

# Programación 3

Nombre: Santiago Vietto

Docente: Juan José Vulcano

Institución: UCC

Año: 2020

# Recursividad

\_ La recursión es una de las herramientas más poderosas para resolver problemas en computación. Una definición recursiva dice cómo obtener conceptos nuevos empleando el mismo concepto que intenta describir, a esto lo llamamos definición inductiva, es decir tiene la característica de definirse sobre sí mismo. Los algoritmos recursivos ofrecen soluciones estructuradas, modulares y elegantemente simples.

\_ Es un concepto complejo, potente, la solución termina siendo simple y es autorreferencial, donde esta última es una característica de la recursión.

\_ Una definición recursiva dice cómo obtener conceptos nuevos empleando el mismo concepto que intenta definir. El poder de la recursividad es que los procedimientos o conceptos complejos pueden expresarse de una forma simple. Un razonamiento recursivo tiene dos partes: la base y la regla recursiva de construcción.

- Base: es lo único que no es recursiva y es el punto tanto de partida como de terminación de la definición o secuencia recursiva que es repetitiva.
- Regla recursiva de construcción: parte que se repite varias veces.

\_ Las funciones recursivas siempre consumen mucha memoria.

## Ejemplo recursión: Factorial

\_ Para calcular el factorial de un numero deberíamos multiplicar todos los números menores al del número que queremos calcular, por ende, viendo el ejemplo vemos que para calcular el factorial de 4, multiplicamos 4 por todos los menores que le siguen. A continuación, vemos la función que lo calcula:

Definición:

$\text{factorial}(n) = n!$  si  $n > 0$

$n! = n * n-1 * n-2 * \dots * 1$  si  $n > 0$

el valor de  $0!$  se define como

$\text{factorial}(0) = 0! = 1$

O sea que

$4! = 4 * 3 * 2 * 1$

\_ A continuación vemos una solución iterativa, donde asignamos a una variable el valor 1 y luego hacemos un bucle for desde 4 hasta 1, reduciendo de uno en uno, vamos a ir multiplicando el 1 inicial por la variable n que es 4, obteniendo como resultado el valor 24.

$\text{prod} = 1$

repetir desde n hasta 1 (-1)

$\text{prod} = \text{prod} * n$

\_ En el caso de un algoritmo recursivo, tenemos que si  $n$  (número a calcular el factorial) es igual a 0, entonces el resultado del factorial va a ser 1, ya que por definición el factorial de 0 es 1. Pero si  $n$  no vale 0, vamos a ejecutar una secuencia donde el producto va a ser igual a  $n$  por la función Factorial de  $n - 1$ , es decir, volvemos a llamar a la función recursiva para resolver un factorial, pero no de 4 sino de 3 y así sucesivamente.

```
Si  $n == 0$   
     $prod = 1$   
Sino  
     $prod = n * \text{Factorial}(n - 1)$ 
```

\_ En el caso de este ejemplo, la base es la parte que indica que si  $n$  es igual a 0 entonces el factorial vale 1. Esta definición no es recursiva. Pero luego viene la secuencia recursiva donde volvemos a usar la definición de la función para resolver el problema del factorial valiéndonos del mismo factorial.

\_ Entonces estamos resolviendo la solución al problema del factorial utilizando una redefinición el uso nuevamente de la función Factorial. Usamos la misma función para poder resolver el problema.

## Conceptos

Recursividad de cola: esta es cuando una llamada recursiva es la última posición ejecutada del procedimiento, también se conoce como recursividad de extremo final o recursión de extremo de cola. Es decir, cuando la estructura recursiva clave está al final de la secuencia, de lo contrario se dice recursividad de inicio.

Recursividad directa: esta es cuando un procedimiento o función incluye una llamada a sí mismo, por ejemplo, factorial vuelve a llamar a factorial.

Recursividad indirecta: esta es cuando un procedimiento o función llama a otro procedimiento y este, causa que el procedimiento original sea invocado. Por ejemplo, la función factorial llama a la función multiplicación y luego esta última llama a la función factorial de nuevo y así sucesivamente.

\_ Al principio algunas personas se sienten un poco incómodas con la recursividad, tal vez porque da la impresión de ser un ciclo infinito, pero en realidad es menos peligrosa una recursión infinita que un ciclo infinito, ya que una recursividad infinita pronto se queda sin espacio y termina el programa, mientras que la iteración infinita puede continuar mientras no se termine en forma manual. La recursividad no va a ser infinita porque siempre hay un caso base que es el que nos va a dar el final del juego.

\_ Cuando un procedimiento recursivo se llama recursivamente a sí mismo varias veces, para cada llamada se crean copias independientes de las variables declaradas en el procedimiento. Cuando nosotros mandamos un parámetro por ejemplo 4 en la función

Factorial, llamamos a la función, el parámetro ingresa como n y calcula nuevamente el factorial de  $n - 1$  que sería 3, donde al hacer esta llamada, llamamos nuevamente a la función Factorial. En la memoria de la computadora lo que sucede es que se activa una nueva función Factorial donde ahora el nuevo parámetro de ingreso es 3 y no es el 4 anterior, sino que esta función está en otra región de la memoria, donde hay otra n que no es la misma que la original en otra zona de la memoria, y así sucesivamente hasta que n sea igual a 0 y acá es cuando la solución el factorial de 0 es 1, por eso en el Factorial de 1 la solución es el factorial de uno por uno, luego en el Factorial de 2 tenemos que la solución es dos por uno, y así sucesivamente hasta que la variable Num toma el valor de 24.

Escribimos código

...

...

...

Num = FACTORIAL ( 4 )

FUNCION FACTORIAL ( n )    n = 4

SI n == 0

    RETURN ( 1 )

Sino

    RETURN ( 4 \* FACTORIAL( 3 ) )

FUNCION FACTORIAL ( n )    n = 3

SI n == 0

    RETURN ( 1 )

Sino

    RETURN ( 3 \* FACTORIAL( 2 ) )

FUNCION FACTORIAL ( n )    n = 2

SI n == 0

    RETURN ( 1 )

Sino

    RETURN ( 2 \* FACTORIAL( 1 ) )

FUNCION FACTORIAL ( n )    n = 1

SI n == 0

    RETURN ( 1 )

Sino

    RETURN ( 1 \* 1 )

FUNCION FACTORIAL ( n )    n = 0

SI n == 0

    RETURN ( 1 )

Sino

    RETURN ( n \* FACTORIAL( n -1 ) )

\_ Entonces tenemos que entender como primer concepto que cuando la función Factorial llama a otra función Factorial, está llamando a otra función en otra posición de la memoria y no se mezclan las n, porque cada una es una variable nueva y en otra posición de memoria, para resolver otro problema que justo es con la función Factorial y esa es la coincidencia.

\_ Hay situaciones en las que iterativamente se va a volver muy complejo de resolver, pero recursivamente va a ser muy simple, con la desventaja de que siempre consumen mucha memoria. Y en algunos casos las funciones iterativas nos complican más, pero no consumen tanta memoria.

\_ Las funciones recursivas siempre tienen una base y una regla recursiva, por ende, siempre al tratar de resolver un problema recursivo deberíamos tratar de pensar cual es la base y cuál es la parte recursiva.

\_ Un conjunto de objetos está definido recursivamente siempre que: algunos elementos del conjunto se especifican explícitamente y aseguren el final o corte (base), y el resto de los elementos del conjunto se definen en términos de los elementos ya definidos (regla recursiva).

Base: la secuenciación, iteración condicional y selección son estructuras válidas de control que pueden ser consideradas como enunciados.

Regla recursiva: las estructuras de control que se pueden formar combinando de manera válida la secuenciación iteración condicional y selección también son válidos.

\_ Para resumir brevemente el procedimiento, en la recursividad el procedimiento se llama a sí mismo, el problema se resuelve resolviendo el mismo problema, pero más simple, pero de tamaño menor, y la manera en la cual el tamaño del problema disminuye asegura que el caso base eventualmente se alcanzará.

\_ Podemos tratar de resolver erróneamente el factorial donde sumar de manera continua 1 a la variable n no conduce a un final definido.

$$n! = (n + 1)! / (n + 1)$$

\_ Tenemos que intentar que el procedimiento de recursividad, haga que yo este llegando cada vez con más chances a la base y no que nos alejemos de esta.

Ejemplo recursión: multiplicación de números naturales

\_ A continuación vemos una función por si no sabemos multiplicar:

$$\begin{array}{ll} a * b = a & \text{si } b = 1 \\ a * b = a * (b - 1) + a & \text{si } b > 1 \end{array}$$

O sea que:

$$6 * 3 = 6 * 2 + 6 = 6 * 1 + 6 + 6 = 6 + 6 + 6 = 18$$

\_ Pensando en hacer una función recursiva, tenemos a la función Multiplicación de a por b, con el caso base de que si b vale 1 el resultado es a, y si no es multiplicar a por b - 1 y sumarle a ese número a.

**FUNCION Multiplic( a, b )**

**Si b = 1**

**prod = a**

**Sino**

**prod = Multiplic( a, b - 1 ) + a**

### Ejemplo recursión: Secuencia de Fibonacci

\_ Analizamos la función y vemos que, si  $n$  vale 0 o 1, el Fibonacci es  $n$ . Ahora si  $n$  es mayor o igual a 2, el Fibonacci de  $n$  es la función Fibonacci de  $n - 2$  más el Fibonacci de  $n - 1$ , de forma análoga decimos que si  $n = 2$  entonces  $(2 - 2) + (2 - 1)$ .

La secuencia de Fibonacci

0, 1, 1, 2, 3, 5, 8, 13, 21, 34, ...

Cada elemento es esta secuencia es la suma de los dos precedentes.

Ej.  $0 + 1 = 1$ ,  $1 + 1 = 2$ ,  $1 + 2 = 3$ ,  $2 + 3 = 5$ , ...

Entonces podemos definir la secuencia como:

$$\begin{aligned} \text{fib}(n) &= n && \text{si } n = 0 \text{ o si } n = 1 \\ \text{fib}(n) &= \text{fib}(n - 2) + \text{fib}(n - 1) && \text{si } n \geq 2 \end{aligned}$$

\_ A continuación tenemos la solución iterativa donde cómo podemos ver es bastante enredado:

```
FUNCION Fib( n )
Si n <= 1
    Salida = n;
loFib = 0;
hiFib = 1;
Repetir i = 2; i <= n
    x = loFib;
    loFib = hiFib;
    hiFib = x + loFib;
Salida = hiFib
```

\_ Pero ahora con el algoritmo recursivo, tenemos que, si  $n$  es menor o igual a 1, la función es igual a  $n$ , y si no es Fibonacci de  $n - 2$  más Fibonacci de  $n - 1$  y no sucede más nada:

```
FUNCION Fib( n )
Si n <= 1
    Fib = n
Sino
    Fib = Fib( n - 2 ) + Fib( n - 1 )
```

\_ Este método de solución es más caro en cuanto a consumo de memoria.

### Ejemplo recursión: Guía telefónica

\_ El objetivo es buscar un nombre dentro de una guía telefónica. El método de búsqueda menos refinado es el de búsqueda lineal o secuencial buscando por nombre y apellido hasta que encontremos el que queremos (si la lista no está ordenada, quizás este puede ser el único método para encontrar algo).

\_ Si pensamos recursivamente podemos decir que el caso base es si encontramos el nombre que queremos, y el caso repetitivo es dividir la guía a la mitad y nos fijamos si la letra que queremos está a la derecha o la izquierda, ya que dependiendo de eso seguiremos con una de las dos mitades, en donde la función se repite, pero ahora con una de las mitades y se divide está en dos nuevamente, y así hasta encontrar la letra. Podríamos tener el caso base también de que el nombre no esté en la guía y llegar a un resultado que no es el esperado. A continuación, tenemos la solución:

```

FUNCION Búsqueda( bajo, alto, x )
Si bajo >= alto
    salida = -1
medio = entero( (bajo + alto) / 2 )
Si x = a[ medio ]
    salida = medio
Si x < a[ medio ]
    Búsqueda( bajo, medio, x )
Sino
    Búsqueda( medio, alto, x )

```

\_ Suponiendo que bajo es 0 y alto 1000, en la primera parte si 0 y 1000 se encontraron y no está el nombre que buscamos entonces el resultado es que no está (-1), de lo contrario dividimos a la mitad, nos fijamos en el medio si esta, en caso de que no, nos fijamos si está a la mitad izquierda o derecha llamando las funciones.

Ejemplo recursión: Algoritmo de Euclides para MCD

Deseamos obtener el MCD entre dos números positivos mayores a 0.

Esto se logra fijando las siguientes reglas:

$$\begin{aligned} \text{mcd}(a, b) &= a & \text{si } b &= 0 \\ \text{mcd}(a, b) &= \text{mcd}(b, c) & \text{si } b > 0 \\ &\text{donde } c = \text{resto de la división entre } a \text{ y } b \end{aligned}$$

\_ A continuación vemos la solución recursiva:

**FUNCION** mcd( a, b )

Si b = 0

mcd = a

Sino

div = entero( a / b )

c = a - div \* b

mcd( b, c )

**Resultados Ejemplo**

mcd (25, 5) = mcd (5, 0) = 5

mcd (21, 6) = mcd (6, 3) = mcd (3, 0) = 3

mcd (57, 23) = mcd (23, 1) = mcd (1, 0) = 1

mcd (35, 16) = mcd (16, 3) = mcd (3, 1) = mcd (1, 0) = 1

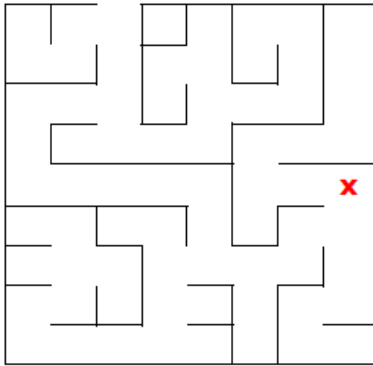


## Propiedades de las definiciones o algoritmos recursivos

- No debe generar una llamada infinita a sí mismo.
- Una función recursiva  $f$  debe definirse en términos que no impliquen a  $f$  al menos en un argumento o grupo de argumentos.
- Debe existir una salida de la secuencia de llamadas recursivas.
- Cualquier caso de definición recursiva tiene que reducirse a la larga a alguna manipulación de uno o casos más simples no recursivos.

### Ejemplo recursión: Laberinto

\_ El objetivo del juego consiste en desplazar la persona (x) de una a otra celda del laberinto hasta encontrar la salida. Se asume que las rayas negras se corresponden con paredes y no pueden ser atravesadas por la persona. No es objetivo encontrar el camino más corto, solo importa encontrar un camino. Proponemos llamar a las celdas que tienen una pared cerrada con un número. Las celdas que tienen más de una pared cerrada, se les asigna un número correspondiente a la sumatoria de las paredes cerradas.



\_ Si se logra el caso base dado en el pseudocódigo quiere decir que salí del laberinto. Cuando queremos encontrar la salida, no pensamos en el final, sino que pensamos en el paso que estamos dando, si en el paso que damos preguntamos si encontramos la salida, en el caso de que no damos otro paso y preguntamos nuevamente.

\_ El caso base es si encontramos la salida, ahora la secuencia repetitiva sería que en cualquier casillero en el que estemos tenemos que mirar cuáles son los movimientos factibles, luego lo que hacemos es movernos en alguna de esas direcciones y llamamos nuevamente a la función para que resuelva el problema que sigue hasta que hayamos encontrado la salida. Viendo en la posición del gráfico, la función puede ir a la izquierda o abajo, entonces prueba todos los pasos por la izquierda, cuando retorna que no se puede ir por ahí ahora intenta por abajo, pero siempre estuvimos en la función al principio de todo.

\_ A continuación vemos la solución:

El Programa: Recorrer( 8, 5, { } )

Recorrer( x, y, visitadas )

SI (x < 0) O (x > 8) O (y < 0) O (y > 8) ENTONCES

MostrarSalida( visitadas )

SINO

SI FueVisitada( visitadas, x, y )

RETORNO

visitadas = visitadas + { x, y }

SI matriz( x, y ) esta en lista( 1, 2, 3, 4, 5, 6, 7) ENTONCES

Recorrer( X, Y-1, visitadas) MOVER HACIA ARRIBA

SI matriz( x, y ) esta en lista( 1, 4, 5, 8, 9, 12, 13) ENTONCES

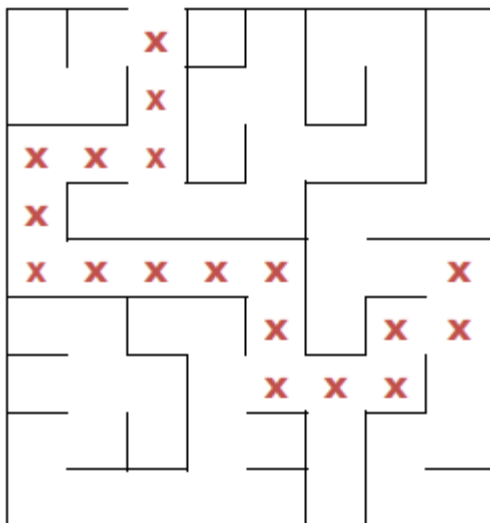
Recorrer( X, Y+1, visitadas) MOVER HACIA ABAJO

SI matriz( x, y ) esta en lista( 2, 4, 6, 8, 10, 12, 14) ENTONCES

Recorrer( X-1, Y, visitadas) MOVER HACIA IZQUIERDA

SI matriz( x, y ) esta en lista( 1, 2, 3, 8, 9, 10, 11) ENTONCES

Recorrer( X+1, Y, visitadas) MOVER HACIA DERECHA



{ 8, 5 }  
{ 8, 6 } { 3, 5 }  
{ 7, 6 } { 2, 5 }  
{ 7, 7 } { 1, 5 }  
{ 6, 7 } { 1, 4 }  
{ 5, 7 } { 1, 3 }  
{ 5, 6 } { 2, 3 }  
{ 5, 5 } { 3, 3 }  
{ 5, 5 } { 3, 2 }  
{ 4, 5 } { 3, 1 }  
....

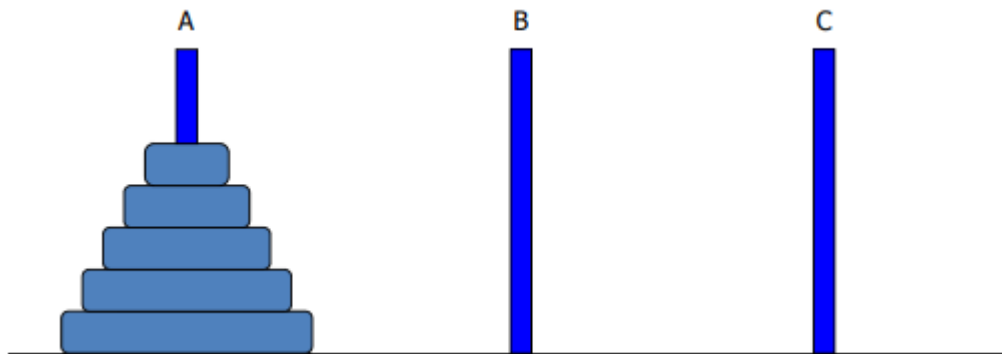
### Ejemplo recursión: Juego de ajedrez

\_ Nos preguntamos cómo haríamos un programa que juegue al ajedrez. No nos interesa que juegue muy bien. Deberíamos tener un listado de todas las movidas posibles y probar la que a nosotros nos interese, entonces movemos una pieza y llamamos a la función ajedrez, luego hacemos lo mismo para el rival llamando a la función ajedrez y así sucesivamente hasta que uno pierda o gane.

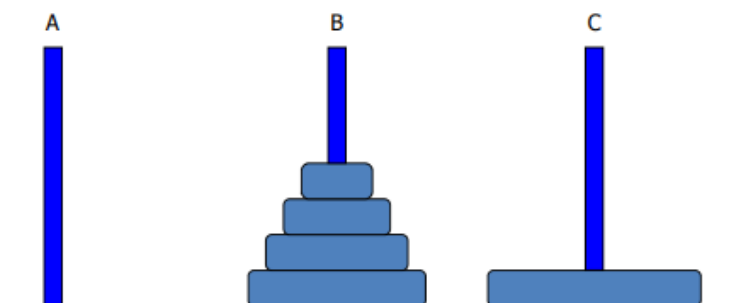
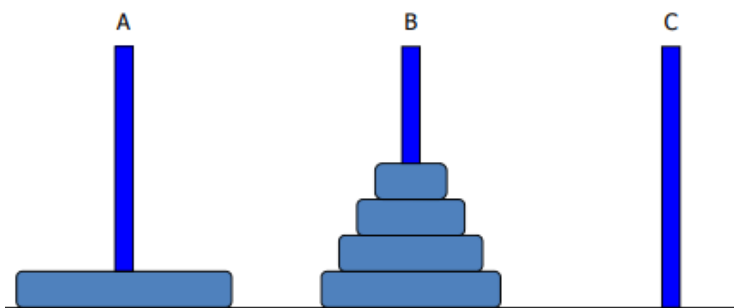
\_ Entonces los casos bases son si comemos el rey ganamos, si nos comen el rey perdimos, si pasa tal cosa empate, etc. Y si no pasa esto, vemos todas las movidas y llamamos nuevamente la función.

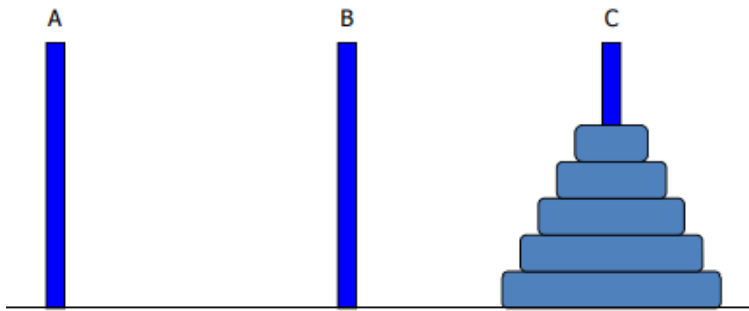
#### Ejemplo recursión: Torres de hanoi

\_ El objetivo del juego es desplazar de a una las piezas del poste A al poste C, usando como soporte el poste B, de manera tal que nunca una pieza de tamaño menor se encuentre debajo de una pieza de tamaño mayor. El modelo se plantea con 5 piezas, pero deberíamos generalizarlo para  $n$  piezas.



