **TIP :** Para el input vamos a pedir que finalice en '\n'. Los **espacios** serán los **separadores** de tokens.

```
Scanner lineScanner = new Scanner(line).useDelimiter("\\s+");  
  
while (lineScanner.hasNext()) {  
    String token = lineScanner.next();  
    ...  
}  
...  
lineScanner.close();
```

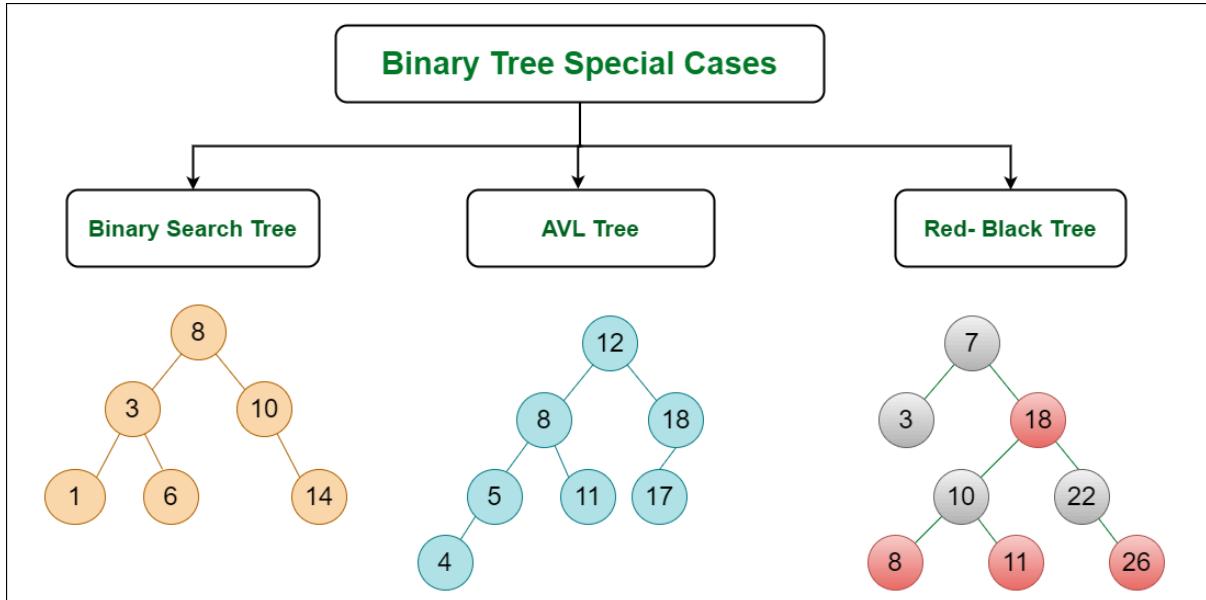
Esto se hace para separar los elementos del árbol:

1. se crea un Scanner para parsear "line", un variable que se pasa por la función, osea, mi expresión a analizar.
2. como los elementos se separan con espacios, se usa el espacio como delimiter (como strtok de PI)
3. luego se cicla por los elementos como una lista
4. una vez que se termina de usar el scanner hay cerrar para buena práctica

Arboles Binarios en General

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.

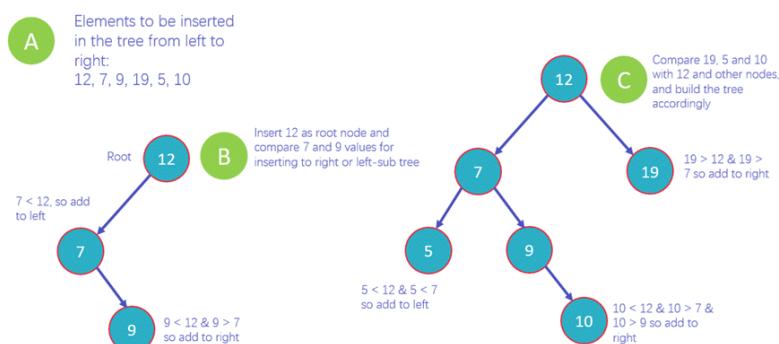
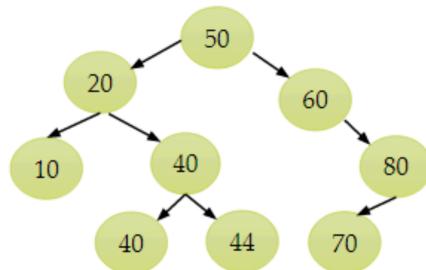


Arboles Binarios de Búsqueda (BST)

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.

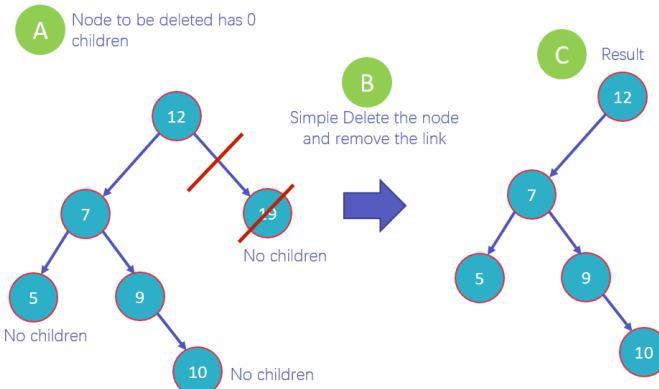
operaciones:

- **Insertar:** un BST crece desde las hojas siguiendo :
 - todos los datos de su **subárbol izquierdo son menores o iguales** que su dato
 - todos los datos de su **subárbol derecho son mayores** que su dato.