\_ Si tenemos  $n$  piezas, vamos a mover  $n-1$  piezas del poste A al poste B, luego movemos la nueva primer pieza de A al poste C, donde luego volvemos a mover  $n-1$  piezas del poste B al C, no importa como mueve las piezas (siempre que las más grandes no aplasten a las más chicas) ya que la recursividad es problema de la computadora. Cada vez que llamamos a la función recursiva la computadora tiene un problema menor porque cada vez tengo menos fichas en el poste A.





\_ La solución por pasos seria:

1. Si  $n == 1$  (caso base), mover la única pieza del poste A al poste C.
2. Mover la pieza superior de A a B,  $n-1$  veces, usando el poste C como auxiliar.
3. Mover la pieza restante del poste A al poste C.
4. Mover las piezas  $n-1$  del poste B al poste C, usando A como auxiliar.

\_ Si queremos pasarlo a código debemos ponernos de acuerdo con el usuario en el nombre de las cosas, llamaremos a los postes A, B, C, y a las piezas mediante un número desde 1 hasta  $n$ , consideramos 1 a la pieza de menor tamaño y  $n$  la de mayor tamaño. El usuario generará una entrada que será  $n$ , cantidad de piezas. Nuestra salida indicará: "MOVER LA PIEZA  $n$  DESDE EL POSTE  $x$  AL POSTE", entonces seria:

```
Torres( n, desde, hacia, auxiliar )
Si n == 1
    "MOVER PIEZA n DESDE POSTE desde HACIA POSTE hacia"
    FIN
Torres( n - 1, desde, auxiliar, hacia )
"MOVER PIEZA n DESDE POSTE desde HACIA POSTE hacia"
Torres( n - 1, auxiliar, hacia, desde )
```

La salida para  $n = 3$

```
MOVER PIEZA 1 DESDE POSTE A HACIA POSTE C
MOVER PIEZA 2 DESDE POSTE A HACIA POSTE B
MOVER PIEZA 1 DESDE POSTE C HACIA POSTE B
MOVER PIEZA 3 DESDE POSTE A HACIA POSTE C
MOVER PIEZA 1 DESDE POSTE B HACIA POSTE A
MOVER PIEZA 2 DESDE POSTE B HACIA POSTE C
MOVER PIEZA 1 DESDE POSTE A HACIA POSTE C
```

\_ La recursividad es un método poderoso usado en inteligencia artificial, su poder es que algunos conceptos complejos pueden expresarse en una forma simple. Una definición recursiva difiere de una definición circular en que tiene una forma de escapar de su expansión infinita. Este escape se encuentra en la definición o porción no recursiva o terminal de la definición. Las fórmulas recursivas pueden aplicarse a situaciones tales como prueba de teoremas, solución de problemas combinatorios, algunos acertijos, etc.

## Conclusión

\_ La recursividad son procesos que se llaman a sí mismos y esa auto llamada nos da la ventaja de resolver de manera simple y elegante un problema complejo, y la computadora es la que resuelve la secuencia más compleja gracias a sus componentes. El cerebro funciona recursivamente, por ejemplo, como salir de un lugar nos preguntamos todo el tiempo si chocamos o no, también por ejemplo la solución de caminar es una solución simple, no es el problema de adonde tengo que llegar sino que es como voy a dar el próximo paso, o sea doy un paso y me pregunto si llegue al objetivo, en caso de que no llegue debo dar el próximo paso y volver a preguntar.

\_ Es importante saber que el cálculo recursivo es mucho más caro que el iterativo.

\_ Si nosotros logramos encontrar la secuencia (instrucción) de pequeños pasos de un problema grande que lo van a resolver, el problema pasa a ser de la computadora que para eso tiene mucha memoria, procesador y velocidad

\_ Las soluciones de los juegos recursivos son elegantemente simples y por lo general muy fáciles de resolver. La complejidad la va a tener la computadora, pero no el software que nosotros desarrollamos, esta es la ventaja que nos da la recursividad.

\_ Siempre los problemas de recursividad se resuelven con 2 acciones, una acción base (el final del juego) y una acción recursiva (tenemos que pensar en un problema muy pequeño).

## **Principios de programación**

\_ Documentar las funciones importantes, pensar antes de empezar, colocar buenos nombres, las funciones tienen que hacer una sola cosa y bien, elegir la parte o función principal del software y dedicar todo el tiempo a esta y no perder el tiempo en funciones que no son tan importantes.

### 1er principio

Nombres: importancia del nombre de variable y del nombre de los subprogramas. Esto es muy importante, como principio de programación, que a las funciones que llamemos y las variables importantes del programa se les ponga un buen nombre, ya que esto ayuda a la lectura y comprensión del código que vamos a tener. Siempre se impone a las variables y subprogramas o funciones con mucho cuidado y se explican de manera detallada. No pensamos en el que está construyendo sino en el que está leyendo (profesor, compañero o yo mismo en 3 años).

1)\_ Se deben elegir nombres significativos y que sugieran claramente la finalidad del subprograma.

- 2)\_ Las variables que se usan poco deben ser nombres sencillos, una sola letra puede ser una elección idónea.
- 3)\_ Servirse de prefijos o sufijos comunes para asociar nombres pertenecientes a la misma categoría general. Por ejemplo, los archivos utilizados, podrían llamarse: ArchivoEntrada, ArchivoTransacción, ArchivoTotal, ArchivoSalida, ArchivoRechazo.
- 4)\_ Evitar errores de ortografía y los sufijos carentes de significado cuya única finalidad es acuñar nombres diferentes, es decir, evitar nombre incorrectos o con fallas en las variables porque nos olvidamos su significado. Ejemplos: índice, índice, indc, indicee, indice2, indice3.
- 5)\_ Evitar nombres rebuscados o nombres elegantes cuyo significado tiene poca o nula relación con el problema. Ejemplos: while, TV, in, empeño, do, estudiar, if, not, tengosueño, then, jugar, else, siesta.
- 6)\_ También no escoger nombres que tengan una grafía semejante o que por otros motivos se confundirían fácilmente.
- 7)\_ Para las variables simples evitar las variables I, O, porque prestan a confusión, pero si utilizar nombres simples. No todas las variables tienen que tener nombre complejo porque no tiene sentido. Ejemplo: `I := 1; x := 1; x := I; x := O.`

## 2do principio

Documentación y formato: es clave la documentación tanto para programas pequeños como para grandes. Necesitamos escribir un documento que diga lo que pretendemos hacer antes de empezar a hacer el código, y ayuda a quien tiene que leer el programa por más de que este no funcione.

\_ Uno lee el enunciado de lo que solicita y rápidamente empezamos a escribir el código, y lo que necesitamos es una pausa de pensar que es lo que estamos por hacer, y en esa pausa uno puede escribir un documento donde decimos que es lo que tenemos pensado hacer en código. Esto ayuda a quien va a leer el programa, ya que además si el programa no anda, pero alguien lee lo que se esperaba de la función, se tiene en cuenta.

\_ A veces no logramos que el código haga lo que pretendía hacer, pero si explicamos lo que pretendíamos probablemente la solución este bien, y eso ayuda a comprender a otra persona que es lo que se está por hacer. Por eso es muy interesante documentar en pequeños párrafos que es lo que uno piensa hacer antes de empezar, y es importante hacerse el hábito de documentar el programa, aunque sientan que están perdiendo el tiempo, pero en realidad estamos ganando.

\_ El tiempo de lectura de los programas es mucho más largo que el de escritura. Haga que la lectura sea fácil.

- 1)\_ Se pone un prólogo al inicio de cada subprograma.
- 2)\_ Cuando cada variable se declare se le pone una explicación, o mejor, el nombre de la variable es evidente.
- 3)\_ Se introduce cada sección importante del programa con un comentario breve que explica su finalidad.
- 4)\_ Se indica el final de esta sección, en caso de no ser obvio.
- 5)\_ No se hacen comentarios que repitan lo que hace el programa. Ej. `cont := cont + 1;`  
{ aumentar el contador en 1 }.
- 6)\_ Mantener la documentación actualizada al modificar un programa.
- 3)\_ No documentar cada línea de código en casos simples porque no tiene sentido.

### 3er principio

Refinamiento y modularidad: no son las computadoras sino las personas quienes resuelven los problemas. La clave para resolver el problema central es dividir este problema en varios problemas más pequeños. Es importante no dejar que los árboles impidan ver el bosque.

\_ Cada subprograma o función que llamamos tiene que hacer una sola tarea, y además esa tarea debe ser interesante, deberá ocultar algo, es decir, eso interesante que hicieron. A una función le decimos que tiene que hacer tal cosa, y cuando la llamamos, nos tiene que decir si la pudo hacer o no, lo que haga la función en el medio no nos importa. Esto se denomina refinamiento descendente, y es la clave para que programas extensos funcionen. Cada subprograma deberá ejecutar sólo una tarea para hacerlo bien.

\_ Cada subprograma debe poder ser descrito de manera sencilla y además deberá ocultar algo. Los subprogramas tienen sus propias tareas a realizar y sólo deben informar lo que sea necesario a los programas que lo llaman.

### 4to principio (principios de prueba)

Codificación, prueba y refinamiento ulterior: si bien hay muchos conceptos juntos, separándolos decimos que la codificación es el proceso de escribir un algoritmo en la sintaxis correcta al lenguaje seleccionado. La prueba es el proceso consistente en correr el programa en los datos muestra, seleccionados para encontrar los errores, si es que los hay. Y para realizar un refinamiento ulterior recurrimos a los subprogramas todavía no escritos y repetimos estos pasos. Para refinar se puede usar el método de los CABOS.

\_ Tenemos que dividir el problema grande en pequeños sub problemas e ir resolviendo cada uno de esos problemas por separado, donde la prueba y el testeo es para cada

subproblema. De esta manera nos facilita el hecho de encontrar errores y que el programa funciona bien.

\_ La calidad de los datos de prueba es más importante que la cantidad. Tenemos tres formas de probar nuestro programa:

1)\_ Caja negra: el usuario trata al programa como una caja negra, le importa entregar valores de entrada y ver su resultado, es decir, ponemos valores de entrada y vemos el resultado de salida para ver si una función clave funciona o no. Los criterios mínimos que nos guiarán al escoger los datos de prueba son: valores fáciles, valores típicos realistas, valores extremos, valores ilegales.

2)\_ Caja de cristal: el usuario desea probar todas las partes del sistema funcionando, para ello deberá generar todos los valores de entrada posible para que todas las partes del programa se prueben, es decir, es un método donde probamos todas las opciones, o sea probamos con todos los valores de entrada. En el caso de cuenta vecino, se deberán probar valores de conteo que den 0, 1, 2, 3, 4, 5, 6, 7, 8.

3)\_ Caja de pandora: cuando no hago ninguna prueba y dejo que lo pruebe el cliente. Este depende muchas veces de los tiempos que tengamos y de las necesidades de entrega.

## Otras reglas

- Casi todos los programadores pasan 90% del tiempo realizando el 10% de las instrucciones. Encuentre ese 10% y concéntrese en mejorar allí la eficiencia.
- Procure que los algoritmos sean lo más simples posibles.
- Analice las exigencias de tiempo y espacio al seleccionar un algoritmo.
- Nunca tema volver a empezar desde el principio. Es posible que el segundo intento sea más breve y fácil.
- Cerciórese de que entiende completamente el problema. Si se debe cambiar los términos del mismo explique exactamente lo que usted ha hecho.
- Comenzar de nuevo suele ser más fácil que parchar un programa viejo.

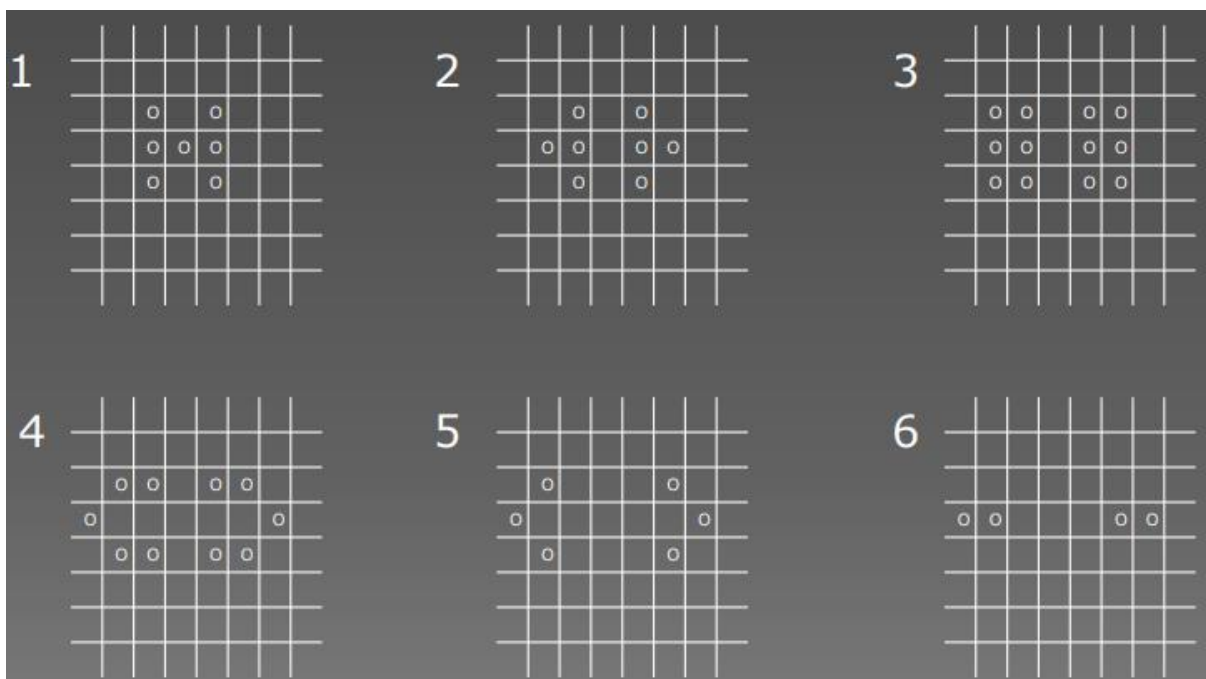
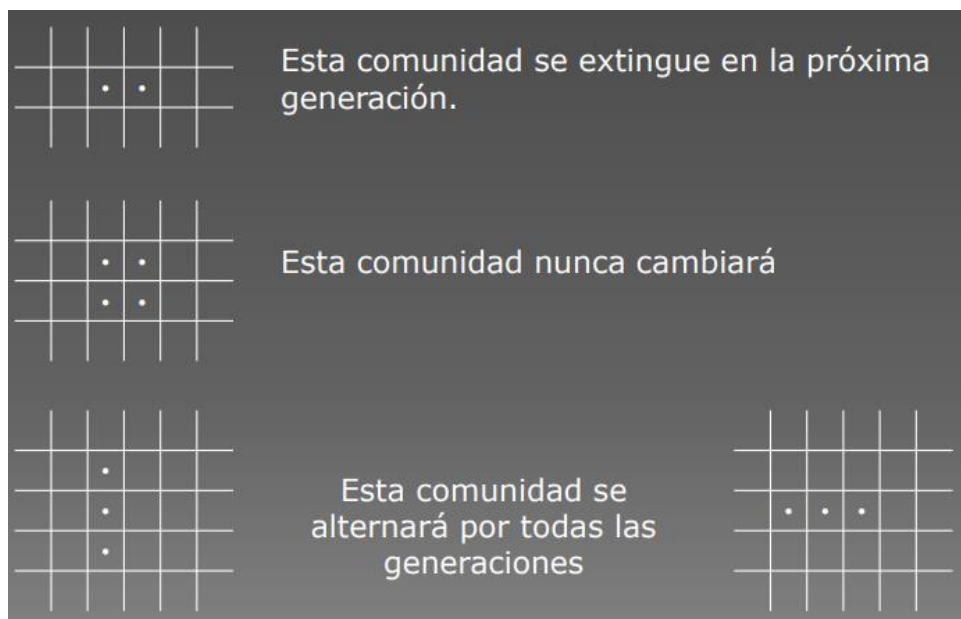
## Juego de la vida

Consigna: La estructura del juego es una rejilla rectangular abierta, en la cual cada celdilla puede estar ocupada o no por un microorganismo. Las celdillas cambian de acuerdo con las siguientes reglas:

- Los vecinos de una celda son los 8 que la tocan vertical, horizontal o diagonalmente.
- Si una celdilla está viva, pero no tiene celdillas vecinas vivas o solamente una viva, muere de soledad en la siguiente generación.



- Si una celdilla está viva y tiene cuatro o más vecinas también vivas, en la siguiente generación muere por hacinamiento.
- Una celdilla viva con dos o tres vecinas vivas permanece viva en la siguiente generación.
- Si una celdilla está muerta, en la siguiente generación recuperará la vida si tiene exactamente tres vecinas, ni una más ni una menos, que ya estén vivas. Todas las otras celdillas muertas permanecen así en la siguiente generación.
- Todos los nacimientos y muertes tienen lugar exactamente al mismo tiempo, de manera que las que mueren ayudan a producir otras; pero no pueden impedir la muerte de otras reduciendo el hacinamiento, ni las que nacen pueden preservar o destruir a las que tienen vida en la siguiente generación.



\_ Proponemos pensar un programa que juegue al juego de la vida. Para ello imaginamos una matriz de un determinado tamaño, luego pensamos en llenarla con letras o números dependiendo si la celda está viva o muerta, controlamos cada casilla para ver si tiene vecinos y de ahí saber si nace o muere alguien, para eso podríamos pensar en dos tableros, uno donde está el estado inicial y otro donde vamos escribiendo el nuevo ciclo de vida. Revisamos mediante ciclos todas las celdas para ver si tienen vecinos, repetimos el ciclo de copiar y pegar los tableros, y por último determinamos cuando va a terminar el juego (cuando todas las celdas estén muertas, cuando el ciclo de vida se repita hasta 50 veces, cuando el usuario decida).

\_ Estas soluciones son necesarias pensarlas antes de empezar a programar o escribir el programa, para ganar tiempo, entonces en la documentación que creo en un Word, yo le cuento al jefe de trabajos prácticos lo que me estoy imaginando, que en este caso pensamos en dos tableros, en uno están las celdas vivas o muertas, y en el otro voy a ir calculando, y cuando los termine los copio y vuelvo a empezar, ahora hay que ver si funciona o no pero por lo menos yo ya entendí lo que estoy por resolver.

Solución: inicializar un arreglo llamado MAPA, que contenga la configuración inicial de las celdillas vivas. Determinamos cuantas generaciones abarcará el juego. Repetimos los siguientes pasos para el número de generaciones deseado: Para cada celdilla del arreglo haga esto:

- Cuente el número de vecinos vivos de la celdilla. Si el recuento es 0, 1, 4, 5, 6, 7 o 8, ponga la celdilla correspondiente en otro arreglo llamado NUEVOMAPA para que esté muerta; si el conteo es 3, se pone la celdilla correspondiente para que este viva; y si el conteo es 2, se pone la celdilla correspondiente para que sea la misma que la del arreglo MAPA.
- Copiar el arreglo NUEVOMAPA en el arreglo MAPA.
- Imprimir el arreglo MAPA para el usuario.

Programa VIDA;

Filas = 50; Columnas = 60;

Mapa, nuevomapa = grilla( filas, columnas)

InicializoVariables;

Generacion = 0;

EscribeMapa;

DESDE generacion = 1 HASTA ultimageracion

    DESDE i = 1 HASTA filas

        DESDE j = 1 HASTA columnas

            vecinos = CuentaVecinos( i, j )

            SI vecinos IGUAL A

                0, 1 : NUEVOMAPA( i, j ) = muerta

                2 : NUEVOMAPA( i, j ) = mapa( i, j )

                3 : NUEVOMAPA( i, j ) = viva

                4, 5, 6, 7, 8 : NUEVOMAPA( i, j ) = muerta

    mapa = nuevomapa

    MuestraMapa

FIN

\_ Explicando un poco el código tenemos un programa de nombre VIDA, definimos un tamaño de 50 filas y 60 columnas. Tenemos dos arreglos que son Mapa y nuevomapa, que son matrices de 50x60. Inicializamos las variables todo en cero, el mapa está vacío. Arrancamos con la primer generación y le permitimos a la persona que dibuje en el mapa donde hay un puntito de vida. Entramos en un bucle desde la primer generación hasta por ejemplo la 50ª, desde las filas y las columnas, analizamos cada una de las celditas cuantos vecinos tienen vivos, y de ahí entramos al condicional. Una vez terminada la secuencia, copiamos el nuevomapa en Mapa, lo mostramos y repetimos esto hasta que pasen 50 generaciones y termina el juego.

Análisis función cuentaVecino(): esta es la función más crítica a programar. Si estamos en los bordes del tablero, cuando contamos los vecinos debemos tener cuidado porque nos vamos a caer afuera de la matriz, ya que no tenemos vecinos del lado en el que estemos y por eso controlamos. Luego de revisar los bordes ahora podemos contar en ese alrededor cuantos vecinos hay y verificamos si está viva o muerta la celda. Revisamos esta función porque si esta falla el programa anda mal, por eso probablemente debamos testearla con una caja de cristal para asegurarnos de que esta ande muy bien. Si tenemos una matriz de 50x60 la recorreremos unas 3000 veces y si debemos hacer 10 muestras de esta entonces son 30000 veces, por ende, es la función clave y programarla bien es fundamental para que el software funcione bien.

\_ La instrucción más compleja para una computadora es un if, en nuestra función tenemos 4 if que se hacen 30000 veces en total son 120000 if. Entonces podríamos optimizarlo agrandando la matriz y trabajando en un rango reducido para así ahorrar los if de los bordes o límites.

```
Funcion CuentaVecino( i, j )
SI i = 1 ENTONCES xInferior = 1 SINO xInferior = i - 1
SI i = Filas ENTONCES xSuperior = Filas SINO xSuperior = i + 1
SI j = 1 ENTONCES yInferior = 1 SINO yInferior = j - 1
SI j = Columnas ENTONCES ySuperior = Columnas SINO ySuperior = j + 1
Contar = 0
DESDE x = xInferior HASTA xSuperior
    DESDE y = yInferior HASTA ySuperior
        SI mapa( x, y ) = viva ENTONCES contar = contar + 1
SI mapa( i, j ) = viva ENTONCES contar = contar - 1

CuentaVecino = contar
```

\_ Analizando un poco el código, los cuatro if del principio sirven para controlar los bordes del tablero, entonces cada vez que llamamos a esta función tengo que fijarme si estoy en alguno de los bordes, porque la cantidad de vecinos en esta parte es distinta a que si estamos en el medio del tablero. Entonces para eso están los condicionales, para no rozar

con los bordes. Por otro lado, tenemos que hacer un bucle para poder contar todas las celdas que están vivas o muertas, pero no la de nosotros mismos, por eso sumamos las celdas y restamos la celda viva en la que estamos o no en caso de estar muerto.

\_ Otra forma de solución sería agrandar el mapa 52x62 pero únicamente permite tener vida en 50x60 ya que a la vuelta todo está relleno con ceros y nos despreocupamos de los bordes. De esta manera en la función `cuentaVecino()` es mucho más simple y ya no tendríamos el control de bordes y ahorrando los `if`. Automáticamente esta función va a contar si está vivo o no el vecino solamente con un `contar = contar + mapa(i, j)` si estamos muertos y `contar = contar - mapa(i, j)` si estamos vivos, en el bucle.

\_ De esta manera la función `cuentaVecino()` se solucionó enormemente, que por otro lado es la más crítica del programa. Entonces pensar esta nueva solución antes de empezar a hacer el programa nos puede simplificar las funciones más complejas. Tenemos que empezar por las funciones más críticas porque son las que se van a ejecutar más veces que todas las otras, luego pensamos su complejidad y tratamos de simplificarla para que el mismo software sea más simple. El software se debe pensar de esta manera. Y por último se realiza una prueba de testing.

\_ Debemos pensar el software y lo que va a hacer y documentarlo. Tenemos que encontrar donde está el problema central a resolver, y debemos poner toda nuestra energía en este. Mientras testeemos y hagamos más simple ese problema central, nos estamos asegurando grandes probabilidades de que nuestro software funcione. Por eso los principios de programación nos dicen:


- Documentar
- Pensar antes de empezar
- Poner buenos nombres a las funciones
- Documentar las funciones importantes
- Las funciones tienen que hacer una sola cosa y bien, no muchas.
- Elegir la parte principal del software y dedicar el 90% del tiempo a esa porque es la esencia. No dedicar tiempo a pequeñas funciones que no hacen nada o no son tan importantes.

\_ Esto ayuda a pensar los puntos clave de un programa y no los puntos simples.

# Introducción a las Estructura de datos

## ¿Qué es Información?

\_ Unidad básica de información (BIT):

1 bit  2 posiciones (0 – 1)

n bit   $2^n$  posiciones

\_ Hay alguien que definió alguna vez un byte de 8 bits por una cuestión de consenso y ponerse de acuerdo entre los desarrolladores de hardware. La composición de unos y ceros que una computadora maneja puede ser administrada, modificada o diseñada en sus comportamientos como queramos los usuarios, entonces algunas cosas se consensuaron y se definieron para poder ponernos de acuerdo todos en como utilizábamos las computadora. Esto hace que las estructuras de los datos o de información que vamos a almacenar se definen y se manejan por consenso.

\_ Una computadora está llena de 1s y 0s, y lo que hacemos es establecer consensos de que van a significar esos bits para que las cosas funcionen de manera más simple.

\_ Tenemos un kit de herramientas para utilizarlas en un momento determinado.

\_ Por ejemplo tenemos un byte de 8 bits, donde podríamos almacenar algunos unos y ceros, generando  $2^8$  combinaciones diferentes, y con esas podría hacer la combinación que quiera. En caso de:

Enteros Positivos: por ejemplo, 00100110  $\rightarrow 2^1 + 2^2 + 2^5 = 2 + 4 + 32 = 38$ , donde tenemos los unos hacemos  $2^n$  obteniendo un numero decimal. Esto se hace así porque son definiciones que tomaron quienes iban diseñando las bases de datos y que fue una buena definición porque permite sumar números binarios y que el resultado parezca la suma de los números decimales.

Números Binarios Negativos: aparecen algunas opciones para manejar los números negativos, y una de las decisiones que se toma es que el primer dígito, si es un 0 es un numero positivo, si es 1 entonces el número es negativo. Y también un consenso a definir es si lo vamos a hacer por complemento a 1 o complemento a 2:

- Notación de complemento a uno: el primer dígito representa positivo o negativo, en caso de ser negativo se invierten todos los valores de los otros bits. Con n bits se puede representar desde  $-2^{(n-1)}-1$  hasta  $2^{(n-1)}-1$ . Observamos que se puede definir el 0 de dos maneras 0+ y 0-. Quedaría 11011001 = -38.

- Notación de complemento a dos: tiene como ventaja que evita que haya un 0+ y un 0-, escribiendo el número de otra manera y no desaprovechamos una combinación de las 256 que se darían en este caso por el  $2^8$ , entonces se aprovechan mejor las combinaciones  $11011010 = -38$ , en este caso se suma un 1 a la representación del complemento a 1 del número negativo. Con n bits se puede representar desde  $-2^{(n-1)}$  hasta  $2^{(n-1)}-1$ .

\_ De todas formas estas no son las únicas maneras de representar enteros binarios.

\_ Todo esto consenso en la construcción y fabricación del hardware.

Números reales: si quisiéramos representar números con decimales, lo que hacemos es una descripción también de cómo podríamos definir esto. La más utilizada es notación de punto flotante, el número real se representa por un número llamado mantisa multiplicado por una base elevada a una potencia entera, llamada exponente. Analizando el caso 387,53 se representa como  $38753 \times 10^{-2}$ , donde lo que se plantea como opción es correr los decimales y escribir el 38753 y el  $10^{-2}$  para representar que debemos correr la coma dos lugares.

\_ Normalmente se usan 32 bits, MANTISA (24) EXPONENTE (8):

000000001001011101100001 11111110

\_ Entonces los primeros 24 dígitos binarios se utilizan para representar el 38753 y los 8 dígitos binarios finales se utilizan para representar el exponente (en este caso representamos por complemento a dos el -2).

\_ Quienes definieron las computadoras fueron tomando estas decisiones en algunos momentos y eligiendo que era lo más conveniente. Por eso cuando nosotros ahora intentamos construir por ejemplo un número real, este automáticamente ocupa 4 bytes, si necesitamos números más grandes podemos tener más bytes en ambos lados para poder representarlos. Pero esto nos permite representar números desde  $2^{23-1} \times 10^{127}$  hasta números como  $10^{-128}$ , el factor que limita la precisión es el número de dígitos significativos de la mantisa, por ejemplo, el número 10.000.001 requiere 24 dígitos y deberá representarse como  $10.000.000 (1 \times 10^7)$ .

Cadena de caracteres: también tenemos que representar de alguna manera las letras. No existen internamente dentro de una computadora una letra A, B, C, etc, siempre representamos las letras como combinación de bits de unos y ceros, para poder expresar cual sería una letra en particular. Alguien dijo que para él una letra A es tal combinación de bits, una letra B es otra combinación, y se hizo de manera tal que todas las letras se van incrementando de uno en uno, entonces una B en la combinación de bits es un número un poco más grande que una A, y eso después nos permite saber si la palabra está ordenada alfabéticamente o no, por ejemplo las ñ se ignoraron en ese alfabeto porque en inglés la ñ no estaba, entonces una ñ en combinaciones de bits técnicamente es mayor que una Z

pero después internamente el software va a hacer todo el ordenamiento alfabético para poner la ñ en el lugar que corresponde aunque técnicamente los bits la mandaron un poco más lejos que la Z.

\_ Entonces no solo la interpretación es de números sino también de caracteres. Si 8 bits representan un carácter, se pueden representar hasta 256 caracteres diferentes.

\_ Esta selección es totalmente arbitraria, en algunas computadoras al principio utilizan 7 bits, otras 8 bits, y otras 10 bits. Y en un momento, como el hardware que se estaba fabricando, algunas placas funcionaban con 7 bits, otras con 8 y otras con 10, después no se podían construir computadoras compatibles una con la otra, por eso alguien dijo en algún momento que había que ponerse de acuerdo con cuanto trabajar, y decidieron con 8 bits con consenso, pero pudo haber sido de otro tamaño.

- Byte: se define así al grupo de bits (8) que permiten representar un carácter en una computadora. Y se define una estructura con determinados tipos de datos, es decir, esta pauta puede hacerse como se desee, pero se intenta que sea de tipo consistente.

\_ Por último observemos que la cadena de bits, 00100110, puede ser el número 38 (binario) el 26 (decimal codificado) o el símbolo & (carácter). Podría cambiar depende como cada uno lo hubiera querido definir. Estos son distintos tipos de datos.

## Hardware y Software

\_ La intención es entender que los datos que administra el hardware y el software se han construido por consenso. Técnicamente adentro hay unos y ceros y esa combinación de unos y ceros alguien fue definiendo y el hardware se fue construyendo para administrarla de determinada manera, y el software se construye en función a como ese hardware administra los datos.

\_ La memoria de una computadora es simplemente un grupo de bits (0 o 1).

\_ Los bits están agrupados en unidades más grandes, los bytes.

\_ Toda computadora tiene un conjunto de tipo de datos nativos, o sea, que se construyó para manejar determinado tipo de datos. Ya se consensuó que el byte es de 8 bits, que un número positivo es de una forma y que el negativo es de otra, que una letra es tal símbolo, todo eso tiene consenso y la información se va a ir poniendo en posiciones de memoria que están almacenadas dentro de la computadora.

\_ Alojarse en una dirección de memoria la suma de las pautas de bits, alojados en otras 2 direcciones de memoria, estas son algunas de las funciones que están previstas en una computadora.

\_ Un lenguaje de programación de alto nivel, ayuda a simplificar esta tarea al programador. La idea es que no tengamos que saber qué pasa con todos esos unos y ceros internamente, ya que los lenguajes de alto nivel nos permiten ignorar todo ese tema.

## Concepto de implantación

### Tipo de dato abstracto

\_ Primero debemos separar el concepto de tipo de datos que se representa en una computadora, por el de tipo de dato abstracto. Este es un tipo de dato que realmente no existe, es abstracto justamente, pero que nosotros vamos a visualizar como un tipo de dato existente, aunque existe de manera abstracta. Estos tipos de datos podrían estar contruidos en el hardware, el hardware podría estar fabricado ya para saber manejar enteros (realizar operaciones), también para manejar reales (realizar operaciones), pero pueden aparecer otras ideas de trabajar cosas que se construyen a nivel del software, o sea se construyen de manera abstracta valiéndonos de algunas estructuras que ya existen, pero creando nuevas estructuras que no existen físicamente en el hardware sino que existen solamente a nivel de software. Si estas estructuras son muy utilizadas en algún momento se podrían implementar en el hardware, se podrían fabricar chips que hagan determinado calculo que al principio se implanta a nivel de software.

\_ Una herramienta útil para especificar las propiedades lógicas de los tipos de datos, es el tipo de dato abstracto ADT, el cual es, una colección de valores y un conjunto de operaciones con esos valores. La colección y las operaciones forman una construcción matemática que puede implantarse utilizando una estructura de datos particular, ya sea de Hardware o de Software. El término ADT se refiere al concepto matemático que define el tipo de datos. Al definir un ADT no nos importa la eficiencia en tiempo y espacio.

\_ Entonces resumiendo, en implantación de Hardware se diseña y construye el circuito necesario para ejecutar la operación requerida. En la implantación de Software se escribe un programa que consta de instrucciones existentes en el hardware para interpretar en la forma deseada las cadenas de caracteres y ejecutar las operaciones requeridas.

\_ Siempre que hablamos de implantación, hablamos de implantación conceptualmente de implantación de software. Mas allá de que quizás el hardware ya resuelve técnicamente esto que tratamos de implantar nosotros.

### Ejemplo: ADT Rational

\_ Los números racionales son números con fracciones. El hardware creemos que no administra números fraccionarios, es decir, no está fabricado el hardware para manejar números fraccionarios, pero nosotros podríamos fabricar o implantar a nivel de software un tipo de dato que vamos a llamar números Racionales. Entonces el código que vemos es



un pseudocódigo que trata de definir con palabras el componente, y luego el componente se podría definir con los lenguajes de programación que utilizamos en las computadoras.

```
abstract typedef <integer, integer> RATIONAL;
condition RATIONAL[1] <> 0;

abstract RATIONAL makerational( a, b )
int a, b;
precondition b <> 0;
postcondition      makerational[0] == a;
                   makerational[1] == b;

abstract RATIONAL add( a, b )
RATIONAL a, b;
postcondition      add[1] == a[1] * b[1];
                   add[0] == a[0] * b[1] + a[1] * b[0];

abstract RATIONAL mult( a, b )
RATIONAL a, b;
postcondition      mult[0] == a[0] * b[0];
                   mult[1] == a[1] * b[1];

abstract RATIONAL equal( a, b )
RATIONAL a, b;
postcondition      equal == (a[1] * b[0] == a[0] * b[1]);
```

\_ Entonces el tipo de dato abstracto lo defino como un numero racional que tiene dos enteros, uno que va a ser el numerador y otro que va a ser el denominador. Hay una condición que voy a definir que el racional sub 1, es decir, el denominador tiene que ser siempre distinto de cero, entonces un numero racional se construye con dos números enteros a y b, con la condición de que b no sea cero y tenemos una postcondición en donde el entero a lo guardamos en la primera posición del racional sub 0 y el entero b en la segunda posición del racional sub 1. Por ende, definimos la suma, multiplicación, y la comparación de si son iguales, de dos números racionales. También podemos proponer otras operaciones (resta, división, factor común, simplificar, etc).

\_ El tipo de dato racional es una explicación que le vamos a dar a la computadora, de con las condiciones que la computadora sabe que son los números enteros ya consensuados de cómo se escribían, ahora le enseñamos a la computadora como con dos enteros a y b podemos simular que los números que vemos se corresponden a un número racional y como son las operaciones entre estos. Por ende, implantamos un nuevo concepto valiéndonos de otro anterior.

\_ Lo que planteamos es que, conceptualmente los números racionales no existen en una computadora ni en la implantación de hardware, pero los podremos construir con una implantación de software generando acciones explicando no solamente como es el

número racional, sino como se hacen todos los cálculos que administran los números racionales. Entonces creamos un nuevo componente en función a componentes ya existentes, o sea creamos un nuevo tipo de dato abstracto valiéndonos de los tipos de datos que ya existen en la implantación de la computadora. Si la implantación que hacemos en el software es muy usada capaz que algún día se implanta en el hardware, y esto es una decisión de consenso.

### Ejemplo: Análisis ejemplo de construir una palabra

\_ Podemos construir una palabra diciendo que son 5 letras y a continuación vienen las letras, lo mismo si son 9 letras y a continuación vienen las letras, y si queremos concatenarlas deberíamos sumar 5 más 9 dando 14 para poder juntar las 14 letras y que eso pase a ser una nueva palabra. Para esto debemos crear un código que sepa unir palabras si la regla es que las palabras son de 5, 9 y 14 letras. Y decimos esto porque la construcción de las palabras de longitud variable se puede hacer de diferente manera.

Definir CONCATVAR( c1, c2, c3 ) para concatenar dos cadenas de caracteres de longitud variable.

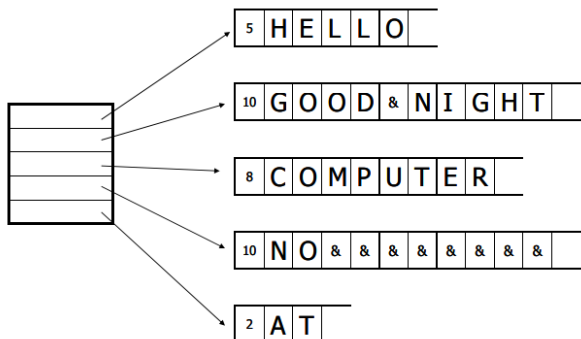
5	H	E	L	L	O
---	---	---	---	---	---

9	E	V	E	R	Y	B	O	D	Y
---	---	---	---	---	---	---	---	---	---

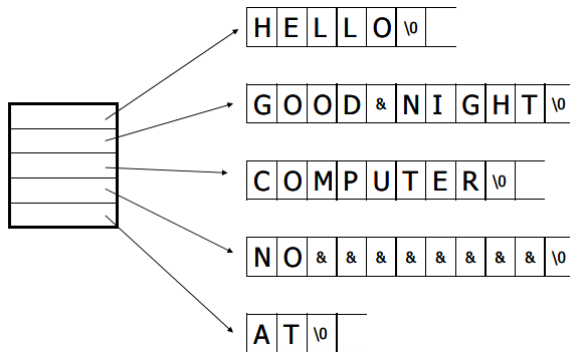
1	4	H	E	L	L	O	E	V	E	R	Y	B	O	D	Y
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

```
Sw.  z = c1 + c2;
      MOVE( z, c3, 1 )
      for (i=1; i<= c1; MOVE( c1[i], c3[i], 1) );
      for (i=1; i<= c2; {
          x = c1 + i;
          MOVE( c2[i], c3[x], 1); }
```

\_ Para construir una palabra de longitud variable, podemos tener un puntero a una posición de memoria a un determinado lugar, en donde allí escriba una determinada palabra de una cierta longitud y así con las demás:



\_ De otra manera seria construir una palabra de longitud variable y cuando encontramos el símbolo especial \0 significa que la palabra termino, entonces deberíamos fijarnos en cada uno de los casilleros si esta ese símbolo que nos da el final de la palabra y así sucesivamente.



\_ Entonces podemos tratar de construir palabras de longitud variable de diferentes maneras de la manera que queramos.

\_ Tenemos que tener como concepto la flexibilidad de poder administrar la computadora como yo quiera, es decir, es absolutamente flexible y yo construyo el tipo de dato abstracto que a mí me interese. Para construir un tipo de dato abstracto nos valemos de los tipos de datos que ya existen, que son los datos básicos (enteros, reales, caracteres), y a partir de ahí construyo un tipo de dato abstracto de la manera que yo quiera. Volviendo al caso de las fracciones, parece que estamos operando con estas, aunque internamente no lo estoy haciendo, sino que tengo un monton de herramientas que me hacen creer que lo que estoy viendo son operaciones de fracciones. Las computadoras están conformadas internamente de manera completa por unos y ceros pero que nos hacen creer todas las funciones que realizamos, porque alguien fue generando nuevas estructuras de datos sobre las estructuras de datos básicas que nos hacen sentir la sensación de hacer tal cosa cuando lo que hacemos en realidad es mandar un monton de paquetes de unos y ceros de un lado hacia el otro, pero todos sentimos lo que hacemos gracias a esta emulación que hace la computadora con sus tipos de datos abstractos para hacernos creer que esos datos son reales.

#### Ejemplo: ADT para arreglos

\_ Lo que hacemos es construir una estructura que nos dé la sensación de estar teniendo un arreglo. Por ejemplo, para construir un arreglo de 10 elementos, usando la memoria de la computadora, reservamos 10 lugares, y la computadora guarda 10 lugares en la memoria para el arreglo, con lo cual sabemos que en la primer celda está el primer elemento del arreglo y así. Pero esto es una abstracción de cómo estamos usando la memoria de la computadora.

```

abstract typedef <eltype, ub> ARRTYPE( ub, eltype);
condition type( ub ) == int;

```

```

abstract eltype extract( a, i )
ARRTYPE( ub, eltype ) a;
int i;
precondition      0 <= i < ub;
poscondition      extract == ai

```

```

abstract store( a, i, elt )
ARRTYPE( ub, eltype ) a;
int i;
eltype elt;
precondition      0 <= i < ub;
postcondition      a[i] == elt;

```

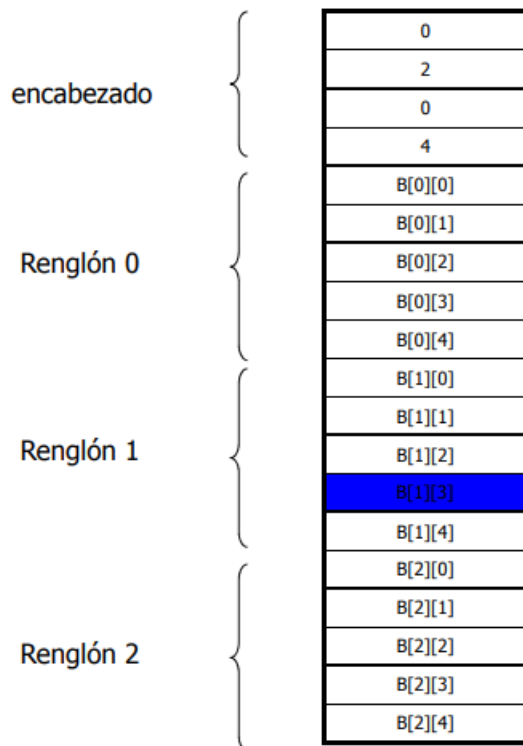
Implantación de arreglos bidimensionales alguien implanto una estructura de datos que se llama un arreglo, entonces por ejemplo nosotros le decimos a la computadora construíme un arreglo de números enteros `b[3][5]`, donde tendríamos una matriz con 3x5 números, pero físicamente no existe ninguna matriz de 3x5 sino que sigue habiendo un montón de ceros y unos, solo que vamos a creer porque alguien nos va a administrar esto que parece un arreglo, que nos hace tener la sensación de que los datos están ubicados en las posiciones de la matriz.

	Columna 0	Columna 1	Columna 2	Columna 3	Columna 4
Renglón 0					
Renglón 1					
Renglón 2					

$\text{base}(b) + (i_1 * r_2 + i_2) * x$

$\text{base}(b) + (1 * 5 + 3) * x$

\_ Entonces tenemos que construir un arreglo de 2, 3 o n dimensiones montado en una computadora que no tiene n dimensiones, sino que sentimos que existe una estructura de datos que administra múltiples dimensiones, aunque los datos estén acomodados en la memoria de la computadora uno debajo del otro, uno a continuación del otro.