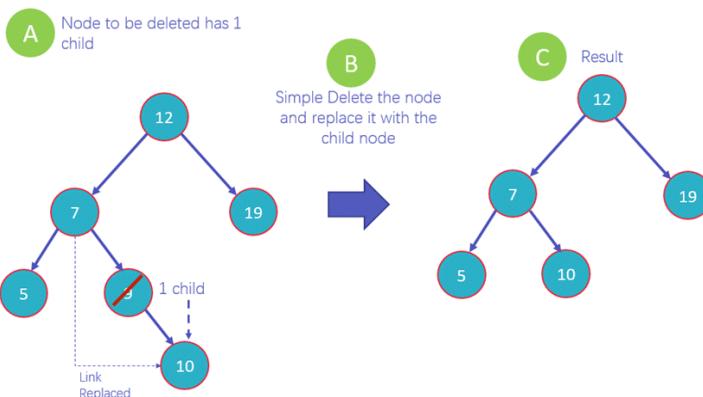


- **Borrar:**

- si el nodo a eliminar es una **hoja** → actualizar quien lo apunta a él
 - padre apunta a null ahora

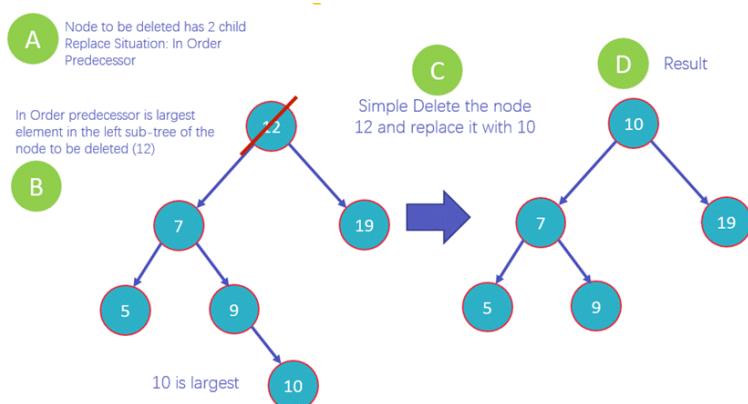


- si el nodo a eliminar **tiene un solo hijo** → actualizar quien lo apunta a él para que en vez apunte al hijo

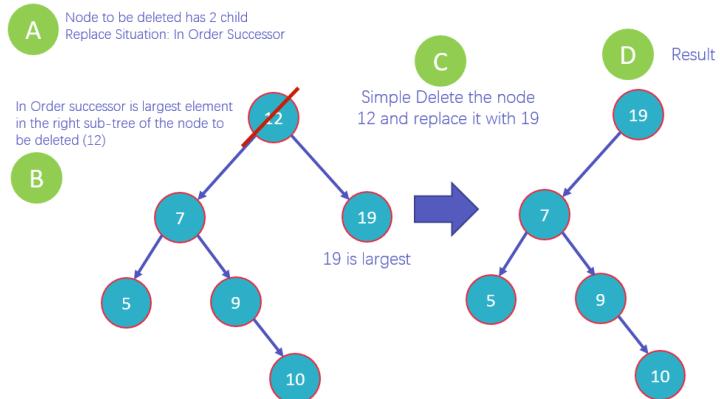


- si el nodo a eliminar **tiene dos hijos:**

1. se lo reemplaza por un nodo **lexicográficamente adyacente**
 - a. **su predecessor inorder**, osea el mas grande de los nodos de su subárbol izquierdo



- b. **su sucesor inorder**, osea el mas chico de los nodos de su subárbol derecho



2. se borra al nodo que lo reemplazó (seguro que dicho nodo tiene a lo sumo un solo hijo, sino no sería lexicográficamente adyacente, y por lo tanto es fácil de borrar)

Arboles Binarios AVL

Un AVL es un BST donde en cada nodo la diferencia(módulo) de alturas entre sus 2 subárboles es **a los sumo 1**.

bien formado $\rightarrow O(\log n)$

Factor balance: es la diferencia entre las alturas del árbol derecho y el izquierdo:

$$\text{balance factor} = \text{altura subárbol izquierdo} - \text{altura subárbol derecho}$$

Si el factor de equilibrio de un nodo es:

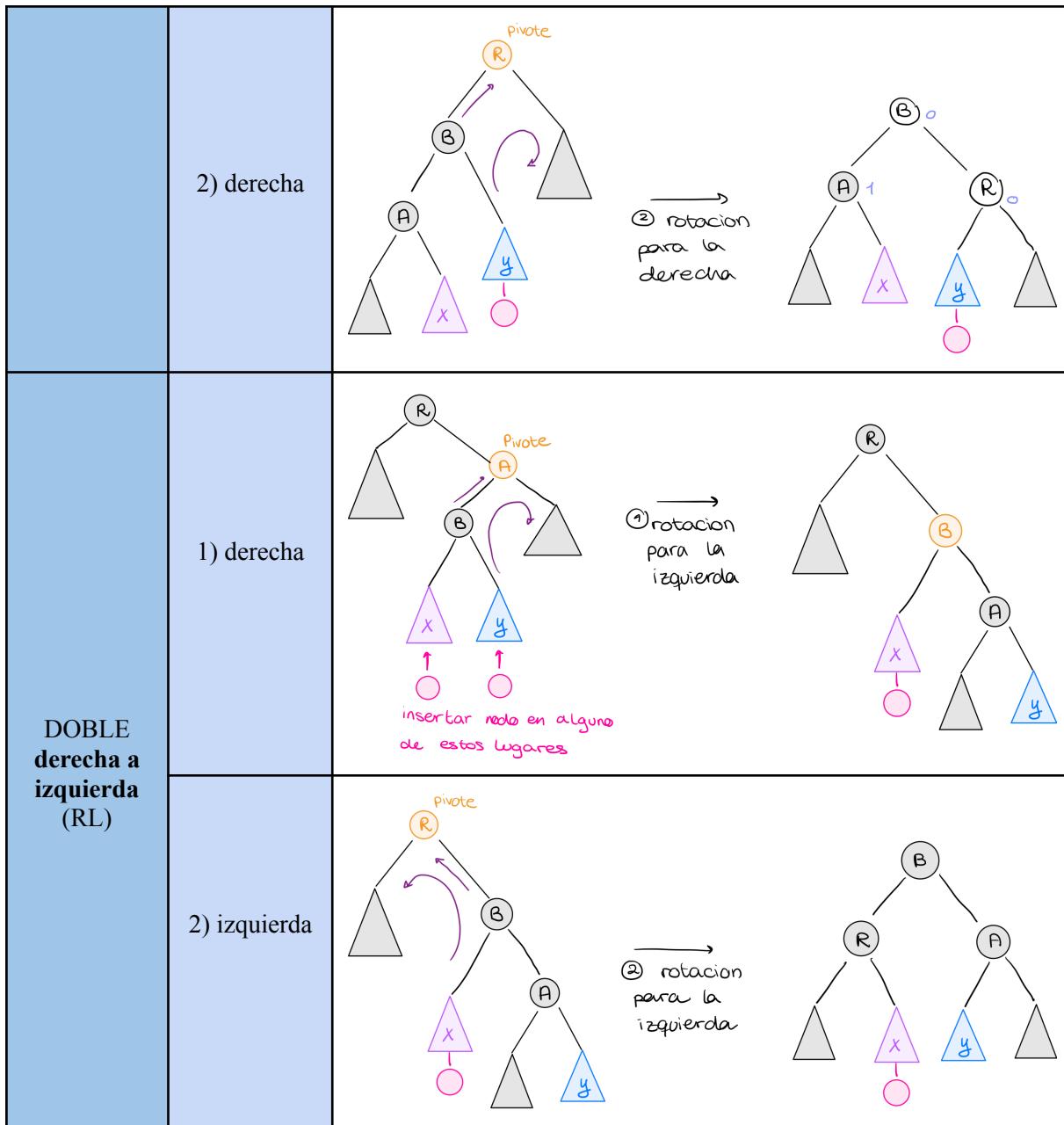
- 0: El nodo está equilibrado y sus subárboles tienen exactamente la misma altura.
- 1: El nodo está equilibrado y su subárbol derecho es un nivel más alto.
- -1: El nodo está equilibrado y su subárbol izquierdo es un nivel más alto.

Operaciones:

- insertar:

1. se inserta en el BST
2. si esta desbalanceado se aplican rotaciones
 - a. si se inserta a la izquierda de un nodo con factor de balance 1, ese nodo va a tener factor balance de 2 y deja de ser AVL
 - b. si se inserta a la derecha de un nodo con factor -1, ese a tener factor de balance -2 y deja de ser un AVL
3. rotaciones

SIMPLE	Derecha	<p>Pivote ALTURA PICTURA = i ALTURA = d insertar nodo en alguno de estos lugares</p> <p>rotacion SIMPLE HACIA DERECHA</p> <p>BF E-E=0 1o-1=0 d-d=0 E</p>
	Izquierda	<p>Pivote ALTURA PICTURA = i ALTURA = d insertar nodo en alguno de estos lugares</p> <p>rotacion SIMPLE HACIA IZQUIERDA</p> <p>BF E-E=0 d-d=0 1o-1=0 E</p>
DOBLE izquierda a derecha (LR)	1) izquierda	<p>Pivote ALTURA PICTURA = i ALTURA = d insertar nodo en alguno de estos lugares</p> <p>① rotacion para la izquierda</p>



- Borrar:

- **Realizar el Recorrido Inorder:**
 - Inicia el recorrido desde la raíz del árbol y desciende por la rama izquierda hasta llegar al nodo más a la izquierda.
 - Procesa el nodo actual.
 - Continúa el recorrido hacia la rama derecha del nodo actual.
 - Repite estos pasos hasta visitar todos los nodos en orden ascendente.
- **Eliminar los Nodos Durante el Recorrido:**
 - Para cada nodo visitado durante el recorrido inorder, realiza los siguientes pasos:
 - Si el nodo actual es el nodo que deseas eliminar, entonces elimínalo según los casos que se describen a continuación.
 - Si el nodo actual no es el que deseas eliminar, simplemente continúa con el recorrido.
- **Eliminar un Nodo:**
 - Encuentra el nodo que se desea eliminar.