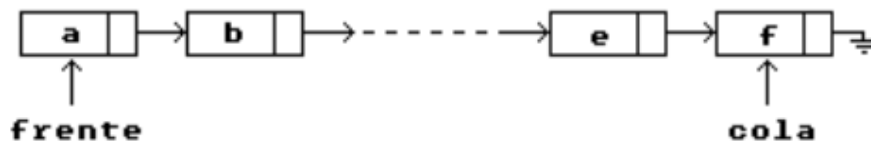
\_ A esto lo llamamos arreglo bidimensional, y así lo imaginamos, aunque el computador lo trate como un arreglo unidimensional de manera interna. El método para representar en memoria este arreglo es el denominado RENGLO-MAYOR. De esta manera el primer renglón del arreglo ocupa el primer conjunto de localidades de memoria.

Implantación de arreglos multidimensionales: se pueden implementar estos de n dimensiones, aunque técnicamente no existen las n dimensiones en las computadoras.

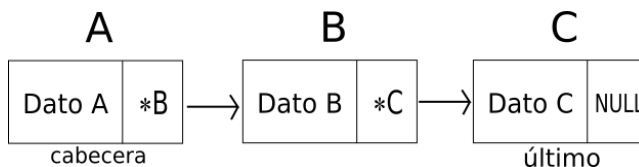
\_ Podemos crear arreglos de 3, 4, o 5 dimensiones, aunque no existen físicamente computadoras de esas dimensiones, ya que es solo imaginario. Construimos estructuras de datos abstractas y vamos a tener la sensación de que la computadora en esa estructura me hace creer que tengo un arreglo de esas dimensiones.

# Listas enlazadas

\_ Una lista enlazada es una estructura de datos abstracta en la que cada elemento (nodo) de la misma se compone de dos partes, una con información (un valor de tipo genérico, dato, información, letra, palabra, etc) y la otra con la dirección o coordenada del nodo siguiente.



\_ Vamos a fabricar una estructura de datos con la computadora valiéndonos de los datos existentes, creyéndonos lo siguiente, que existen nodos, donde cada nodo tiene dos componentes, uno es el dato en sí que podría ser un numero entero, una letra, una palabra, etc, entonces el dato estaría en la zona de tipo de dato genérico, y luego siempre hay un puntero que me direcciona a una nueva posición de memoria donde me vuelvo a encontrar con un nodo que tiene una zona que es del tipo de dato genérico y otra zona que es puntero que me lleva al próximo nodo, y así secuencialmente hasta un nodo que se llama cola (el ultimo nodo de la lista) que debería tener un puntero de tipo NULL (vacío) que nos dice que acá termino la lista, entonces podríamos direccionar mediante un puntero a una determinada posición de la memoria o del disco duro tener un dato guardado y otro puntero a la próxima dirección de memoria, así sucesivamente, y en cada zona tener paquetes de datos y nuevos punteros que me lleven a otra posición de memoria.



\_ La lista enlazada o ligada es una estructura de datos abstracta (inventada), ya que no existe físicamente en la computadora, sino que imaginariamente, pero voy a usar la memoria de la computadora como tenga ganas, vamos a guardar datos en algún lugar y de ahí voy a mandar un puntero a seguir buscando más datos en otro lugar.

Ejemplos de listas que usamos diariamente: uno podría ser un documento de Word porque cuando empezamos a escribir en este, nadie sabe cuántas hojas vamos a llegar a necesitar y que tamaño va a tener el documento mismo, entonces como no sabemos el tamaño y cuánto va a ocupar en el disco, los documentos de Word se han diseñado con un mecanismo del tipo de listas enlazadas, por ejemplo si tenemos capacidad para unas 1000 palabras, una vez que superamos el límite pone un puntero y lo dispara a una nueva posición de memoria donde tendremos capacidad para otras 1000 letras más y así

sucesivamente, por eso mi documento de Word puede crecer toda la capacidad que yo quiera, creciendo de a paquetes con punteros que me llevan a la próxima instancia de memoria, entonces los archivos de Word se construyen con una lista enlazada que va a ir creciendo según la necesidad que presente el mismo.

\_ De esta manera, usando listas enlazadas, se construyen muchas cosas en la computadora, estas al ser un tipo de dato abstracto son muy útiles en las computadoras porque se adapta a lo que requiere el usuario, es decir, las listas tienen la gran ventaja de ser muy flexibles, o sea pueden crecer lo que me haga falta y las puedo achicar lo que me haga falta de manera muy simple, cosa que no pasa con los arreglos. Tomando el ejemplo del juego de la vida, nosotros creamos un arreglo de 50x60, y estamos consumiendo 3000 posiciones de memoria fijas por más que usemos poco espacio en la matriz, si declaramos un Word fijo de 20 hojas estamos consumiendo la memoria para esa cantidad, pero si el usuario escribe dos renglones, seguimos ocupando toda la memoria reservada de las 20 hojas, y si alguien quiere escribir y usar 100 hojas no va a poder porque ya lo limitamos a 20 hojas. Los arreglos son estructuras de memoria muy rígidas, en cambio las listas enlazadas son estructuras de memoria muy flexibles.

\_ Tenemos que saber estas diferencias porque cuando desarrollamos un software deberíamos analizar si nos hace falta una estructura rígida o una flexible dependiendo de lo que el programa me pida, de lo que el cliente quiera hacer. Si tuviésemos que hacer un juego de ajedrez, ya sabemos que el tablero es de 8x8, no tiene sentido que intente representar un juego de ajedrez como una estructura flexible, porque este tablero nunca va a cambiar su tamaño, entonces nos conviene usar una estructura rígida. En el caso del juego de la vida puede ser que nos sirva un tablero de 50x60 o puede ser que no, tal vez puede ser más grande o más chico, a esto lo podríamos hacer con listas y hacerlo crecer o achicar todo lo que yo quiera, ya que en realidad la consigna de este juego decía que el tablero no tenía límites de tamaño.

\_ Perfectamente a la pantalla de una computadora la podríamos representar con un arreglo de dos dimensiones, porque la pantalla no se va a estirar y no va a cambiar su tamaño, entonces no nos haría falta usar una lista. Las fotos, es conveniente meterlas dentro de un arreglo, y yo diseño mi máquina de fotos que saque en un rango de 1000x5000, toda la memoria la administro con un arreglo, a todo el software también, y las fotos no van a cambiar nunca más de tamaño, por ejemplo. Si queremos una cámara con mejor definición, el hardware no me sirve entonces cambio la cámara.

\_ Volviendo al concepto inicial, la memoria de la computadora es una memoria donde está un byte a continuación del otro, ahora yo podría administrar la memoria para creerme que tengo por ejemplo un arreglo de 50x60 y ahora guarde 3000 lugares y adentro de esos 3000 lugares manejo todo mi arreglo con mis datos, o podría usar la memoria de la computadora de forma de una lista enlazada donde cuando necesito un

casillero nuevo pido uno y la memoria la voy usando en la medida que la necesito y el resto está libre para que la use otro.

\_ Se entiende que las estructuras de datos, los tipos de datos abstractos justamente son abstractos porque cuando yo prenda la computadora voy a usar la memoria como yo tenga ganas, o creyéndome que es un arreglo, o creyéndome que es una lista enlazada, o creyéndome que sumo fracciones, o creyéndome que uso wpp y mando mensajes de texto, cuando todo esto que planteamos no es más que una combinación de unos y ceros puestos en la memoria de una forma especial haciéndonos creer que parece todo lo que vemos. Por eso hablamos de tipos de datos abstractos, porque técnicamente los tipos de datos dependen de lo que el hardware ya tiene construido, pero a partir de ahí son implantaciones que nosotros hacemos de la forma que nosotros queramos para hacernos creer que esos tipos de datos son una cosa o son otra cosa.

\_ Vamos a construir soluciones a nuestros clientes valiéndonos de unos y ceros. Construir un tablero por ejemplo de una manera o de otra obtendrá mejores resultados. El aprovechar mejor los recursos va a producir que el resultado sea mejor en eficiencia para el cliente y también para nuestro trabajo.

### Inserción y eliminación de nodos

\_ Entonces una lista enlazada dijimos es una estructura de datos dinámica, y esto es una ventaja. El número de nodos puede cambiar dramáticamente (agregar nodos si me hacen falta o achicar nodos si es necesario, por ejemplo). Esto es a diferencia de los arreglos que son una estructura de datos rígida, por ejemplo, si hacemos un arreglo de 10 elementos solo tendremos 10 elementos y no puede crecer a menos que construyamos otro arreglo, en cambio las listas cuando nacen no ocupan nada de memoria y a medida que necesitamos más nodos vamos agregando.

\_ Para poder crear la lista necesitamos (imaginariamente) un mecanismo o función para generar varios nodos vacíos que se llama: `p = getNode()`, que nos debería entregar un nodo, en este caso `p` es un nuevo nodo vacío generado por la función. La computadora nos da un lugar del disco duro o la memoria los espacios libres para trabajar y cuando necesitamos más capacidad llamamos de vuelta a la función `getNode()` donde hay más memoria, y un puntero conecta la parte que ya hicimos con la próxima que vamos a hacer (se puede pensar con la lógica de una carta en Word).

Acciones que puedo realizar sobre una lista ligada: para que sea un tipo de dato abstracto no solo necesitamos crear la forma en la que se comporta sino también las funciones que me van a permitir manejar esa nueva estructura de datos. Necesitamos estas 5 funciones que son las que van a manejar todo lo que yo necesito:

- `p = getNode()`: me crea un nodo vacío con posiciones de memoria desocupadas, y de manera tal que no se pisen los nodos (posiciones de memoria). La letra `p` está

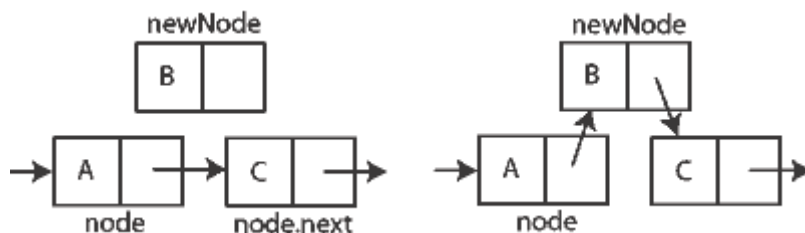


representando un puntero de memoria. Cuando le decimos a la computadora dame un nodo, y queremos escribir 200 letras en un Word, la computadora nos da un puntero a un lugar de la memoria donde a partir de ese puntero los próximos 200 casilleros son nuestros, entonces lo que me devuelve es una coordenada de memoria habilitada.

- guardarInfo(x): me permite guardar información en las zonas o sectores de los datos del nodo. En este caso por ejemplo es la función que nos permite escribir las 200 letras que tendríamos en el lugar reservado.
- guardarPuntero(p): me permite guardar información en el sector de punteros del nodo. Esta es la función, que después de las 200 letras, me permitiría guardar en ese puntero a donde van a estar las próximas 200 letras en la próxima dirección de memoria o si el documento termina acá.
- x = verInfo(): función para leer la información (me muestra la información del nodo). Esta sería la que me permite ver las 200 letras que escribí en un determinado lugar.
- p = verPuntero(): función para leer el puntero (me muestra la información del puntero). Esta me permite ver el puntero que esta al fondo de las 200 letras para ver a donde sigo.

\_ Entonces yo puedo guardar datos, guardar punteros, leer datos, leer punteros y fabricar nuevos nodos. Valiéndonos de estas funciones que nos puede proveer el hardware o el software de base, acabamos de fabricar una estructura de datos dinámica y abstracta que me va a hacer creer que la memoria de mi computadora se comporta como una lista enlazada de información a la cual le puedo agregar nodos, borrar nodos, etc, con las funciones que tenemos. La p y la x son distintos tipos de datos que voy a manejar.

Inserción en una lista: cuando escribo información en una lista podría poner información al principio de la lista, al final de la lista o en el medio de la lista. Podemos administrar la lista como nosotros queramos. Esto permite por ejemplo mantener una lista ordenada. Para esto último se debe considerar la búsqueda de un dato en la lista. Nosotros podemos ingresar los nodos donde tengamos ganas. Si agregamos un nuevo nodo a la lista, lo que sucede es que los punteros cambian su conexión de manera tal que la lista se reacomoda.



\_ Si tuviésemos un arreglo de 1000 nombres y queremos sacar un nombre (el primero, por ejemplo), tenemos que ir copiando todos los que estén abajo un lugar para arriba, en este caso todos los 999 un lugar para arriba. En una lista enlazada con solo redirigir un puntero,

eliminamos un nodo de la lista o lo cambiamos de lugar y esta se mantiene ordenada. Entonces mantener una lista enlazada ordenada, es mucho más fácil que mantener un arreglo ordenado.

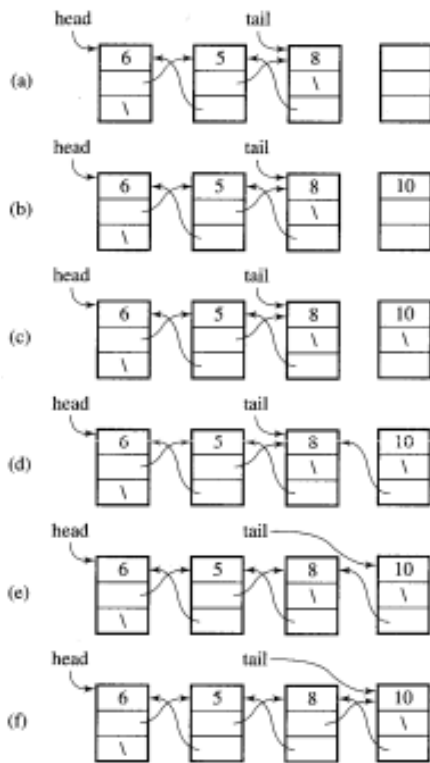
\_ Hay cosas que son más fáciles con un arreglo y hay otras cosas que son más fáciles con una lista. Si en un arreglo con 1000 personas quiero ir al medio de la lista, vamos a la posición 500 y automáticamente estoy en el medio del arreglo, en el caso de una lista si yo quiero ir al medio la única forma en que lo puedo hacer es recorriendo porque no tengo chances de entrar en el medio de la lista porque no se en que posición de memoria está el medio.

Eliminar un nodo: podemos eliminar nodos del principio del medio o del final de manera muy fácil solamente cambiando algunos punteros. Consideremos el caso de eliminar un nodo al final de la lista, esta operación ahora es muy simple, ya que el último nodo tiene acceso a su predecesor. No olvidemos que intentar borrar un elemento de una lista vacía pudiera ocasionar que nuestro programa fracasara, así que habrá que verificar siempre que la lista no esté vacía antes de borrar.

## Clasificación de listas

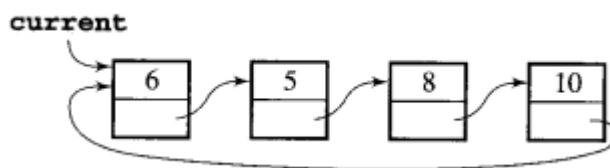
Listas simplemente enlazadas: las listas que vimos hasta ahora se llaman así, donde cada nodo contiene un único enlace que conecta al nodo siguiente o sucesor, es decir, son listas que van enganchando un puntero con el próximo y así sucesivamente (recorrer hacia adelante).

Listas doblemente enlazadas: se podrían fabricar estas estructuras de datos donde cada nodo contiene 2 punteros, uno al nodo siguiente y uno al anterior (recorrer hacia delante y hacia atrás). Estas son un poco más complejas, donde tendríamos doble punteros, es decir, tengo un nodo y dos punteros, uno para ir al siguiente y el otro para ir al anterior, entonces podemos recorrer la lista hacia la derecha o la izquierda eligiendo porque puntero me guio. Estos son mucho más difíciles de mantener porque cada vez que creo un nodo y lo inyecto a la lista tengo que poner un puntero que apunte para adelante y un puntero que apunte para atrás y si borro algún nodo tengo que corregir todos los punteros para que se sigan viendo la lista para adelante y para atrás, pero tengo la ventaja de poder ir hacia un lado o hacia el otro, ganando flexibilidad. En el caso de una lista vacía, tanto head como tail apuntan al mismo nodo.



Lista circular simplemente enlazada: similar a la primera (simplemente enlazada) sólo que el último nodo enlaza con el primero para poder seguir circulando por los otros nodos, es decir, esta lista no tiene final (recorrer hacia adelante en forma circular). El concepto es el mismo, pero en algunas situaciones, una lista circular es necesaria. En esta lista finita cada nodo tiene un sucesor. Un ejemplo de ello es cuando varios procesos están usando el mismo recurso por la misma cantidad de tiempo y tenemos que asegurarnos de que ningún proceso acceda al recurso antes de que todos los demás lo hayan hecho.

Lista circular doblemente enlazada: similar a la segunda (doblemente enlazada) con el último nodo enlazado al primero y el primero al último.



- \_ Una lista vacía que no contiene nodos se representa con el nodo “frente” en null.
- \_ Estos elementos o estructuras de datos, están en un monton de lugares de la computadora, por ejemplo, cuando apretamos alt tab en la computadora podemos ver todas las aplicaciones que tenemos abiertas en el momento, y este funciona como una lista circular por que el que la diseño pensó que estaba bueno que sea de esta forma y se valió de esta. En la galería de fotos, un álbum parecería funcionar como una lista doblemente enlazada, pasamos a la foto siguiente o la anterior.
- \_ Tenemos estructuras de datos, y también tenemos combinaciones de estas para tener sensaciones nuevas de cómo funcionan las cosas.

# Pilas

\_ Una pila es una colección ordenada de elementos en la que pueden instalarse y suprimirse elementos por un extremo llamado tope. La idea de una pila es una manera de guardar las cosas y de retirar las cosas. Esta es dinámica y por lo tanto variable.



F
E
D
C
B
A

\_ Es una estructura de datos que vamos a usar en computación que hace que guardemos las cosas arriba o sea en el tope y las saquemos del mismo tope. Pensando en una pila de libros, ponemos uno arriba del otro, y por ahí nos surge la intención de retirar uno que este al medio y físicamente podemos, pero el modelo afirma que solo se puede retirar el que está arriba o primero de todos. Entonces únicamente podemos sacar el que está arriba, si nosotros necesitamos sacar algo que está en el medio o al fondo, entonces la pila no es la estructura que necesitaríamos para eso. La pila sirve como estructura de datos cuando yo quiero poner y sacar las cosas del tope.

\_ Existen estructuras de datos para resolver distintos tipos de problemas, no tenemos que transformar una estructura en otra, sino ver cuál es la más apropiada para resolver nuestro problema. Entonces tenemos un kit de herramientas para usar la más apropiada en cada momento.

\_ Las pilas son estructuras que se encuentran frecuentemente en la vida diaria. Algunos ejemplos se encuentran en la forma en que se acomodan los platos en algunas cafeterías, la manera en que se colocan libros en un escritorio, una lata de pelotas de tenis porque no tenemos chances de sacar ninguna que no esté más arriba o bien algunas tareas que se desean realizar. La pila es uno de los conceptos más útiles dentro de la ciencia de la computación.

Ejemplo apilar libros: se puede añadir un libro a una pila poniéndolo hasta arriba de todos, se puede ver qué libro se encuentra hasta arriba y de esta misma forma si se desea retirar uno, se tomará el último que se colocó.

\_ Cada componente de la torre de hanoi funciona como una pila.

\_ Esta propiedad de poner las cosas por encima y sacar el ultimo que ingreso se conoce como LIFO (last in first out), es decir, el último en ingresar es el primero en salir y al que se

Acciones que puedo realizar con una pila: las únicas operaciones que pueden realizarse con pilas son las siguientes:

- \_ Con crearPila, pilaVacía, poner y sacar son las cuatro funciones esenciales de la pila, es decir, con estas 4 podemos hacer funcionar una pila.

[illegible]

Ejemplo paréntesis: Consideremos una operación matemática que contenga varios paréntesis anidados,  $7 - ((x * ((x + y) / (j - 3)) + y) / (4 - 2.5))$ . Para resolver esto, tratamos de contar cuantos paréntesis abiertos y cerrados hay, y si todos estos están bien o están mal. Leemos los elementos de la ecuación y cuando encuentro un paréntesis abierto lo apilo, es decir, lo pongo en la pila, y cada vez que encuentro un paréntesis cerrado sacamos un paréntesis abierto que esté en la pila, entonces si la pila queda vacía significa

que cada paréntesis que se abrió tuvo el compañero que se cerraba, en caso de que no entonces está mal la ecuación.

\_ En este caso tendríamos problemas como “(( a + b )” donde tenemos dos paréntesis abiertos y no cerramos uno, el caso de “a + b (“ donde abrimos un paréntesis y no lo cerramos, o también “) a + b ( - c” donde hay un paréntesis cerrado con uno abierto después que darían pares pero daría error porque estaríamos tratando de sacar un paréntesis pero no hay nada. La única forma de que la ecuación este bien escrita es que nunca intente sacar algo que no hay y que al finalizar la pila haya quedado vacía. La solución sería:

```
funcion verificarParentesis ( ECUACION );
crearPila();

FOR i=1 TO len(ECUACION)
  SI ECUACION[ i ] = " ( "
    poner( ECUACION[ i] );
  SI ECUACION[ i ] = " ) "
    SI pilaVacia()
      MOSTRAR( "Error en posición ", i );
    SINO
      sacar();
ENDFOR

SI pilaVacia()
  MOSTRAR( " La ecuación esta correcta ");
SINO
  MOSTRAR( " Error, la ecuación esta incorrecta ");
FIN
```

\_ Analizando el código primero pasamos por parámetro una ecuación. Creamos una pila vacía. /Con un for recorreremos la ecuación y me fijo elemento por elemento si es un ( o no, en caso de que lo sea lo agrego a la pila. Si el elemento de la ecuación es un ), chequeamos que la pila no este vacía, en caso de que si mostramos error, de lo contrario sacamos el último elemento. Si encontramos algo distinto a los paréntesis, no hacemos nada. Cuando terminamos el for, tenemos que verificar si la pila está vacía y de ser así está correcto, caso contrario tira error.

Ejemplo varios tipos de paréntesis: complicando la cosa, vamos a tratar de hacer una función que también chequea ecuaciones pero no solamente que tengan paréntesis sino que también tienen corchetes y tienen llaves. La consigna sería igual que el anterior y verificar si la ecuación está bien o está mal. Lo hacemos con una sola pila y no más complicado que la resolución pasada:

```

funcion verificarParentesis ( ECUACION );
crearPila();

FOR i=1 TO len(ECUACION)
  SI ECUACION[ i ] = " ( " o " [ " o " { ";
    poner( ECUACION[ i ] );
  SI ECUACION[ i ] = " ) " o " ] " o " } ";
    SI pilaVacía()
      MOSTRAR( "Error en posición ", i );
    SINO
      SI opuesto(valorTope(), ECUACION[ i ] );
        sacar();
      SINO
        MOSTRAR( "Error en posición ", i );
FINFOR

SI pilaVacía()
  MOSTRAR( " La ecuación esta correcta ");
SINO
  MOSTRAR( " Error, la ecuación esta incorrecta ");
FIN

```

\_ Analizando el código primero pasamos por parámetro una ecuación. Creamos una pila vacía. Con un for recorremos la ecuación y me fijo elemento por elemento si son (, [ o {, en caso de que lo sean los agrego a la pila. Si encuentro los opuestos, verifico si la pila está vacía, de ser así mostramos error, caso contrario nos fijamos el valor tope. Si el tope es justo el opuesto lo sacamos, si es un símbolo diferente tira error ya que la ecuación está mal escrita. Al salir del for, si la pila está vacía entonces la ecuación está bien, de lo contrario tira error.

## Pilas como tipo de dato abstracto

\_ Es otra forma de escribir. Siempre que yo defino un tipo de dato abstracto, en la definición decimos que el objeto se llama pila y las pilas funcionan de esta manera, y lo que describe son las tres funciones principales para poder explicar el funcionamiento, que son verificar si están vacías o no, necesito poder sacar cosas y poner cosas. Lo que tratamos de decir es como va a funcionar este tipo de dato, es la descripción de cómo funciona más allá de las palabras que se utilicen.

```

abstract typedef STACK( eltype);

abstract empty( s )
STACK( eltype) s;
postcondition empty == ( len( s ) == 0 )

abstract eltype pop( s )
STACK( eltype ) s;
precondition  empty( s ) == FALSE
postcondition pop == first( s' );
               s == sub( s', 1, len( s' ) - 1 );

abstract push( s, elt )
STACK( eltype ) s;
eltype elt;
postcondition s == <elt> + s';

```

\_ En el código anterior verificamos si la longitud de la pila es cero, entonces la pila está vacía, sino tenemos dos funciones, una es poner push (agrega un elemento a la pila) y la otra es pop (saca un elemento de la pila).

Ejercicios notaciones: existen tres tipos de notación en matemática, estas son la notación infija, prefija y posfija. La notación infija es como escribimos habitualmente y necesitamos de paréntesis para darle más prioridad a la suma o resta que la multiplicación. Pero la notación prefija y la posfija tienen la ventaja de no requerir los paréntesis. Las pilas son una buena herramienta en computación para hacer cálculos matemáticos en estas notaciones y también sirven para convertir cosas que están en infija a posfija y así.

Infija	A+B	A+B*C	(A+B)*C	(A+B) * (C-D)
Prefija	+AB	+A*BC	*+ABC	*+AB-CD
Postfija	AB+	ABC*+	AB+C*	AB+CD-*

\_ Las pilas sirven en muchos casos para resolver ecuaciones matemáticas, no solamente para controlar los paréntesis, sino que también para resolver las mismas.

\_ A continuación vemos una solución para notación prefija, donde la función se vale de poner y sacar elementos para poder obtener el resultado de la ecuación \*+AB-CD:

```
CreaPila()
Repetir mientras len( cadena ) > 0
    symb = primerelemento( cadena )
    cadena = sub( cadena, 2, len( cadena ) - 1 )
    Si symb ==+ o symb ==- o symb ==* o symb ==/
        Poner( symb )
    Si !(symb ==+ o symb ==- o symb ==* o symb ==/)
        tope = ValorTope()
        Si tope ==+ o tope ==- o tope ==* o tope ==/
            Poner( symb )
        Sino
            valor1 = Sacar()
            operación = Sacar()
            resultado = Evaluar( valor1, operación, symb)
            cadena = resultado + cadena
Fin Repetir
Imprimir( resultado )
```

\_ Analizando un poco el código tenemos que la función se repite mientras que la cadena sea mayor a cero. Y analiza elemento por elemento. Al encontrar un \* se fija que tipo de símbolo es y lo pone en la pila, al encontrar un + lo vuelve a poner en la pila. Cuando el símbolo no es un operador, se fija en el valor tope, en donde si este es un operador volvemos a poner la letra A. repetimos el bucle y cuando encuentra la letra B que no es un operador, lo que sucede es que sacamos A y sacamos el operador + y calculamos el



resultado de  $A+B$ , en donde a este último lo ponemos en la pila pero como  $A+B$ . Continuamos con la cadena, donde sucede lo mismo con  $C-D$ , terminando con la operación  $A+B * C-D$  que se imprime como resultado.

Ejemplos de pila en la computadora: cuando nos metemos en una página de internet y queremos volver atrás el sistema tiene una pila, que para volver debemos salir de la última en la que estamos, en Spotify también. El ctrl Z del Word, con el deshacer y el rehacer también. También las funcionalidades de la PC porque va desde el inicio de Windows, prendo el monitor, prendo los parlantes y abro el Word, y para apagar la PC sacaríamos cosas de la pila por ende primero se cierra el word, luego se apagan las cosas de uno en uno hasta que la pila está vacía.

## Conclusión

\_ Resumiendo entonces decimos que la pila es uno de los conceptos más útiles en la ciencia de la computación. Como sucede en las listas, esta es una estructura abstracta y dinámica en la que crece lo que nos haga falta y esta codificada de tal forma que nos haga creer la computadora de que estamos usando una pila, y el uso de esta dependerá de las necesidades del usuario. Cada software que se va montando en la memoria de la computadora necesito tener un poner para saber dónde estaba y un sacar para volver a ese programa. Cada cosa que la computadora va haciendo, las múltiples tareas para poder pasar de una tarea a la otra es esencial usar una pila para ir poniendo donde estaba y poder volver a la acción anterior. Las pilas son una estructura de datos clave en la computadora, donde cada uno de los programas se van apilando, y al momento de cerrar se tienen que ir sacando.

\_ Si hiciéramos un programa recursivo, donde un programa llama al otro pero nos olvidamos la función de base y se sigue repitiendo (va poniendo cosas en la pila), hasta que en algún momento explota y da un error stack overflow.

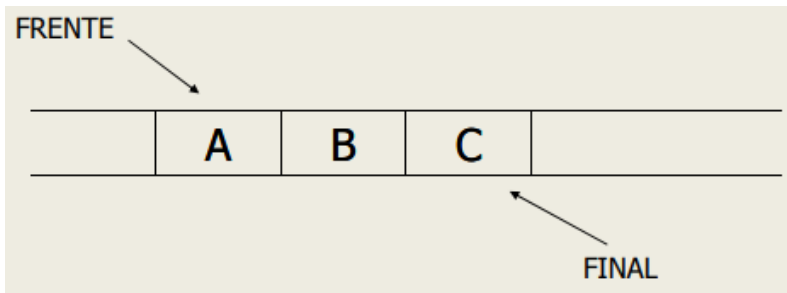
Stack overflow: este significa que la pila se llenó, consumimos toda la memoria en la pila porque apilamos tantas cosas hasta que la memoria se llenó. La memoria se llenó por que la pila se agotó/llenó.

\_ Las computadoras cuando arrancan, una de las primeras cosas que se hacen es valerse de una especie de pilas abstractas (imaginarias), donde apilan todos los programas que vamos abriendo, y si esta no existiera la pila no habría forma de volver de nuevo a los programas anteriores.

\_ La pila es el elemento esencial de funcionamiento de una computadora o teléfono desde el momento que arranca, es decir, es la estructura de datos primaria.

# Colas

\_ Una cola es una colección ordenada de elementos, de la que se pueden borrar elementos en un extremo (FRENTE de la cola) o insertarlos en el otro (FINAL de la cola), es decir, los datos van saliendo a medida que van llegando. Este es dinámico, por lo tanto puede ir creciendo de acuerdo a nuestras necesidades.



\_ Las colas son estructuras que se encuentran frecuentemente en la vida diaria. Algunos ejemplos se encuentran en la fila de un banco, un grupo de automóviles esperando en una cabina de peaje, personas esperando para comprar un boleto para el cine, peaje, etc. También podemos tener la impresora donde tenemos una cola de hojas de impresión y la primera que se escanea es la primera que se imprime.

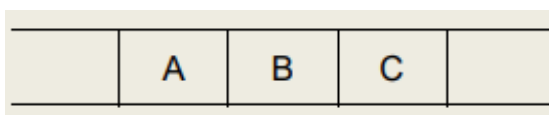
Ejemplo peaje: la forma en que se acomodan los coches que llegan a una cabina de peaje es la siguiente: se puede añadir un coche al final de la cola, y el primero que se va es el que está al principio de la cola. O sea que el primero que llega es el primero que está al principio de la cola. O sea que el primero que llega es el primero que se va.

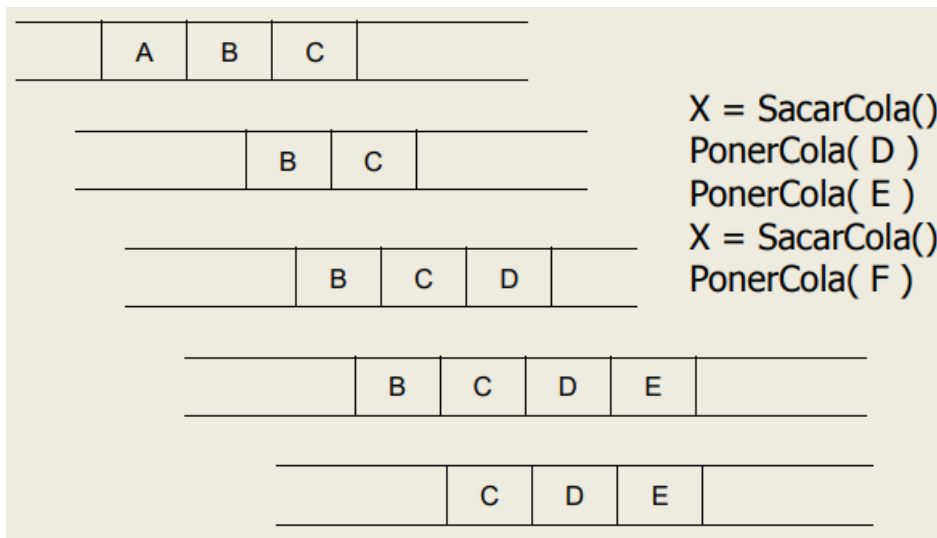
\_ La propiedad de las colas se conoce como FIFO (First In First Out), es decir, el primero en entrar será el primero en salir.

Acciones que puedo realizar con una cola: las únicas operaciones que pueden realizarse con colas son las siguientes:

- crearCola(): esta función me permite crear una cola vacía.
- ColaVacía(): esta función pregunta si la cola está vacía o está llena, y responde si hay elementos o no adentro de la misma. Esta es importante porque si intento sacar cosas y no hay nada el sistema debería dar error.
- Poner(x): función que me permite ingresar elementos adentro de la cola.
- x = sacar(): función que saca un elemento de la cola y lo almacena en x y devuelve para que sepamos cual es el primero que sale.

\_ Veamos lo que sucede con nuestra cola si seguimos la siguiente secuencia de acciones:





\_ Cuando sacamos A de la cola (inicio), el puntero de inicio apunta ahora a B y así cada vez que quitemos elementos, y los que se agreguen van al final de la cola.

## Colas como tipo de dato abstracto

\_ Es otra forma de escribir. Esto es algo representativo de lo que el modelo está haciendo, y es que está tratando de modelar de manera genérica como se debería comportar el software para que la estructura de datos sea una pila o una cola, pero siempre a nivel de modelado. Lo que estamos tratando de modelar se parece a un string (ABCDE...) donde con el insert agregamos caracteres y con el remove lo que haría es sacar un elemento del string, y en este caso como es la cola sale el primero.

```

abstract typedef QUEUE( eltype);

abstract empty( s )
QUEUE( eltype) s;
postcondition  empty == ( len( s ) == 0 )

abstract eltype remove( s )
QUEUE( eltype ) s;
precondition  empty( s ) == FALSE
postcondition  remove == first( s' );
               s == sub( s', 1, len( s' ) - 1 );

abstract insert( s, elt )
QUEUE( eltype ) s;
eltype elt;
postcondition s == s' + <elt>;

```

\_ Entre un pila y una cola como tipos de datos abstractos, la diferencia es que la pila pone los datos por delante y la cola por detrás, entonces la función sub hace lo mismo.

## Cola de prioridad

\_ Es una estructura de datos en la que el ordenamiento de los elementos se determina por el resultado de operaciones básicas. Podemos realizar dos operaciones, una es añadir con prioridad donde se añade un elemento a la cola, con su correspondiente prioridad, y la otra es eliminar un elemento de mayor prioridad donde se devuelve y elimina el elemento con mayor prioridad más antiguo que no haya sido desencolado de la cola.

\_ La cola de prioridad puede ser de tipo ascendente o descendente. Las colas de prioridades con ordenamiento ascendente, en ellas los elementos se insertan de forma arbitraria, pero a la hora de extraerlos, se extrae el elemento de menor prioridad. En las colas de prioridades con ordenamiento descendente, son iguales que la colas de prioridad con ordenamiento ascendente, pero al extraer el elemento se extrae el de mayor prioridad.

\_ Un ejemplos de la vida diaria serían la sala de urgencias de un hospital, ya que los enfermos se van atendiendo en función de la gravedad de su enfermedad.

\_ El mecanismo interno para priorizar las actividades seria pensar en una cola para cada tipo de prioridad. Como por ejemplo un banco u hospital donde sacamos turno para ser atendidos y a su vez hay distintos tipos de tramites que son por turno y orden de llegada, en donde van saliendo los turnos uno después de otro según las prioridades de cada uno.

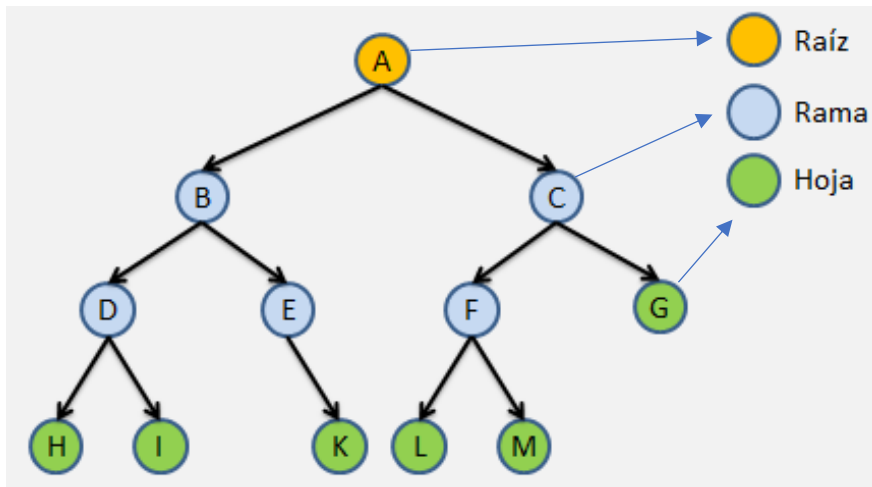
## Arboles

\_ Es una estructura de tipo de dato abstracto muy útil en muchas aplicaciones. Existen varias formas de árboles. Nos interesa el tratamiento por su forma de implantación, no lo analizaremos como tipo de dato abstracto (TDA), sino por su estructura de dato. Es una estructura de datos no lineal, es decir, en las listas, pilas y colas por lo general pasábamos de un nodo a otro de manera lineal pudiendo recorrerlo en un sentido o ambos trabajando en una secuencia lineal, en el caso de árboles no tiene una secuencia lineal sino que va a admitir distintos caminos.

Definición formal: a un árbol lo podemos definir con dos conceptos:

- Caso base: es un árbol con un solo nodo (es a la vez raíz del árbol y hoja).
- O sino cada nodo podría a su vez tener  $k$  arboles raíces donde cada uno de estos próximos nodos  $n_1$  hasta  $n_k$  se constituye en sí mismo en un nodo que podría ser un elemento o que podría tener  $k$  subárboles en las raíces, o sea denominarse cada uno en sí mismo como subárboles. Explicado de otra forma un nuevo árbol a partir de un nodo  $n_r$  (nodo raíz) y  $k$  árboles de raíces  $n_1, n_2, n_3 \dots n_k$  con  $N_1, N_2, N_3 \dots N_k$

elementos cada uno, puede construirse estableciendo una relación padre hijo entre  $n_r$  y cada una de las raíces de los  $k$  árboles. El árbol resultante de  $N = N_1 + N_2 + \dots + N_k$  nodos tiene como raíz el nodo  $n_r$ , los nodos  $n_1, n_2, n_3, \dots, n_k$  son los hijos de  $n_r$  y el conjunto de nodos hoja está formado por la unión de los  $k$  conjuntos hojas iniciales. A cada uno de los árboles  $A_i$  se les denota ahora subárboles de la raíz.



\_ Un árbol tiene la sensación de ser una definición recursiva. La definición de la estructura en sí ya da la sensación de una definición recursiva. Es decir hay un nodo, y ese nodo o es solamente el nodo o ese nodo a la vez tiene hijos, donde cada uno de los hijos lo defino como un subárbol con lo cual vuelve a ser o un nodo o tener  $n$  hijos.

\_ Al tener una definición de tipo recursiva vamos a necesitar ahora en cualquier proceso o acción que hagamos dentro de una estructura tipo árbol, vamos a encontrarnos con soluciones todas recursivas. El concepto de recursividad aparece netamente en este modelo de árboles. Cada nodo puede ser un caso base en sí mismo o auto llama a nuevos subárboles donde cada nuevo nodo es un caso base o llama a otro subárbol y así se repite la recursividad.

\_ Los árboles son por definición  $n$ -arios,  $n$  cantidad de hijos,  $n$  cantidad de ramas, o sea cada nodo podría tener múltiples hijos, y cada subnodo hijo a su vez se puede dividir en un nuevo subárbol  $n$ -ario con  $n$  cantidad de hijos. Para simplificarnos nuestro proceso o problema dentro de este curso vamos a tomar el concepto de árboles binarios, pero el modelo real es de tipo  $n$ -ario.

\_ Las carpetas del escritorio del sistema operativo tienen una estructura de árbol, en donde antes se llamaba árbol de escritorio.

## Arboles binarios

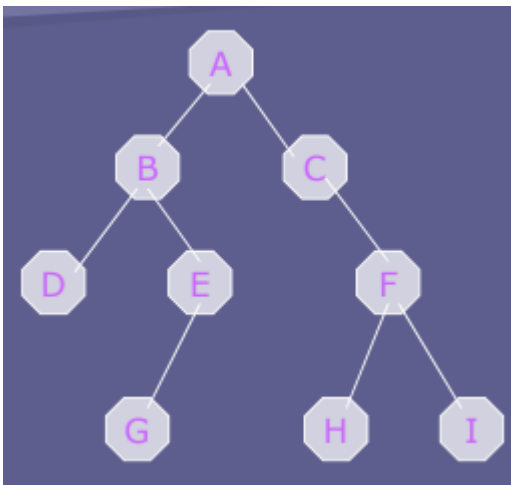
\_ Es un árbol que tiene la particularidad de que cada nodo puede estar vacío o puede estar dividido en tres elementos, que serían el nodo raíz, o sea el nodo en sí, y solamente dos subárboles, uno árbol izquierdo y un árbol derecho, cada uno de estos subárboles podrían estar vacíos o pueden ser a su vez nuevos árboles.

\_ Desde otro punto de vista, un árbol binario es un conjunto finito de elementos que o está vacío o está dividido en tres subconjuntos. El primer subconjunto contiene un solo elemento llamado raíz del árbol. Los otros dos son en sí mismos árboles binarios, y se llaman subárbol izquierdo y derecho. Cualquiera de estos puede estar vacío. Cada elemento del árbol binario se llama nodo del árbol.

Representación gráfica: donde A es el nodo raíz de este árbol y B representa el subárbol izquierdo, C representa el subárbol derecho, donde cada uno de estos nodos B o C a la vez pueden tener subárbol izquierdo o derecho y cada uno de los nodos podrían tener también subárbol izquierdo o derecho, que podría ser que estuviera vacío, o que fuera un nodo hoja o que fuera un nodo que vuelve a tener nuevos subárboles.

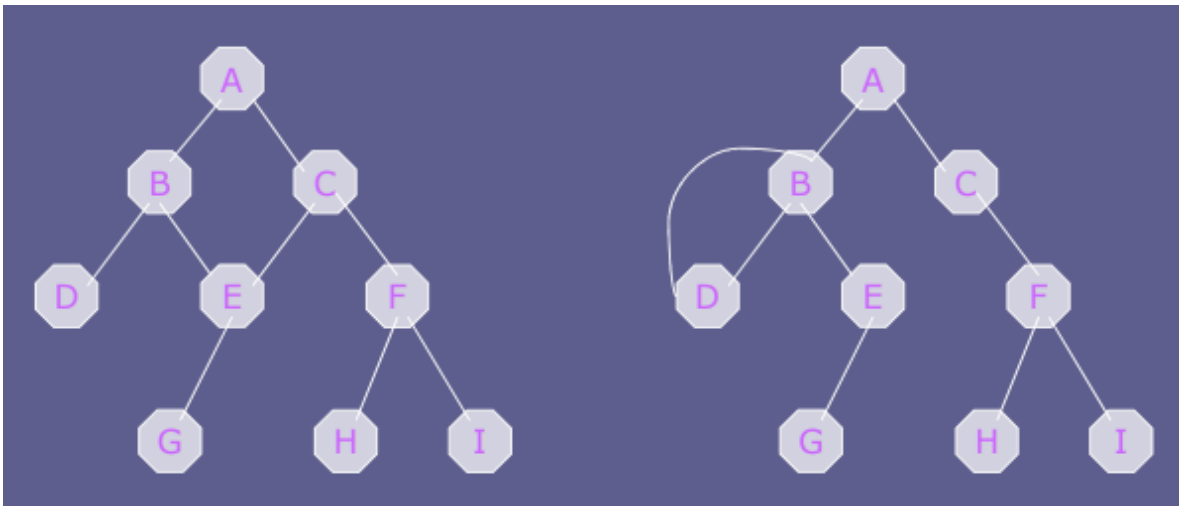
\_ En este caso como A es la raíz del árbol y B representa al subárbol izquierdo, entonces A es padre de B y también de C ya que es su subárbol derecho. No se admiten más de un nodo padre, si se admiten múltiples nodos hijos, en este caso como es binario dos hijos nada más.