- Si el nodo tiene al menos un hijo nulo, simplemente elimina el nodo y reemplázalo con su único hijo (si tiene uno).
- Si el nodo tiene dos hijos, encuentra el sucesor inorden (el nodo más pequeño en su subárbol derecho), copia su valor al nodo actual y luego elimina el sucesor inorden.
- Mantener la Propiedad AVL:
 - Después de eliminar un nodo, es posible que la propiedad AVL se vea comprometida, ya que la altura de los nodos ancestros puede haber cambiado.
 - Realiza rotaciones (simples o dobles) para restaurar la propiedad AVL.
 - Actualiza las alturas de los nodos afectados despues de cada rotación.
- Rebalanceo durante el Retorno:
 - Durante el retorno desde la recursión (si estás utilizando una implementación recursiva), asegúrate de actualizar las alturas de los nodos y realizar rotaciones según sea necesario para mantener el equilibrio.

Complejidad temporal	search	$O(\log h)$ $h = \text{altura de árbol}$
	insert	
	delete	
complejidad espacial	$O(n)$ $n = \text{cantidad de nodos}$	

⚠ IMPORTANTE PEOR DESBALANCE POSIBLE → Arbol fibonacci

1. Fibonacci de orden 0 es el árbol nulo.
2. Fibonacci de orden 1 es un nodo.
3. Fibonacci de orden h es un árbol que tiene:
 - a. Como hijo izquierdo un Fibonacci de orden $h - 1$.
 - b. Como hijo derecho un Fibonacci de orden $h - 2$.

Sea un AVL de altura h con la menor cantidad de nodos posibles en esa altura. Este AVL de altura h tendrá la siguiente cantidad de nodos:

$$1 \text{ nodo} + (\text{cantNodos AVL altura } h - 1) + (\text{cantNodo AVL altura } h - 2)$$

fibonacci:

$$f(h) = \begin{cases} 0 & \text{is the null tree,} \\ \text{Fibonacci of order 1} & \text{is a node,} \\ \text{Fibonacci of order } h \geq 2 & \begin{cases} \text{a) as the left child} & = h - 1, \\ \text{b) as the right child} & = h - 2. \end{cases} \end{cases}$$

$$\text{fibonacci}(n) \geq \frac{a^n}{\sqrt{5}}$$

a is the golden number $a = \frac{1 + \sqrt{5}}{2}$

AVL altura h con menos cantidad de nodos	Cant de nodos en relación a los números de Fibonacci
$H = 0$ Cant de Nodos = 1	$Fibo(H + 3) - 1 = Fibo(3) - 1 = 1$
$H = 1$ Cant de Nodos = 2	$Fibo(H + 3) - 1 = Fibo(4) - 1 = 2$
$H = 2$ Cant de Nodos = $1 + 1 + 2 = 4$	$Fibo(H + 3) - 1 = Fibo(5) - 1 = 4$
$H = 3$ Cant de Nodos = $1 + 4 + 2 = 7$	$Fibo(H + 3) - 1 = Fibo(6) - 1 = 7$
$H = 4$ Cant de Nodos = $1 + 7 + 4 = 12$	$Fibo(H + 3) - 1 = Fibo(7) - 1 = 12$
$H = 5$ Cant de Nodos = $1 + 12 + 7 = 20$	$Fibo(H + 3) - 1 = Fibo(8) - 1 = 20$
...	
$H = h$ Cant de Nodos = ?	$Fibo(h + 3) - 1$

Arboles Binarios Red-Black tree

Los Red Black Tree son árboles balanceados que siguen las siguientes reglas:

- Todo los nodos son negros o rojos.
- La raíz es negra, al igual que los nil o null.
- Si un nodo es rojo, no puede tener hijos rojos.
- Todo camino tienen que tener la misma cantidad de nodos negros.

operaciones:

- inserción:

●●OBS: los nodos que se insertan siempre son rojos

- Hay cuatro casos posibles para la inserción:
 - Si el nodo a insertar es la raíz → se pinta de negro.
 - Si el tío del nodo a insertar es rojo → se re-colorea el padre, el abuelo y el tío.
 - Si el tío del nodo a insertar es negro (forma un triangulo) → se rota el padre del nodo a insertar.
 - Si el tío del nodo a insertar es negro (forma un linea) → se rota el abuelo del nodo a insertar y re-colorear padre y abuelo originales.