\_ También podemos realizar alguna asociación de tipo familiar y decir que B y C son hermanos. Y también que B es un ancestro de G, pero no lo es de H.

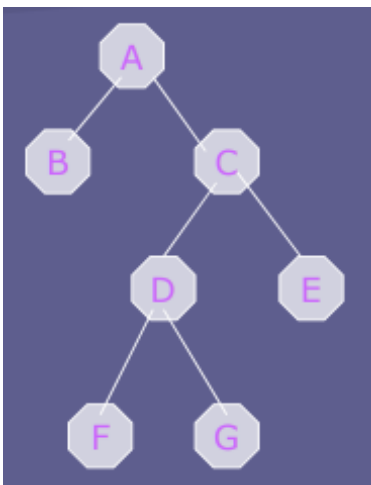


Nodos hoja: son aquellos nodos que no tienen hijos, en este caso son D, G, H y I.

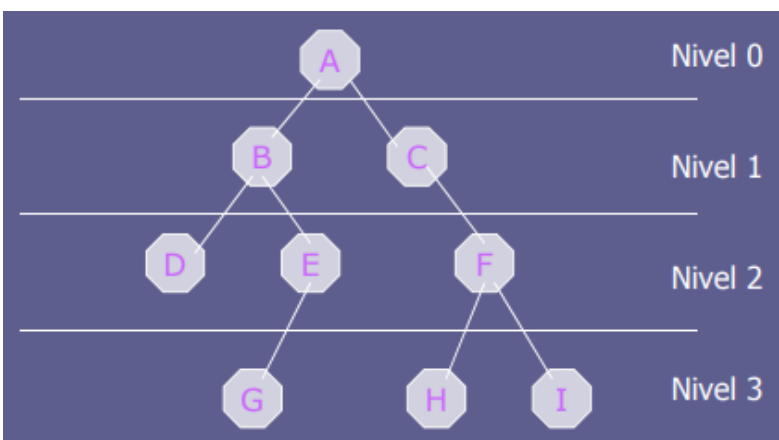
Casos que no representan arboles binarios: ninguna de los gráficos a continuación representa un árbol binario porque no se cumplen las reglas de que tengo un solo nodo padre, y no puedo tener un nodo que es padre de su propio padre.



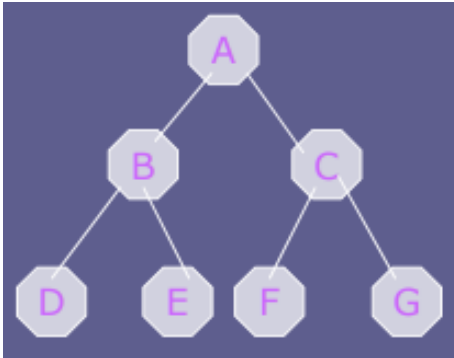
Árbol estrictamente binario: decimos que un árbol se denomina así si tiene o ambos dos hijos o ningún hijo, es decir, si un nodo que no es hoja de un árbol binario, tiene subárboles izquierdo y derecho no-vacíos. Una particularidad del árbol estrictamente binario con  $n$  hojas, siempre tiene  $2n - 1$  nodos, por ejemplo si tienen 4 hojas entonces tienen 7 nodos, es decir, sabiendo cuantas hojas tiene podríamos saber cuántos nodos tiene el árbol.



Profundidad de los árboles: el nivel de un nodo de un árbol se define de la siguiente manera, la raíz del árbol tiene nivel 0, luego cada nodo del árbol es un nivel más que su padre. Es decir si tienen un solo nodo la profundidad es cero si tiene hijos la profundidad es 1, si esos hijos a su vez tienen hijos la profundidad es 2 y así sucesivamente.



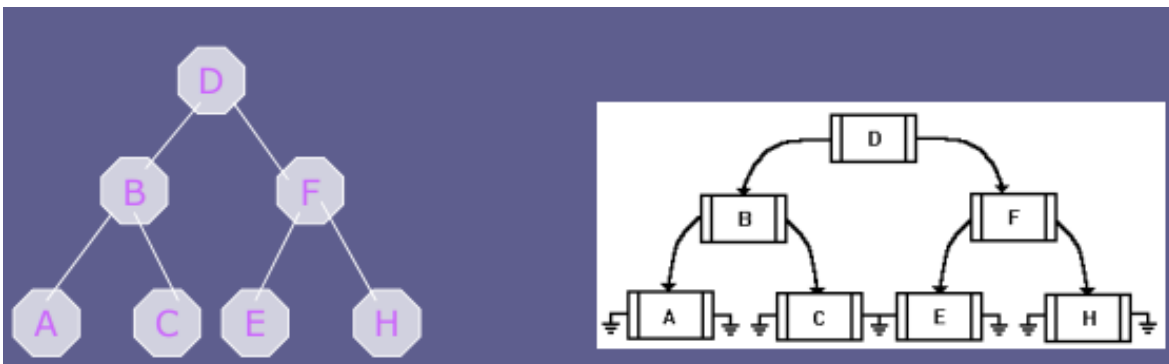
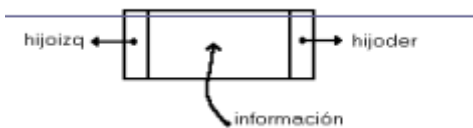
Árbol binario completo: es un árbol estrictamente binario de profundidad D, cuyas hojas tienen la misma profundidad D. Un árbol estrictamente binario pero con la misma profundidad, llegan todos al fondo. Podemos decir que el grafico es un árbol estrictamente binario completo de profundidad dos.



\_ Estos conceptos que son en binario podríamos generalizarlos a un modelo de tipo n-ario.

Ejemplos de pila en la computadora: buscando algún ejemplo de árboles en la computadora tenemos el explorador de carpetas donde al abrir una carpeta parecería que tenemos un árbol con subcarpetas o vemos ya los nodos hojas que son los archivos que están en esa carpeta, a la vez si me meto en otra carpeta lo que tengo son nuevas subcarpetas y nuevos archivos que serían los nodos hojas de esa carpeta y puedo entrar así a todas las subcarpetas que quiera. Entonces la estructura de carpetas de una computadora se asemeja a una estructura de tipo árbol.

Representación ligada: la representación más usada es la ligada o dinámica , donde cada nodo o celda tiene la forma de nodo como lo estuvimos trabajando en lista, pero tenemos la información al medio, un puntero a la derecha que conecta con el nodo hijo derecho y otro puntero a la izquierda que conecta con el nodo hijo izquierdo. Como en este caso es binario tenemos dos punteros que apuntan a los dos nodos hijos. Cada nodo tiene múltiples caminos a seguir, en caso de los binarios solo dos caminos a seguir. Con NULL representamos que la hoja no tiene hijos ni izquierdo ni derecho.





Funciones para administrar un árbol: si consideramos  $p$  como un puntero a un nodo, entonces las operaciones que se pueden realizar sobre un árbol binario son:

- $\text{Info}(p)$ : me devuelve el contenido del nodo.
- $\text{Left}(p)$ : me devuelve como se llama el puntero al hijo izquierdo del nodo, en caso de que no tenga nada devuelve NULL.
- $\text{Right}(p)$ : me devuelve como se llama el puntero al hijo derecho del nodo, en caso de que no tenga nada devuelve NULL.

\_ Por lo general se inventan un par de funciones más, como  $\text{isLeft}(p)$  que me dice si tengo hijo izquierdo y  $\text{isRight}(p)$  me dice si tengo hijo derecho. Otras funciones podrían ser:

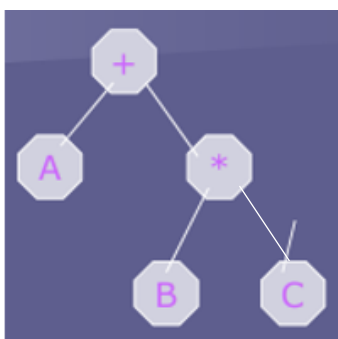
- $\text{Maketree}(x)$ : sería parecida al  $\text{getNode}()$  que teníamos en listas, que me daría un nodo con los hijos, automáticamente todos en NULL para que después lo llene con los datos que me hagan falta.
- $\text{SetLeft}(p, x)$ : podría generar automáticamente un hijo izquierdo y conectarlo al padre mediante un puntero  $p$ .
- $\text{SetRight}(p, x)$ : podría generar un hijo derecho y con un puntero  $p$  conectarlo al padre.

\_ Todas estas funciones conforman la lógica de comportamiento de un árbol como un tipo de dato abstracto, por que como ya hemos hablado la memoria de la computadora no tiene forma de árbol, sino que creemos que los datos están guardados en algo que parece un árbol aunque no es real ya que la memoria sigue siendo lineal y lo que hacemos es emular mediante ciertas funciones y comportamientos tener la sensación de que parece que la estructura parece comportarse como un árbol.

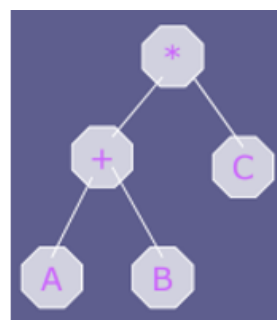
## Aplicaciones

\_ Los árboles tienen múltiples usos, por ejemplo:

Representación y resolución de fórmulas: un uso muy potente de los árboles es por ejemplo para representar ecuaciones, nosotros podríamos representar formulas, como las de a continuación, donde estamos haciendo operaciones con dos términos por eso las podemos representar con un árbol binario, pero si tuviésemos una ecuación de  $n$  términos, podría representarlo con un árbol  $n$ -ario. Entonces, los árboles son muy útiles para representar fórmulas matemáticas, sin duda son la herramienta que las computadoras utilizan para representar y resolver ecuaciones matemáticas.

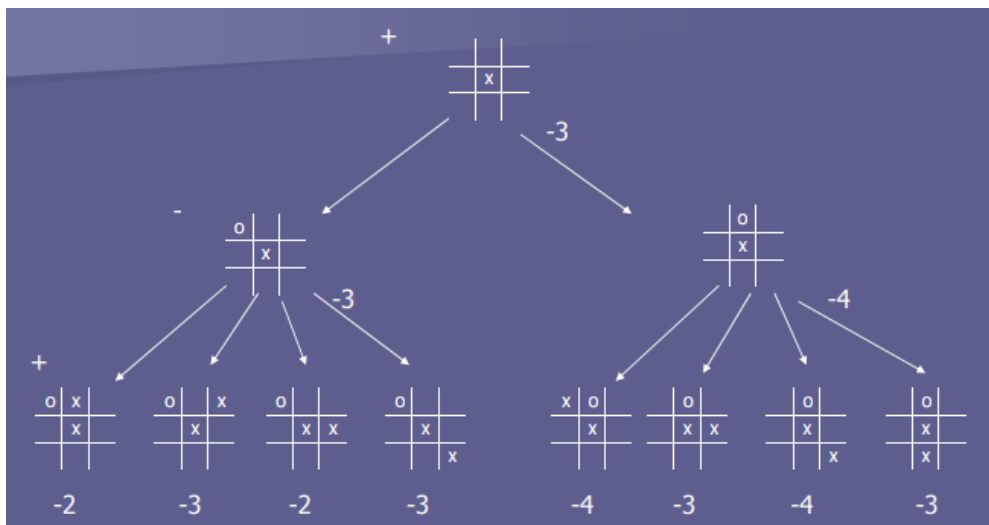


$A + B * C$



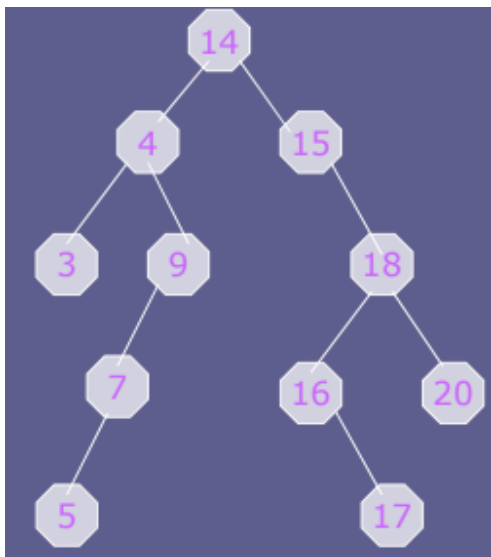
$(A + B) * C$

Juegos: otra estructura por la que se usan muchos los árboles son para los juegos, en general cuando hay juegos de tipos de rivales como cuando hablamos del ajedrez, el TA-TE-TI, a las damas, serían juegos que se podrían representar con un árbol. En el caso del ejemplo del TA-TE-TI, pero al igual que los otros juegos, lo que representaríamos sería donde está la ficha en ese momento, y cuáles son las opciones de respuesta que yo tengo (opción 1, opción 2, que en realidad en el ejemplo tendríamos muchas respuestas por eso sería un árbol n-ario), donde yo muevo una pieza, luego el rival mueve otra y me fijo cuáles son las opciones de respuesta que yo puedo tener y podría elegir alguno de los caminos.



Búsqueda de datos (caso de agregar números): vamos a ingresar números de forma verbal, y lo que vamos a intentar es detectar si alguien del curso dijo algún número repetido con el que dijo otro compañero, solamente hace falta saber si se repitió o no el número, en caso de que haya un número repetido se informa. Cada vez que alguien dice un número los podríamos guardar en un arreglo y vemos si el número que queremos ingresar está o no, si no está lo agregamos al arreglo y así podríamos ir agregando números en el mismo, a medida que más alumnos van diciendo más números el arreglo va a ir siendo cada vez más largo con lo cual cada vez que una persona diga un número, la computadora va a demorar cada vez más tiempo en revisar dentro del arreglo si el número ya está o no guardado dentro del mismo, cada vez ese tiempo se va a ir incrementando cada vez más, entonces como el arreglo no es una estructura de datos dinámica debimos comenzar con asignarle un tamaño para poder administrar todos los datos que van entrando. Si en vez de hacerlo con arreglos lo hacemos con listas enlazadas, acá no tenemos problemas de límites ya que es dinámica y crece todo lo que haga falta, y cada vez que me dicen un número debemos ir controlando si ese número ya lo dijeron para poder decir repetido o no, y en caso de que el número no esté lo agrego al final de la lista, como la lista se va a ir haciendo cada vez más larga, el proceso de revisión también se va a ir haciendo cada vez más lento.

\_ Entonces proponemos ahora resolver este problema con una estructura de tipo árbol, donde la idea sería la siguiente, supongamos que la lista de números es 14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9, 14, 5, cuando entre el numero 14 vamos a ponerlo en el primer nodo del árbol, al ingresar el 15 verificamos en el árbol si el nodo existe o no, como no existe proponemos que como el 15 es más grande que el 14, lo ponemos en la rama de la derecha, en cambio como el número que sigue es el 4 y es menor al 14 lo ponemos en la rama izquierda. Con lo cual el criterio que proponemos es el siguiente, vamos a usar un árbol para guardar los números que la gente dice, si el numero esta repetido aviso que lo está, caso contrario mi decisión es verificar si este es más grande o más chico que el que acaba de entrar para yo elegir si voy a seguir una rama hacia la derecha o una rama hacia la izquierda. Si ingresa en este caso un número más chico que 14 y lo debemos buscar, no usamos la rama derecha y nos ahorramos la mitad de los números y no los controlamos por que como son más grandes que 14 no tiene sentido que este preguntando si es el 6 por ejemplo.



\_ Podemos usar una estructura de tipo árbol para guardar números e ir verificando esos números están repetidos o no en el árbol, y el tiempo que demoro en encontrar o no un número no es tan grande como el que demoraría en un arreglo o en una lista, y esto se debe gracias al criterio que usamos de colocar los mayores a la derecha y los menores a la izquierda.

- Tiempo promedio de búsqueda: si la lista o un arreglo tuvieran 400 números, el tiempo promedio en encontrar un elemento seria  $n/2$ , en este caso 200 intentos. A medida que se van diciendo más números el tiempo se va incrementando. En el caso de un árbol, si son 400 números, podemos encontrarlo en el primer intento o no, pero al tener divididos por más chicos automáticamente descartamos la mitad de los números, y cuando pasamos al nodo siguiente volvemos a dividir y así hasta encontrarlo, pero esto depende de que si el árbol estaría ordenado. Entonces en el

primer intento tenemos 200 números, en el segundo 100, en el tercero 5, cuarto 25 y así hasta el noveno intento donde lo encontramos. El tiempo de búsqueda en un árbol es de  $\log_2 n$ .

\_ La eficiencia del árbol depende mucho de que número dijeron primero, por ejemplo si fuese del 0 al 10 tendríamos una escalera del lado derecho y parecería ser una lista enlazada donde sabríamos que todos los números son mayores a 0 que es el inicio. Es decir, dependemos mucho de cual se al primer nodo del árbol para que queden la mitad de los números más chicos a la izquierda y los más grandes a su derecha y de esta manera ordenados. En este caso los tiempos para encontrar a alguien son muchos más chicos que los de una lista o arreglo.

\_ Este método puede servir por ejemplo para buscar a una persona por el número de su DNI en una base de datos llena de personas, donde en vez de buscar uno por uno entre todas las personas, en una base de datos de 1000000 de personas, en vez de hacer 500000 (promedio de búsqueda) preguntas con un arreglo, con un árbol haría solamente 20 preguntas para ver si la persona está o no. Esta estructura de árbol, dividiendo en mitades acelera la búsqueda de un dato.

Toma de decisión de doble opción: una aplicación de árbol binario puede ser muy útil cuando en cada punto o nodo de un proceso hay que tomar una decisión de doble opción. En el caso anterior tenemos dos opciones, o es más chico o es más grande, y esto dividiría automáticamente la base de datos en dos mitades permitiendo buscar en una de las mitades y en la otra no. Entonces al igual que analizamos el caso de los números supongamos que se quiere encontrar todos los números duplicados que una persona teclea mediante un ingreso de datos, cada número que se ingresa debemos revisarlo contra todos los números ingresados anteriormente, en cambio utilizando un árbol, podemos reducir considerablemente la revisión.

\_ Acá trabajamos con árboles binarios, pero si fuese n-ario podríamos dividir por ejemplo la base en un 33% y así sucesivamente para encontrar un todo con muy pocas preguntas pero a la vez más complejas, por más que haya millones de personas. Por eso seguimos en binario para no complicarnos, donde dividimos en dos mitades.

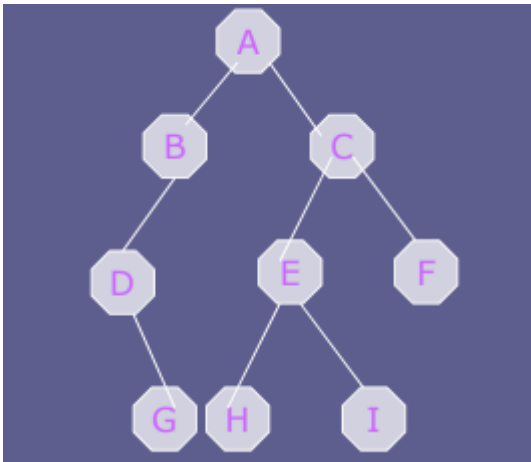
\_ Esta es una ventaja importante del uso de árboles, que se usan muchísimo para lo que es base de datos, por ejemplo buscar gente. Las funciones que vamos a aplicar son todas recursivas por que la definición de un árbol es una definición de tipo recursiva, al ser en sí una estructura dinámica y no lineal, es decir, de múltiples salidas la solución recursiva va a ser una solución muy elegante para resolver este problema de manera muy eficiente.

## Recorrido de árbol

\_ Necesitamos de alguna manera poder recorrer el árbol, poder visitar todos los nodos del mismo, poder ver todas las hojas que tiene y recorrer todas las ramas asegurándonos que pasamos por todos los nodos y también asegurándonos que no repetimos pasar varias veces por el mismo nodo que sería una pérdida de tiempo. Como la definición del árbol es recursiva, la función que necesitamos para recorrerlo también lo será. Este planteo es un poco más complejo por ser una estructura no lineal y debemos asegurarnos de visitar las ramas, no olvidarnos ninguna y no pasar varias veces por la misma rama.

\_ Existen 3 formas más eficientes de recorrer un árbol que nos asegura esto de visitar todos los nodos y de no repetir nodos en varias oportunidades. Estos recorridos son para árboles binarios para hacerlo más simple, pero nosotros vamos a generalizar en algún momento. Algunos métodos nos permiten descender más rápidamente y después visitar las raíces, o primero visitar las raíces y después descender, depende cual sea el problema que tengamos, elegiremos la más conveniente. Estos tardan lo mismo.

Recorrido preorden: u orden con prioridad a la profundidad. La forma de recorrer el árbol es la siguiente, se propone visitar la raíz, luego visitar el árbol izquierdo siempre en preorden y luego visitar el árbol derecho siempre en preorden. Dado el grafico accedemos al árbol por el nodo A que es la raíz y el primero que visitamos, luego visitamos el subárbol izquierdo en preorden (recorremos todo lo que hay), cuando terminamos de recorrer todo el árbol izquierdo vamos a volver de alguna manera de forma recursiva para poder visitar el subárbol derecho también en preorden. Encontramos la raíz al principio.