💡 Hay tres casos a considerar después de la inserción básica:

- Caso 1: Si el padre del nodo recién insertado es **negro**, entonces el árbol sigue siendo **válido**.
- Caso 2: Si el padre del nodo recién insertado es **rojo** y el **tío** también es rojo, **recolorea** los nodos para mantener las propiedades.
- Caso 3: Si el padre es **rojo** y el **tío** es **negro o inexistente**, realiza **rotaciones y recolorea** para restaurar el equilibrio.

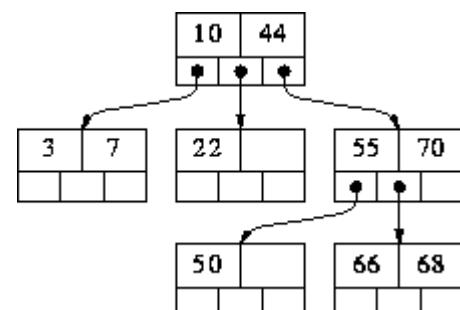
- eliminar:
 - Eliminación Básica del Árbol de Búsqueda Binaria:
 - Realiza una eliminación estándar en el árbol de búsqueda binaria, manteniendo el orden del árbol.
 - Identificar el Caso de Eliminación:
 - Identifica el caso de eliminación basado en la cantidad de hijos del nodo eliminado y el color del nodo reemplazo.
 - Aplicar Reglas de Equilibrio:
 - Realiza rotaciones y ajustes de colores para mantener las propiedades del Árbol Rojo-Negro después de la eliminación.
 - Hay varios casos a considerar, como el caso en el que el nodo eliminado o su reemplazo es rojo, o cuando ambos son negros.

Complejidad temporal	search	O(log n)
	insert	
	delete	

Arboles Binarios k-ario

Árbol Multicamino

En los árboles multicamino k-ario, los nodos **guardan k-1 claves** de información, con un **máximo de k hijos**. Cada clave C_i de un cierto nodo será tal que las claves almacenadas en su subárbol izquierdo son menores y las almacenadas en su subárbol derecho serán mayores



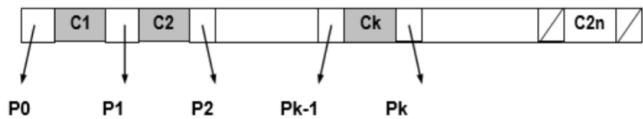
(foto sería arbol 3-ario → 2 claves en cada nodo y mx 3 hijos)

Arboles Binarios B-tree

Un Árbol B de Orden N es aquel que cumple:

- Cada nodo contiene a lo sumo **2n claves**.
- Cada nodo, **excepto la raíz**, contiene **por lo menos N claves**.
- Cada nodo o es hoja o tiene **M + 1 descendientes** donde **M** es el **número de claves** que posee realmente ese nodo (lugares ocupados).
- **Todas las hojas están al mismo nivel**.

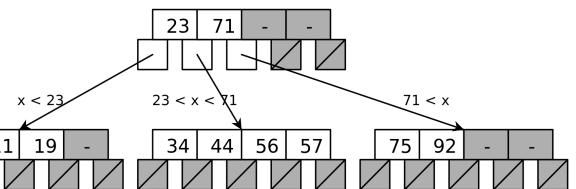
Un nodo genérico de un árbol B de orden N será de la forma:



Operaciones:

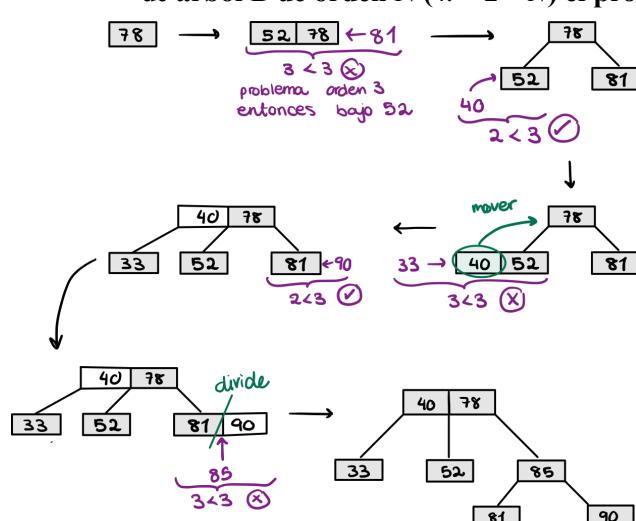
- búsqueda:

- 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:
 - i. Si $X < C_i$, 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 .
 - ii. Si $X = C_i$ para algún $i \leq k$ entonces lo encontré.
 - iii. Si $C_i < X < C_{i+1}$ para algún i prosigo en la búsqueda en el subárbol apuntado por P_i .
 - iv. Si $C_k < X$ siendo k la cantidad de claves que posee, entonces sigo la búsqueda en el subárbol apuntado por C_k .
 - v. Si en algún caso el puntero por donde hay que seguir la búsqueda fuera null, entonces el elemento buscado no está.



- inserción:

- Si se quiere insertar una clave X en un árbol B de orden N, se procede de la siguiente manera:
 - i. La inserción siempre se hace en las hojas (para poder detectar si el nodo a insertar ya está presente o no).
 - ii. Para insertar se coloca el elemento X en la hoja que corresponda (el nodo debe estar ordenado).
 - iii. 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 se repite.**



- **borrado:**

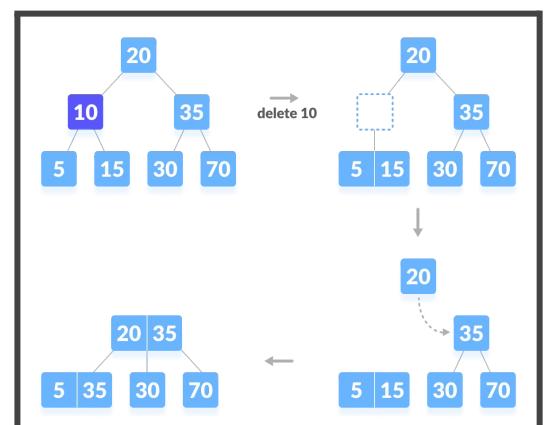
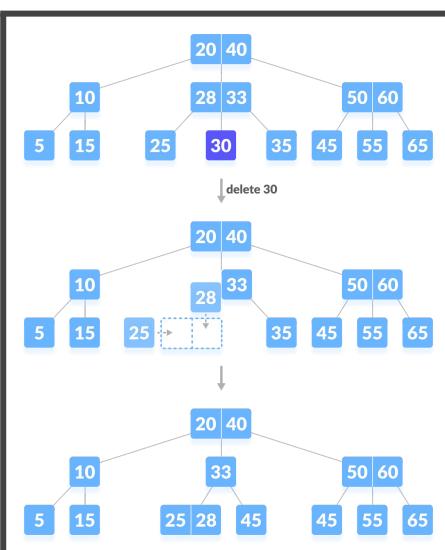
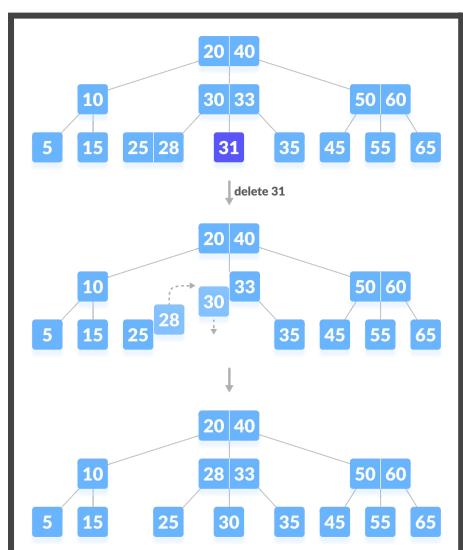
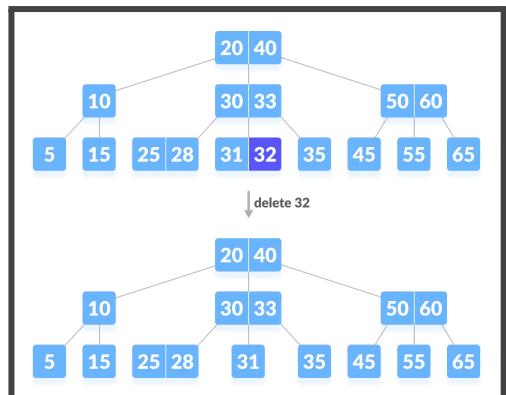
- Si no se encuentra en un nodo hoja, se lo reemplaza por una clave lexicográficamente adyacente, por ejemplo el sucesor in order y se lo elimina de dicha hoja.

i. Si fuera hoja se elimina directamente.

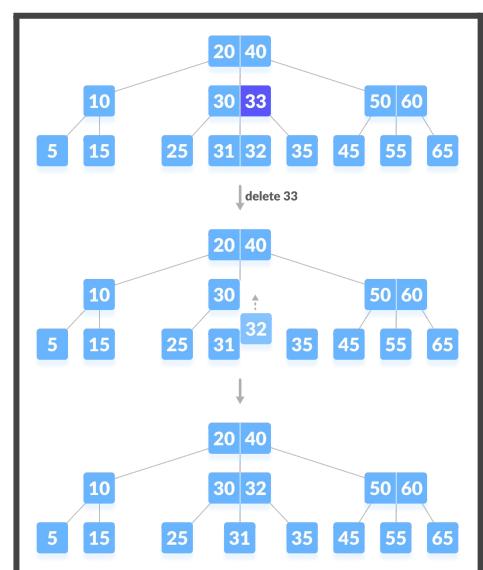
- Luego, para la hoja que colaboró en el borrado se analiza si cumple las condiciones de árbol B de orden N.

i. 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:

- si no, se lo **particiona subiendo** el elemento del **medio**,
- Despues se analiza qué sucede con el nodo donde estaba su medio antecesor, y se sigue el proceso **recurrentemente** hasta llegar a la raíz.



	Complejidad temporal
Search	$O(\log n)$
Insert	$O(\log n)$
Delete	$O(\log n)$



Grafos

Un grafo es un conjunto de nodos (o vértices) y ejes (o aristas). Cada arista conecta dos nodos.