- En preorden: A B D G C E H I F

\_ A continuación desarrollamos el código. Creamos la función y pasamos un parámetro con el puntero de donde empieza el árbol. Primero mostramos la información que está en el puntero (nodo), caso base. Luego preguntamos si tiene hijo izquierdo, en donde si es así recorremos el árbol en preorden hasta el hijo izquierdo, donde nuevamente volvemos a

llamar la función pero apuntando al nodo izquierdo en el que estamos y así sucesivamente, y mientras se imprimen los valores. Cuando terminamos con el lado izquierdo, preguntamos si tiene hijo derecho y ahora recorremos el árbol en preorden pero entramos al hijo derecho, cuando termina va a return y si no tiene más hijos vuelve hacia la raíz y empieza con la rama derecha.

```
FUNCION recorrerPreorden( puntero )
```

```
Mostrar ( Info( puntero ));
```

```
SI isLeft( puntero )
```

```
    RecorrerPreorden( Left( puntero ));
```

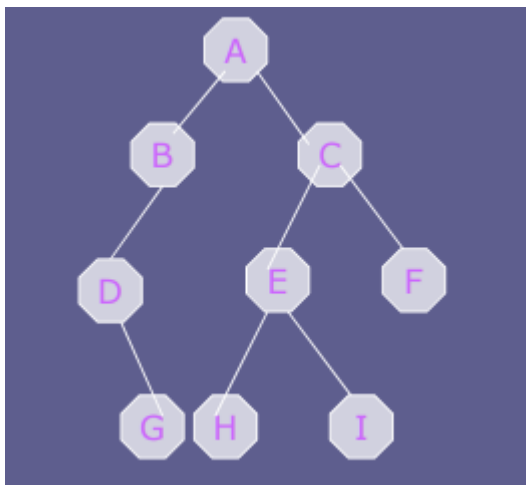
```
SI isRight( puntero )
```

```
    RecorrerPreorden( right( puntero ));
```

```
Return
```

\_ Esta termina cuando muestro el puntero y no tengo ni hijo izquierdo ni hijo derecho. Las funciones recursivas ocupan mucha memoria pero no son lentas, las funciones recursivas no consumen tanto esfuerzo de programación pero la desventaja es que consume más memoria.

Recorrido en orden: u orden simétrico. La idea es la siguiente, primero que nada visito el subárbol izquierdo, después visitamos la raíz y luego de esto recién visitamos el árbol derecho. Visitamos todos los nodos y no repetimos ninguno. Encontramos la raíz al medio.



- En preorden: A B D G C E H I F
- En orden: D G B A H E I C F

\_ En este caso la función del código cambia:

```
FUNCION recorrerOrden( puntero )
```

```
SI isLeft( puntero )
```

```
    RecorrerOrden( Left( puntero ));
```

```
Mostrar ( Info( puntero ));
```

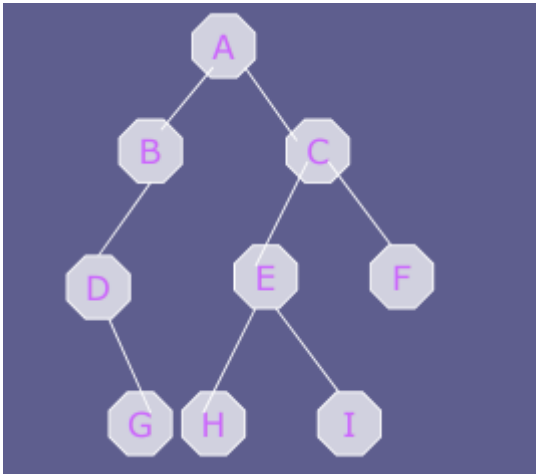
```
SI isRight( puntero )
```

```
    RecorrerOrden( right( puntero ));
```

```
Return
```

\_ Entramos en la raíz y preguntamos si tiene hijo izquierdo, y así hasta que no tenga más hijos izquierdos, visitamos y retornamos el ultimo nodo raíz, luego preguntamos si tiene hijo derecho para continuar y repetir lo mismo. Hasta que terminamos con return.

Recorrido en postorden: la propuesta es la siguiente, primero que nada recorremos el subárbol izquierdo, si no hay hijos en el árbol izquierdo, recorremos el subárbol derecho en postorden, y cuando no haya hijos ni en el derecho ni el izquierdo recién ahí visitamos la raíz. En este visitamos todos los nodos y lo último que visitamos es la raíz. Este método va más rápido a las profundidades. Encontramos la raíz al último.



- En preorden: A B D G C E H I F
- En orden: D G B A H E I C F
- En postorden: G D B H I E F C A

\_ A continuación tenemos el código:

```
FUNCION recorrerPostorden( puntero )
```

```
SI isLeft( puntero )
```

```
    RecorrerPostorden( Left( puntero ));
```

```
SI isRight( puntero )
```

```
    RecorrerPostorden( right( puntero ));
```

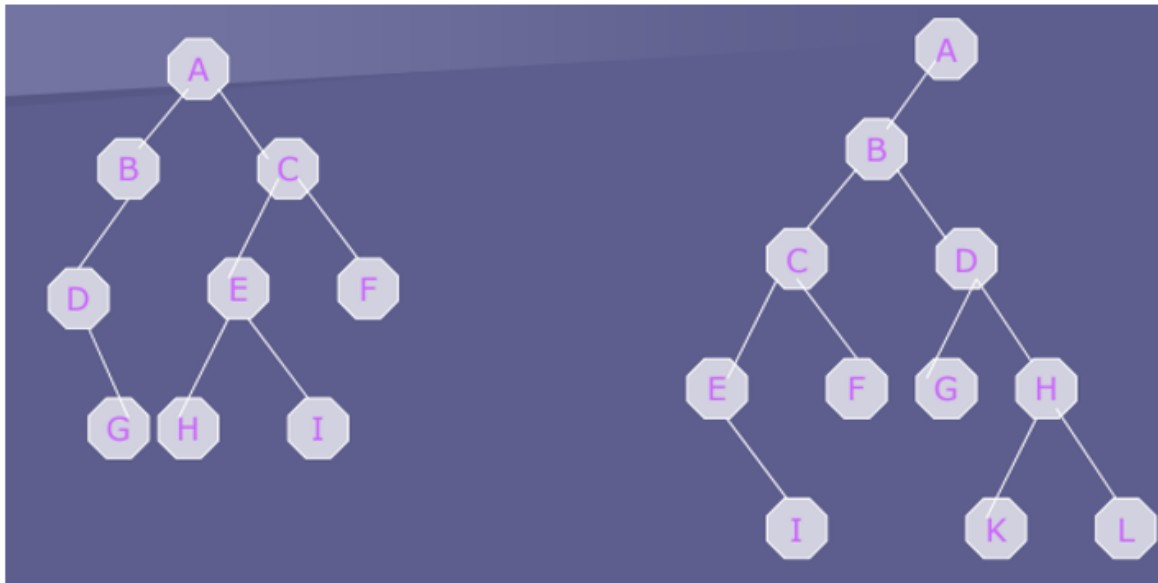
```
Mostrar ( Info( puntero ));
```

\_ Primero preguntamos si tiene hijo izquierdo, luego preguntamos si tiene hijo derecho y por último visitamos y retornamos la raíz.

\_ Acá nos pasa algo parecido a lo que nos pasó con las torres de hanoi, nosotros nos preguntamos como hace la función para armarlas, bueno lo hace porque justamente las funciones recursivas lo que hacen es en vez de mirar el problema como gigante, lo miran al problema como muy chico, y la recursividad es la que me asegura de que todo esto se resuelve fácilmente, no es el problema en si sino que es el proceso recursivo el que me

garantiza esto. Por más que el árbol sea gigante las funciones super simples que tenemos, recorren el árbol asegurándose pasar por todos los nodos siempre con el mismo criterio porque va a seguir siempre la misma forma de recorrer.

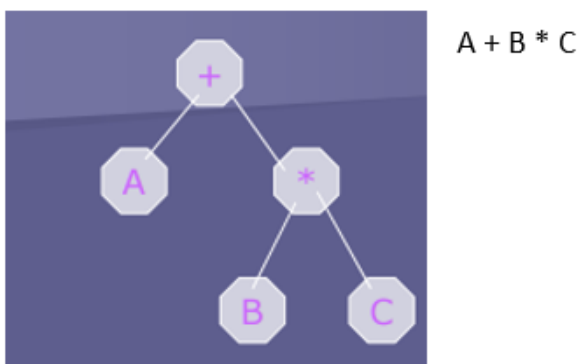
\_ A continuación tenemos ejemplos de recorridos:



- En preorden: A B D G C E H I F
- En orden: D G B A H E I C F
- En postorden: G D B H I E F C A

- En preorden: A B C E I F D G H K L
- En orden: E I C F B G D K H L A
- En postorden: I E F C G K L H D B A

Ejemplo representación de fórmulas: ahora suponemos que tenemos una fórmula matemática.

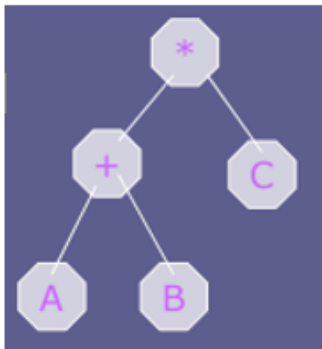


- Cuando recorremos el primer grafico nos queda que cuando recorremos en preorden nos conduce a una expresión en forma prefija:  $+ A * B C$
- Cuando recorremos el árbol en postorden nos conduce a una expresión en forma postfija:  $A B C * +$



- Y por último cuando recorremos el árbol en orden, nos conduce a una expresión infija, con el problema de los paréntesis:  $A + B * C$

\_ Esto significa que cuando tengo una fórmula matemática en un árbol, recorrerlo en preorden produce notación prefija, recorrer en orden produce notación infija, y recorrer en postorden produce una expresión en notación postfija. Por eso están asociados los nombres. Aplicamos lo mismo para el segundo grafico:



$(A + B) * C$

- Recorrido preorden, forma prefija:  $* + A B C$
- Recorrido preorden, forma prefija:  $A B + C *$
- Recorrido preorden, forma prefija:  $A + B * C$

\_ Un árbol me podría servir para pasar de una notación a otra, y dependiendo la forma de recorrer que elijamos será la notación que tendría como resultado. Las tres formas de recorrer el árbol demoran el mismo tiempo en recorrer la ecuación, pero producen distintos efectos como el tipo de notación y el resultado de cada una.

Ejemplo en árbol binario: volviendo al árbol de búsqueda, que usamos para encontrar un numero adentro del mismo, tenemos los números 14 15 4 9 7 18 3 5 16 4 20 17 9 14 5, y a medida que ingresan empezamos a fabricar el árbol, ahora recorremos el árbol en orden quedando 3-4-4-5-5-7-9-9-14-14-15-16-17-18-20, entonces lo que produce el recorrido en orden es poder ver los números ordenados de menor a mayor, con lo cual si tenemos un árbol binario, nos tomamos el trabajo de poner los números más chicos a la izquierda y los más grandes a la derecha, cuando recorra el árbol en orden lo que aparecen son los números perfectamente ordenados de menor a mayor. Entonces los árboles que se arman con el criterio de poner las cosas más chicas del lado izquierdo y las más grandes del lado derecho, son árboles que sirven para buscar gente ahorrando tiempo porque en cada paso divido la base de datos a la mitad, y sirven para ordenar gente.

\_ Los árboles sirven para ordenar datos, buscar datos, y sirven para obtener datos ordenados con muy buenos tiempos de demora en ordenar y en encontrar.

## Conclusión

\_ En resumen, los árboles es una estructura de datos cuya definición es recursiva lo cual vuelve muy fáciles de programar las cosas que queramos hacer con el árbol por que la recursividad nos simplifica los tiempos de programación, solamente consume más memoria pero eso es problema de la computadora, a nosotros nos va a ahorrar tiempo porque las soluciones que hagamos con el árbol van a ser elegantes porque este es recursivo. Los árboles sirven para los juegos, sirven para las ecuaciones matemáticas, sirven para ordenar datos en una base de datos, sirven para buscar gente o números adentro de una base de datos, etc. Los árboles tienen muchas virtudes y son una estructura de datos muy potente, nos van a servir siempre que tengamos que tomar decisiones con distintos caminos y opciones, donde el árbol es una solución que optimiza tiempos pudiendo recorrer y visitar todos los nodos sin repetir alguno, nos permite distintas formas de recorrido con lo cual los resultados que salen son diferentes. Para el caso de los datos repetidos, podemos decir que no entran o que si entran siguiendo el criterio de menor a la izquierda y mayor o igual ( $\geq$ ) a la derecha. Teniendo un buen criterio de ordenamiento podemos buscar y encontrar cosas más rápido.

## Tabla hash

### Introducción

\_ Las tablas de dispersión o funciones de hash son una estructura de datos especial que se usan en aplicaciones que manejan secuencia de elementos. Cada elemento se asocia a una clave (número entero) perteneciente a un rango de valores relativamente pequeños, ósea cada uno de estos elementos debería tener una clave que lo identifique unívocamente, por ejemplo el número de matrícula del alumno.

\_ La potencia central de las tablas hash es poder ubicar un elemento en la tabla con poco esfuerzo de consumo de tiempo, y poder encontrar el elemento también con un poco esfuerzo de consumo de tiempo. Se habla de un tiempo de orden 1, o sea lograr en un solo intento encontrar en un elemento dentro de una tabla de hash. Comparado con árboles, si alguien dice números y nosotros intentamos encontrar ese número en una lista enlazada o en un arreglo, el tiempo que demorábamos es un tiempo relativamente alto y a medida que la tabla se iba haciendo cada vez más grande nuestro tiempo se iba haciendo más lento. Entonces lo que lográbamos con los árboles, en la estructura de árbol binario ordenado con los números más chicos a la izquierda y los más grandes a la derecha, era que nos permitía que nosotros podíamos encontrar un elemento dentro del mismo en mucho menos tiempo que si lo buscamos en un arreglo o en una lista. En el caso de las tablas de hash, buscan conformar una estructura de datos que intenta encontrar un elemento en la tabla en un solo intento. Entonces la potencia de una tabla hash es lograr encontrar elementos en un arreglo en muy pocos intentos, para esto vamos a tener que cumplir ciertas condiciones y esas condiciones si se dan, optimizan el tiempo para guardar

elementos en la tabla de hash y para encontrar elementos en la misma. Esta sería su gran virtud. La velocidad también depende de la cantidad de elementos.

\_ Siempre vamos a hacer la referencia en la que las estructuras de datos tienen ciertas características con ciertas virtudes y defectos, donde nosotros lo que tenemos que conocer es cuáles son las virtudes y los defectos de cada estructura de datos y ver cuál nos conviene utilizar en cada momento, por ejemplo no sirve que hagamos una pila si queremos ver los elementos que están en el medio por que esta no nos deja, o no nos sirve una lista doblemente enlazada si nunca vamos a recorrer para adelante o hacia atrás. Entonces en cada estructura de datos vemos sus cualidades y elegimos la que nos hace falta para resolver cada problema. Las tablas de hash nos van a mostrar ciertas virtudes y ciertos defectos y veremos cuando nos sirve resolver problemas de este tipo usando esta estructura de datos.

Hashing: el hashing es una técnica que tiene por objetivo insertar, borrar y encontrar elementos en un tiempo promedio constante de orden uno. Decimos o se habla de orden en cuanto tiempo nos lleva encontrar algo, pero no se mide en segundos porque esto podría depender de la potencia de cada computadora, sino que cuando hablamos de orden uno decimos cuántas preguntas o cuantos intentos tenemos que hacer hasta encontrar el elemento, entonces cuando decimos orden uno (  $O(1)$  ) quiere decir que en un solo intento nosotros encontramos el elemento que estamos buscando.

\_ Por ejemplo, supongamos que yo tengo un arreglo con 100 números y queremos buscar un número ahí adentro, en este caso la acción que debo hacer en un arreglo es ir preguntando uno por uno los números a ver si son el que busco o no, puede ser que lo encuentre rápido o no pero el tiempo promedio que yo demoro en este caso son 50 intentos para encontrar el elemento, este tiempo promedio sería de un orden 50 (demoro 50 intentos hasta que encuentro el elemento), en las tablas de hash vamos a generar una estrategia que en un solo intento voy a tratar de encontrar el elemento que estoy buscando.

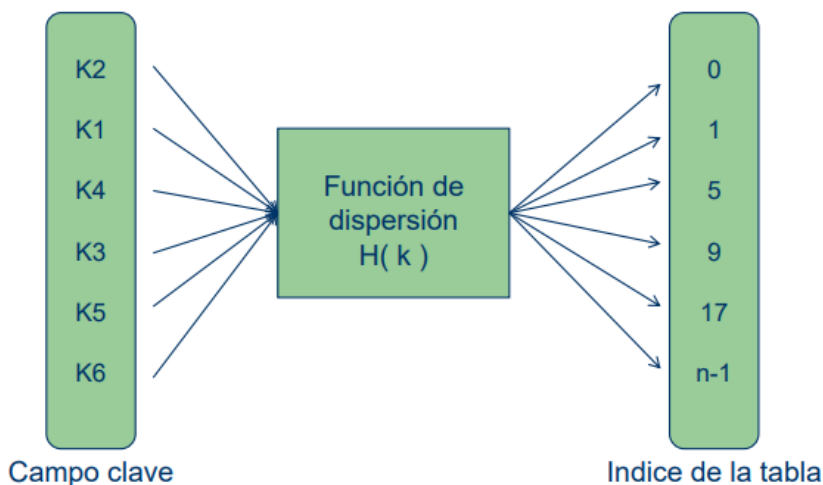
\_ La tabla de hash se puede pensar como un arreglo de tamaño fijo, por ejemplo 1000 elementos, con una cierta estrategia de como guardo los datos adentro de ese arreglo. Entonces luego en un solo intento podríamos encontrar de vuelta el elemento que estamos buscando. En general lo que necesitamos es tener una llave o clave que la voy a mapear en un número entre 0 y 1000 en este caso, pero sería entre 0 y  $n-1$  donde  $n$  es el tamaño de la tabla (0-999).

\_ Entonces la estrategia que vamos a plantear es mapear mediante una función de hash que tiene que ser simple y tenemos que poder encontrar la celda donde el dato está, es decir, el mapeo es llamado función de hash, que idealmente debe ser simple de calcular y debe asegurar que dos llaves distintas correspondan también a dos celdas distintas. Esto último no es posible ya que por principio de cuentas hay un número finito de celdas y podría haber un número infinito de llaves.

## Función de dispersión o hash

\_ Dado el gráfico, tenemos números (K) que queremos guardar, aplicamos una función de hash y eso me ubica una posición dentro de la tabla donde deberíamos poner el elemento. La tabla es de tamaño fijo, eso es un defecto si se quiere, pero a veces vamos a saber que ese tamaño fijo nos alcanza, y K son las claves.

\_ Por ejemplo, supongamos que estamos buscando personas y conozco el número de documento de la persona, la clave K que tenemos es el número de documento, entonces metemos ese número de documento en la función de dispersión, y con ese DNI tenemos que generar un número que vaya entre 0 y 1000-1, ese número me va a llevar a una posición de la tabla de hash y en esa ubicación guardamos el DNI, esto haría que la próxima vez cuando estoy buscando a ver si el DNI por ejemplo 42654882 ya está o no en la tabla, nosotros volvemos a entrar con el número de documento, pasamos por la función de dispersión que vuelve a generar la misma posición de recién y en ese lugar verificamos si está el 42654882, si esta lo encontré en un solo intento y si no está quiere decir que el número no está en la tabla.



\_ Este es un mecanismo sin duda para encontrar elementos en una base de datos, y para ello se asocia esto que llamamos una clave. La lógica que estamos viendo en las estructuras de datos se aplica posteriormente en las bases de datos. Si decimos el número o la palabra clave, pasamos por la función de hash que a esa palabra clave la convierto en una celda y voy a esa celda a buscar el elemento justo en ese lugar, eso hace que la base de datos se comporte de forma muy ágil y rápida si puedo usar funciones de hash en la solución que quiero plantear. Es más fácil buscar a una persona por su DNI o su clave de la UCC que por su nombre y apellido. El tiempo de procesamiento es corto porque la función de hash es una función simple que consume poco tiempo.

\_ El primer defecto de las tablas de hash es que tienen tamaño fijo, o sea el arreglo es de longitud fija (no son dinámicos), y tenemos el problema del límite de tamaño y del desperdicio. Si tenemos un arreglo de 1000 lugares no es lo mismo ubicar gente en un

arreglo de 1000 de 10000 y debemos cambiar la función de hash y usar una distinta, y deberíamos reubicar las 1000 personas en el nuevo arreglo en este caso.

\_ En el caso de las cripto monedas, lo que se hace es guardar o distribuir información en distintos puntos de la nube (block change) para que nadie pueda volver a reconstruir los datos, solamente los puede reconstruir quien tiene la estrategia para llegar a los distintos puntos de la nube y juntar elementos para volver a formar el resultado. Este problema no es puntualmente un problema de tablas de hash, es una estrategia de seguridad que ponen las cripto monedas, lo lento es el block change y no la función de hash ya que esta justamente lo que logra es lo más rápido que se pueda, entonces algunos dicen, ya que tenemos tanta rapidez podemos perder tiempo en buscar los block change en distintos lugares para que en el mix obtengamos más niveles de seguridad, y esta sería la estrategia.

\_ Pero no es lenta la función de hash, al contrario, es lo más rápido que podemos tener. Estas no son aleatorias sino que producen alta dispersión.

Ejemplo 1: como vemos, podríamos usar una tabla hash para nombres de personas o para números. La idea es poder escribir un nombre de persona una clave alfanumérica y mediante la función de hash esto debería aparecer en alguna coordenada de la tabla, entonces cuando se escribe John calcula y da 3 o sea siempre que escribas John lo busquemos en la posición 3 y así con el resto.

John es mapeado a 3  
Phil a 4  
Dave a 6 y  
Mary a 7

0	
1	
2	
3	john 25000
4	phil 31250
5	
6	dave 27500
7	mary 28200
8	
9	

Ejemplo 2: una opción sería sumar los valores ASCII de los caracteres de la cadena. Por ejemplo escribimos la palabra JONES y convertimos cada letra de esta palabra en un código ASCII (cada letra del alfabeto está asociada a un código ASCII que es un número de 0 hasta 255 para poder convertirla en bytes).

- JONES ->  $74 + 79 + 78 + 69 + 83 = 383$ .

\_ La suma de los caracteres debería dar 383, entonces JONES debería estar en la posición 383 pero como tenemos una tabla de 120, dividimos 383 en 120 elementos calculamos el resto (mod) que es 23, entonces guardamos a JONES en la posición 23 de la tabla. La próxima vez que se escriba JONES repetimos el procedimiento y vemos si se encuentra o

no en la tabla. Desafortunadamente si alguien escribiera la palabra STEEN (83+84+69+69+78 = 383) si sumamos sus caracteres también da 383 entonces debería estar en la misma celda en la que esta JONES lo cual nos lleva a un conflicto.