Características (repaso mate discreta)

Propiedades:

- Cada arista puede tener un valor **asociado**, lo que lo convierte en un **grafo trivial**
- Si las **aristas pueden recorrer en un único sentido**, se representan con una **flecha** y el
- grafo se llama **digrapho o grafo dirigido**.
- Si entre dos nodos puede **haber más de una arista**, el grafo se llama **multigrafo**.
- Cuando una arista conecta a un nodo con sí mismo, se llama un **lazo**.

Grafo trivial

$$\bullet \quad \{ \#V = 1 \wedge \#E = 0 \}$$

Lazo



Arista propia



Simple



Multigrafo



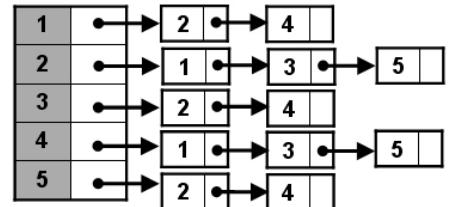
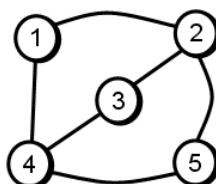
Los grafos pueden ser representados a través de:

- **Matriz de adyacencia:** donde cada fila y columna corresponden a los vértices, y la entrada en la posición (i, j) indica si hay una arista entre los vértices i y j , siendo 1 para presencia de arista y 0 para ausencia.

complejidad temporal	si buscamos si hay aristas de $v \rightarrow u$	$O(1)$
	si buscamos que aristas hay en v	$O(n)$
complejidad espacial	$O(n*m)$ n = cantidad de vértices m = cantidad de aristas	

- **Lista de adyacencia:** Cada vértice tiene una lista de vértices los cuales son adyacentes a él.

es un array de listas



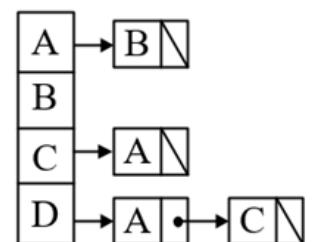
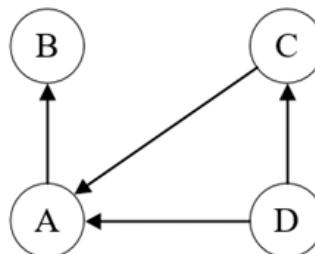
complejidad temporal	si buscamos si hay aristas de $v \rightarrow u$	$O(n)$
	si buscamos que aristas hay en v	$O(1)$
complejidad espacial	$O(n + m)$ n = cantidad de vértices m = cantidad de aristas	

- **Matriz de incidencia:** donde las filas representan los vértices y las columnas representan las aristas, y la entrada en la posición (i, j) indica si el vértice i está incidente (conectado) a la arista j , siendo **-1 si el vértice es el extremo inicial de la arista, 1 si es el extremo final(entrante), y 0 si no está conectado a esa arista.**

complejidad temporal	si buscamos si hay aristas de $v \rightarrow u$	$O(1)$
	si buscamos que aristas hay en v	$O(n)$
complejidad espacial	$O(n*m)$ $n =$ cantidad de vértices $m =$ cantidad de aristas	

- **Lista de incidencia:** Las aristas son representadas con un vector de pares (ordenados, si el grafo es dirigido), donde cada par representa una de las aristas.

es un array de listas



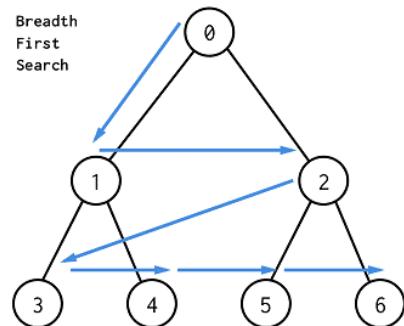
complejidad temporal	si buscamos si hay aristas de $v \rightarrow u$	$O(n)$
	si buscamos que aristas hay en v	$O(1)$
complejidad espacial	$O(n + m)$ $n =$ cantidad de vértices $m =$ cantidad de aristas	

Formas de recorrer un grafo

BFS(Breadth-first Search)

Una búsqueda en anchura (BFS) es un algoritmo de búsqueda para lo cual recorre los nodos de un grafo, **comenzando en la raíz (eliendo algún nodo como elemento raíz en el caso de un grafo)**, para luego **explorar todos los vecinos de este nodo**.

- se recorre “de a niveles”
- Se empieza marcando el nodo inicial y luego en cada paso se recorren y marcan los nodos no marcados que son adyacentes a un nodo marcado.



v: the total number of vertices in the graph

Complejidad temporal: $O(v+e)$

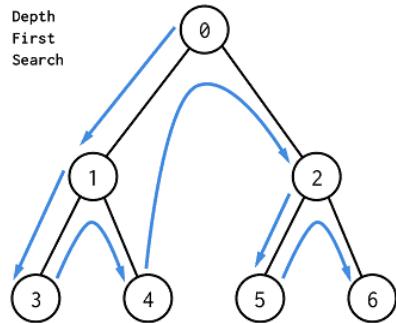
e: the total number of edges in the graph

Complejidad espacial: $O(v)$

DFS(Depth-first Search)

Recorre el grafo revisando primero el nodo actual y moviéndose después a uno de sus sucesores para repetir el proceso. Si el nodo actual no tiene sucesor a revisar, regresamos a su predecesor y el proceso continúa (moviéndose a otro sucesor)

- se recorre “por profundidad”



v: the total number of vertices in the graph

Complejidad temporal: $O(v+e)$

e: the total number of edges in the graph

Complejidad espacial: $O(v)$

Dijkstra

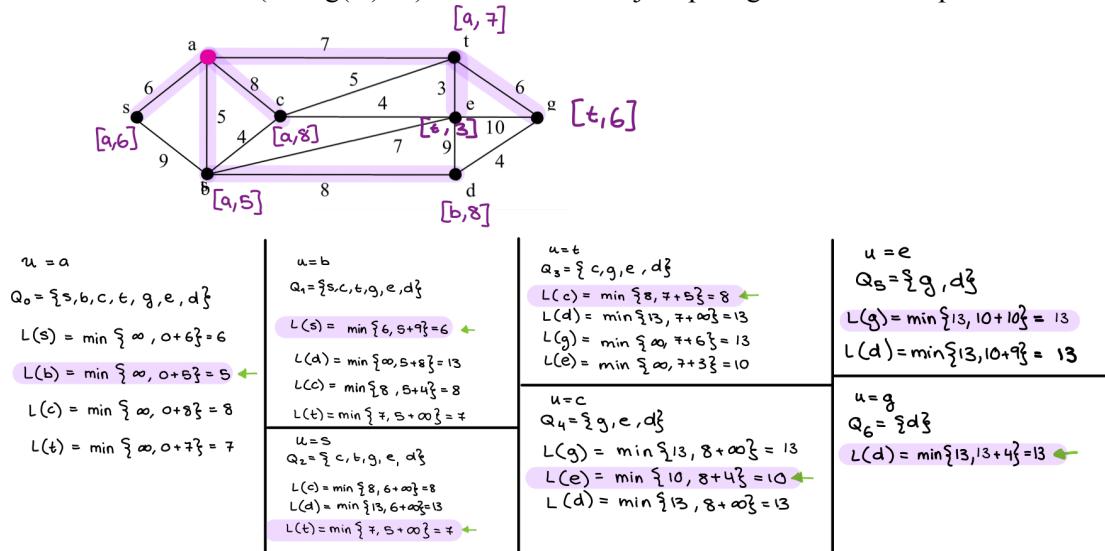
- objetivo: encontrar el camino mas corto entre cualquier par de nodos
- requisito → grafo con pesos no negativos

Los pasos del algoritmo de Dijkstra son:

1. Empezamos marcando todos los nodos con infinito excepto el nodo inicial. Este número representa el menor costo conocido de llegar hasta ese nodo.
2. En cada paso, miramos al nodo con menor costo que aún no haya sido visitado y actualizamos el costo de sus vecinos.

La complejidad de este algoritmo es $O((N+E)*\log(N))$ siendo N la cantidad de nodos y E la cantidad de aristas.

- La E es porque cada arista se recorre 2 veces (1 por cada nodo que conecta) y N surge de que cada nodo se recorre una única vez.
- Los $\log(N)$ sumados a ambas letras salen de que cada vez que se analiza un nodo hay que hacer un remove de la PriorityQueue que tiene costo $\log(N)$ y los add de la PQ se terminan haciendo E veces.
- Existen formas de modificar el algoritmo para que la complejidad en el peor de los casos termine siendo $O(N*\log(N)+E)$ lo cual es una mejora para grafos densos especialmente.



caminos $\langle a,s \rangle, \langle a,b,d \rangle, \langle a,c \rangle, \langle a,t,g \rangle, \langle a,t,e \rangle$

Tipos de algoritmos-heurísticas

Greedy

Es una 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.

complejidad temporal	$O(n \log n)$ or $O(n)$
complejidad espacial	$O(1)$ or $O(n)$

Backtracking

estrategia para encontrar soluciones a problemas que satisfacen restricciones.

la idea es encontrar la mejor combinación posible en un momento determinado.

complejidad temporal	$O(n \log n)$ or $O(n)$
complejidad espacial	$O(n^2)$ or $O(n^3)$

Stack o Queue

Calcula todas las posibles soluciones. Si se agregan restricciones, recién en el final se evalúa cual es la mejor. Es decir, en las hojas se evalúa si se satisface la restricción pedida

Divide and conquer

Es una 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.

Ejemplo: Mergesort, Quicksort, búsqueda en un BST, búsqueda en un arreglo ordenado.

complejidad temporal	$O(n \log n)$ or $O(n^2)$
complejidad espacial	$O(n \log n)$ or $O(n^2)$

Exhaustiva

Técnica para buscar todas las posibles soluciones explorando el espacio de soluciones en forma implícita. Sirve para problemas que tienen muchas soluciones.

Idea para esta técnica (Típicamente recursiva):

1. Si el nodo no puede expandirse más (no hay más opciones a partir de él), entonces retornar/imprimir el resultado.
2. Si no, por cada posibilidad para ese nodo de expandir un próximo 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.