Técnica de plegamiento: por ejemplo el registro de pasajeros de un tren se identifica con un campo (claves) de 6 dígitos, y se va a crear una tabla de dispersión de 1000 registros (posiciones), o sea podríamos guardar hasta 1000 personas en la base de datos. Tenemos las claves que son:

245643                  245981                  257135

\_ Ponemos nuestro número y hacemos alguna función de hash que hace cualquier cosa, como en este dónde nos da como resultado un número que representa la celda de la tabla:

$$h(245643) = 245 + 643 = 888$$

$$h(245981) = 245 + 981 = 1226 \quad \text{se toma } 226$$

$$h(257135) = 257 + 135 = 392$$

\_ De otra forma:

$$h(245643) = 245 + 346 = 591$$

$$h(245981) = 245 + 189 = 434$$

$$h(257135) = 257 + 531 = 788$$

\_ La función de hash puede ser cualquier cosa, de hecho tenemos otra resolución que sería la técnica de mitad del cuadrado que sería elevar el número al cuadrado, y seleccionar por ejemplo 3 dígitos, siempre los mismos para todos los casos, pero cualquiera de estas dos soluciones puede ser funciones de hash. En los números que tenemos como resultados, guardamos la clave del alumno más otros datos como el apellido, nombre, teléfono, dirección, etc.

$$h(245643) = 245643^2 = 60340483449 = 483$$

$$h(245981) = 245981^2 = 60506652361 = 652$$

$$h(257135) = 257135^2 = 66118408225 = 408$$

\_ Entonces en las funciones de hash pueden ser números o pueden ser nombres, yo convierto esos números y esos nombres de alguna forma a una variable numérica que tiene que tener un tamaño límite, por ejemplo si tengo 1000 registros mi variable numérica va de 0 a 1000-1, ninguna persona de la base de datos puede estar fuera de las celdas entre ese rango porque tenemos un arreglo de 1000 personas donde vamos a guardar los datos. La función de hash tiene la habilidad de convertir la clave en una celda del arreglo, y cada vez que escriba la clave el resultado siempre produce la misma celda con lo cual yo vuelvo a intentar encontrar ese dato en un solo intento en lugar de encontrar el dato en muchos intentos. La estrategia de la función de hash es inventar una

fórmula matemática que disperse lo mejor posible los datos, pero por más que lo logremos, siempre va a haber la posibilidad de que dos o varios datos caigan en el mismo lugar, con lo cual la segunda estrategia a definir es una que resuelva las colisiones.

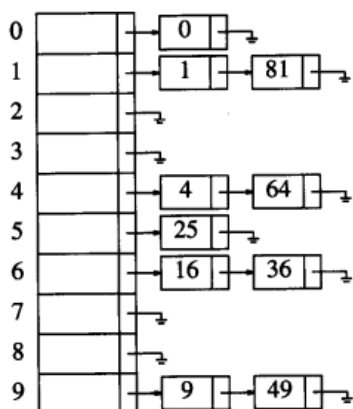
Ejemplo 3: resumiendo, tenemos que meter 1000000 de personas mediante una función de dispersión, en una tabla que tiene 10000 lugares. No hacemos una tabla de un millón de lugares para no desperdiciar espacio, al menos que me haga falta el millón. Entonces la función de hash me convierte las un millón de personas en 10000 lugares. El defecto sería que tenemos capacidad solo para 10000 y si me excedo vamos a tener un conflicto. Otro problema que tenemos es que vengan dos personas con nombres diferentes pero que ambos caen en la misma celda, esto se llama colisiones en las funciones de hash, por más eficiente, espectacular o enredada que haga la función de hash siempre se puede producir una colisión entonces debemos resolver este problema cuando sucede.

## Manejo de colisiones

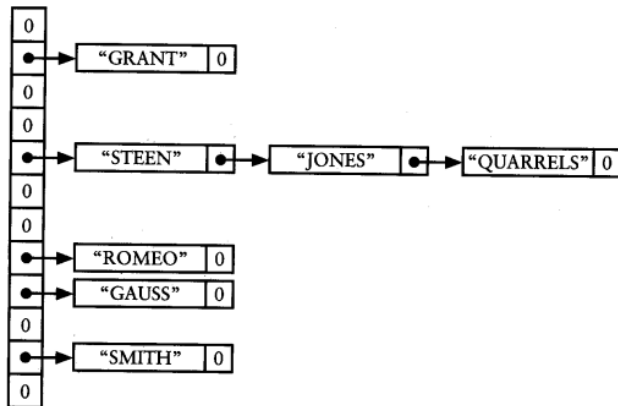
\_ Las funciones de hash tienen que ser lo más eficientes posibles para que los números se dispersen en mi arreglo, pueden ser que dos números caigan en una misma celda. Existen dos formas de manejar las colisiones, una se llama direccionamiento abierto y la otra direccionamiento cerrado. Cuando la estrategia que elegimos produce muchas colisiones la tabla de hash no sirve.

\_ Si tenemos una tabla vacía la probabilidad de colisión es cero, pero a medida que vamos ingresando la probabilidad empieza a aumentar. Si la tabla está muy llena, todos los tiros producen casi colisiones, entonces gracias a esto todos los tiros empiezan a demorar más de uno dependiendo que tan llena esté la tabla.

Open hashing (direccionamiento abierto): se trata de mantener una lista con todos los elementos que sean direccionados a la misma celda. Por ejemplo nosotros escribimos el DNI de una persona y caemos en la celda 6, si esa celda ya está llena, podemos hacer una lista enlazada donde van a estar todos los números relacionados con el 6. Si el arreglo por ejemplo era de 10000 ya no lo es más si no que ahora es de 10000 pero en los lugares donde se produce colisión necesito más posiciones de memoria y enlazar una lista enlazada para poder encontrar la solución. La función es flexible pero el tiempo de encontrar ahora puede ser uno o puede empezar a ser un poco más largo, porque si no lo encuentro en uno lo será en dos o lo que haga falta.



\_ Entonces, el problema de las colisiones que siempre se da, hace que la función de hash no siempre encuentre en un solo tiro, sino que a veces necesito 2, 3, etc, tiros hasta encontrar, y esto se lo resuelve con el hashing abierto donde necesitamos más consumo de memoria, una lista enlazada enganchada a cada una de las celdas, para que si se produce una colisión siga buscando en las celdas consecuentes de la lista. De otra forma sería, las funciones de hash tienen la virtud de encontrar en un solo tiro, pero vemos que esto ya no es tan real, ya que depende de las colisiones, si hay colisiones esto me fuerza a que en vez de un intento sean los que haga falta volviéndose un poco más lento si existen muchas colisiones. En el hashing abierto, cuando se produce una colisión, guardamos los números en una lista enlazada para encontrar todos los que están en una misma línea de colisión. Otro ejemplo sería con nombres de persona que caen en la misma posición:



Close hashing (direccionamiento cerrado): si existe una colisión, se intentan celdas alternas, hasta que se encuentre una vacía. Este hace lo siguiente, por ejemplo si yo busco un número, caigo en una celda y guardo el número, si cuando busco otro número y caigo en la misma celda, en lugar de seguir con una lista enlazada, lo que hago es irme a la celda siguiente, si la celda siguiente está vacía guardo el número. Entonces no crece el tamaño del arreglo, no tengo listas enlazadas a continuación sino que busco un dato y choco, tengo que fijarme en la celda siguiente si está el dato, porque en las colisiones no se resuelven saliendo con una lista sino que yendo a la celda siguiente. No crece el tamaño pero si crece el tiempo porque si no encuentro en un tiro, paso a la siguiente y sino a la siguiente y así nos desplazamos, en el caso de que choque en la última vuelvo a empezar desde el inicio y frenaría cuando encuentro un hueco.

$$h_i(x) = (Hash(x) + f_i(i)) \bmod HSize$$

\_ Con  $f(0)=0$ . La función  $f$  es la estrategia de manejo de colisiones. La fórmula crece cuadráticamente en la medida que  $\lambda$  tiende a uno, o sea se está llenando, se acerca a 0 porque  $1-1$  tiende a cero elevado al cuadrado tiende más a cero y como esta en el factor de división la cantidad de colisiones empieza a crecer exponencialmente. Es decir, a medida que el factor de llenado es más alto las colisiones van a ser mucho mayores de manera exponencial.

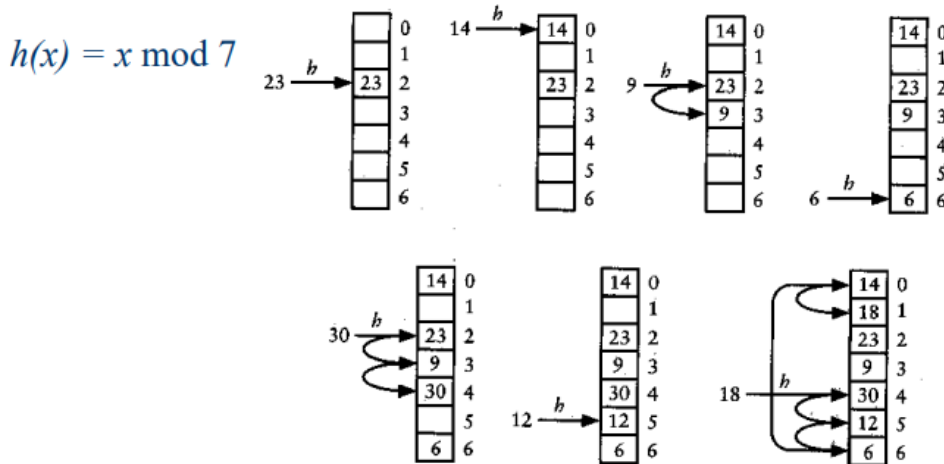
\_ La ventaja acá es, el arreglo es de 10000 por ejemplo y seguiría siendo de 10000, no crece con una lista, y tengo que demorar más tiempo en buscar porque si se vuelve a



producir una colisión debo revisar el siguiente hasta encontrar donde se produce la colisión y el número que estoy buscando. Es decir, al ingresar un número si hay colisión paso al siguiente hasta encontrar un lugar vacío y poner el número, y para buscar busco y si no lo encuentro pasamos al siguiente para ver si la colisión hizo que lo guarde un lugar más abajo cambiando el tiempo dependiendo de si hay muchas colisiones.

\_ A continuación tenemos un ejemplo:

{23, 14, 9, 6, 30, 12, 18}



\_ Existen algunos problemas con esta estrategia:

- El tiempo para encontrar una celda vacía puede ser largo.
- Pueden aparecer bloques de celdas ocupadas (clusters).
- El número esperado de intentos es cercano a  $\frac{1}{2} * (1 + 1/(1 - \lambda)^2)$  búsquedas infructuosas, donde el factor de carga  $\lambda$  es la relación del número de elementos en la tabla al tamaño de la tabla.

\_ Si la estrategia lineal falla, podemos usar una cuadrática que sería en vez de buscar en la celda siguiente, podríamos desplazarnos más celdas aplicando por ejemplo alguna otra función de hash que me reposicione en otro lugar tratando de encontrar lugares vacíos. Entonces la estrategia sería voy al siguiente o intento en otra posición cualquiera mediante una fórmula cuadrática o una función de hash nueva que me meta en otra celda para intentar que no se produzcan colisiones.

\_ Las tablas de hash deberían ser bastante más grandes de lo que me imagino que va a llegar.

\_ La tabla de hash empieza a ser útil cuando los elementos no son tantos, es decir, cuando la cantidad de elementos es limitada y cuando encuentro alguien de un solo tiro o a lo sumo con dos o tres colisiones, ahí la tabla se vuelve óptima. Que el DNI de una persona por ejemplo no cambie es una ventaja que favorece las funciones de hash, pero si cambia la tabla de hash pasa a no ser muy eficiente, entonces que los datos no muten es una

ventaja para las tablas de hash. Entonces las funciones de hash sirven cuando la tabla no va a estar muy llena. Esta estructura es de tipo estático, pero la podemos transformar en dinámica si le enlazamos listas.

Ventajas de las tablas de hash: en vez de que una tabla de hash sea de 1000 lugares por ejemplo, conviene sea de un tamaño correspondiente al de un número primo por ejemplo 977. Las funciones de hash donde yo tengo que buscar entre 977 celdas seguramente me van a forzar a fórmulas más eficientes que si fuera 1000 ya que es un número redondo y no sería tan dispersa la función como en 977. Esta es una ventaja porque los números primos tienen pocos números por los que son divisibles entonces hace que la función de hash se vuelva más extraña y genere más dispersión.

\_ Antes de diseñar una tabla de hash se deben considerar dos cuestiones primero seleccionar una buena función de dispersión y segundo, seleccionar un buen método para resolver colisiones (o hashing abierto o cerrado, depende del problema que tengamos), estos son dos elementos claves.

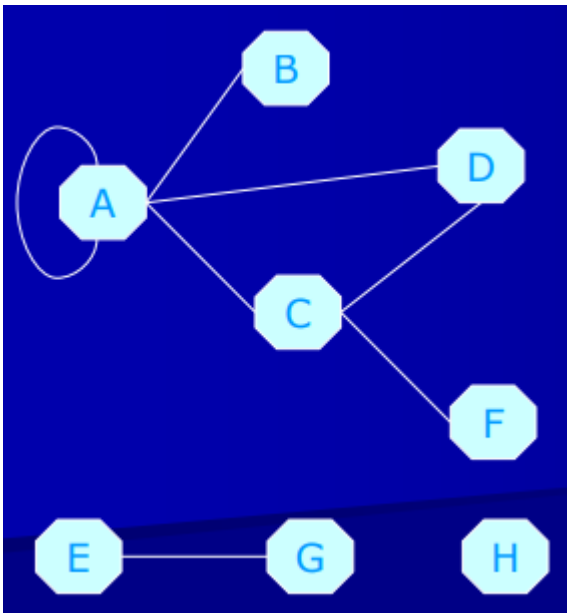
Desventajas de las tablas de hash: el principal inconveniente del direccionamiento abierto es el espacio adicional de cada elemento, es decir, crece el tamaño de la tabla, necesario para enlazar un nodo de la lista con el siguiente nodo, pero se nos puede volver muy grande el espacio en memoria. Por eso debemos tener precaución en hashing abierto hasta cuando puede desarrollarse la memoria. Si la memoria fuese limitada deberíamos hacer hashing cerrado y no abierto para no poder no arriesgar espacio del disco

\_ Si el hashing es cerrado y la exploración es lineal, el principal inconveniente, es cuando el factor de carga de la tabla es superior al 50% empieza a haber muchas colisiones, y en estos casos se empieza a volver más lenta la tabla de hash, es decir, situar los elementos en posiciones contiguas aumenta el tiempo medio de la operación de búsqueda volviéndose ineficiente.

# Grafos

\_ Un grafo  $G$  es una estructura formada por un conjunto  $V$  de vértices y un conjunto  $E$  de pares de vértices llamados arcos o aristas. En un árbol había algo parecido a vértices, que se llamaba padre y tenía hijos, no había arcos ni aristas sino que se conectaban los nodos padres con los nodos hijos y no se tenía en cuenta el nombre de esa conexión pero aquí sí. En un árbol hay jerarquía de datos, es decir, sabemos quién es el padre y sabemos quién es el hijo, pero en un grafo no ya que en este las conexiones son como nos interesen y ninguno es más importante que el otro. Los nombres que se les asigne a los vértices pueden ser números o etiquetas, lo que se necesite. No es una estructura lineal.

\_ Un grafo consta de un conjunto de nodos (o vértices) y un conjunto de arcos (o aristas). Cada arco del grafo se especifica como un par de nodos. Un grafo no necesita ser un árbol, pero un árbol si es un grafo. Un nodo no necesita tener arcos asociados. Esta imagen representa un grafo, donde los nodos son  $\{A, B, C, D, E, F, G, H\}$  y  $(A, B)$ ,  $(A, D)$ ,  $(A, C)$ ,  $(C, D)$ ,  $(C, F)$ ,  $(E, G)$ ,  $(A, A)$  son los arcos. Si los pares de nodos que constituyen los arcos son pares ordenados, se dice que el grafo es un grafo dirigido:

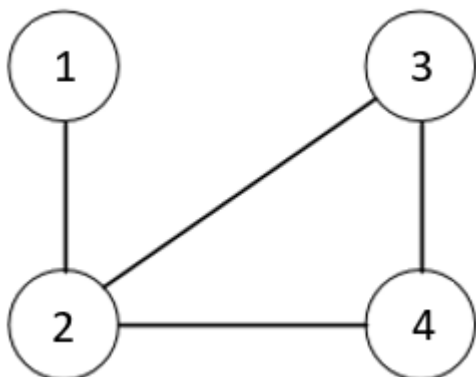


\_ Grafo es un concepto matemático, basado en conjuntos, no es un concepto computacional pero se lo utiliza mucho dentro de la ingeniería, ya que tiene mucha aplicabilidad, pero es un concepto que existe desde antes de la ingeniería.

- $V(G)$ , conjunto del vértice del grafo.
- Los arcos del grafo se pueden nombrar como pares ordenados, cuando es dirigido y hay flechas tenemos  $E(G) = \{(1,2), \dots\}$  pero también tenemos el caso donde no hay flechas y no indique el sentido, por eso puede ser  $(1,2)$  o  $(2,1)$ , entonces

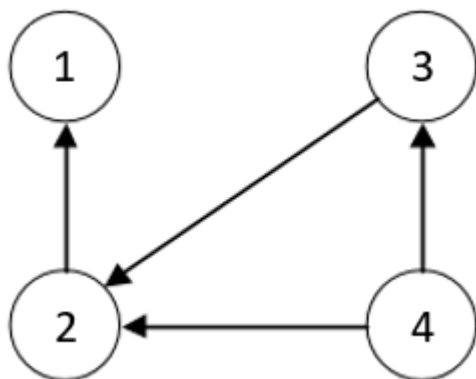
cuando tenemos de ida y de vuelta podemos usar la notación  $E(G) = \{<1,2>, \dots\}$  para representar esas conexiones.

Grafo no dirigido: significa por ejemplo que da lo mismo ir de 1 a 2 que de 2 a 1, no hay diferencia y no se ponen ambas direcciones sino que se pone solo una indicando que existen las dos y notando con los  $< >$ . Decimos de otra forma que es no dirigido si el par de vértices  $(V1, V2)$  que representa cualquier arco no es ordenado.



- $V(G) = \{1, 2, 3, 4, 5\}$
- $E(G) = \{<1,2>, <2,3>, <2,4>, <3,4>\}$

Grafo dirigido: también se llama dígrafo, se define cuando el par de vértices  $(V1, V2)$  que representa cualquier arco es ordenado, es decir, no es lo mismo ir de 1 a 2 que de 2 a 1 si la flecha lo indica. Como acá si son importantes las direcciones de los caminos de las aristas o arcos, entonces los arcos se nombran con pares ordenados.



- $V(G) = \{1, 2, 3, 4, 5\}$
- $E(G) = \{(2,1), (3,2), (4,2), (4,3)\}$

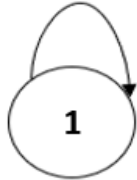
\_ De los gráficos ambos son grafos, ambos tienen más o menos los mismos vértices pero ambos no tienen los mismos arcos o aristas. El gráfico 1 al no tener flechas se considera grafo no dirigido y el segundo gráfico al tener flechas se considera grafo dirigido.

\_ Una de las restricciones básicas de los conjuntos es que los elementos no pueden repetirse. Matemáticamente en un conjunto los elementos no se pueden repetir ya que

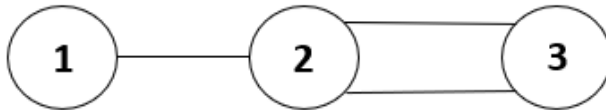
dejaría de ser un conjunto. Un grafo al ser una entidad matemática, debe de seguir las normas matemáticas.

Restricciones:

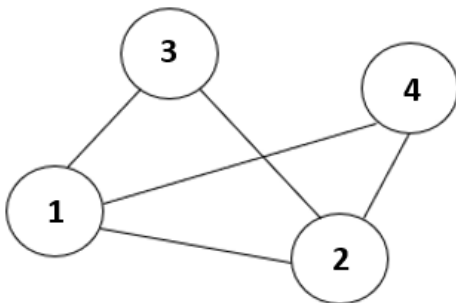
- En un grafo no puede haber arcos recursivos, por ejemplo empezar en el vértice 1 y terminar en el vértice 1, ni siendo dirigido y no dirigido.



- Como  $E(G)$ , que es la representación matemática de un grafo, es un conjunto el par ordenado  $(V1, V2)$  cualquiera de esos dos solo puede aparecer una vez, es decir,  $G$  no puede tener 2 o más arcos porque debe de seguir las reglas de los conjuntos. Cuando esto ocurre, el objeto resultante se llama multígrafo. No podemos tener arcos duplicados porque no cumple las normas matemáticas de un conjunto.

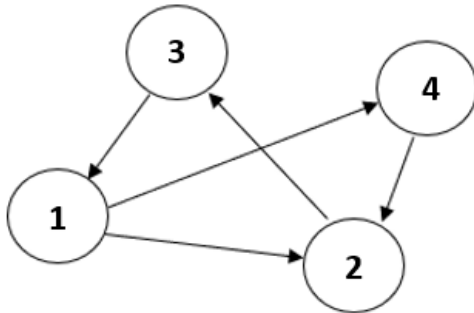


Adyacencia: si  $\langle V1, V2 \rangle$  es un arco de  $E(G)$ , siendo  $G$  un grafo no dirigido, se dice que los vértices  $V1$  y  $V2$  son adyacentes y que el arco  $\langle V1, V2 \rangle$  es incidente en esos vértices. Si tenemos cualquier vértice por ejemplo el vértice 1 la pregunta es cuales son los adyacentes a este y es tan sencillo como a los que puedo llegar directamente desde el vértice 1, y lo mismo para el resto de los vértices. No se trabaja mucho con la definición de incidencia pero más comúnmente se trabaja con la definición de adyacencia pero es importante conocer que puede existir incidencia.



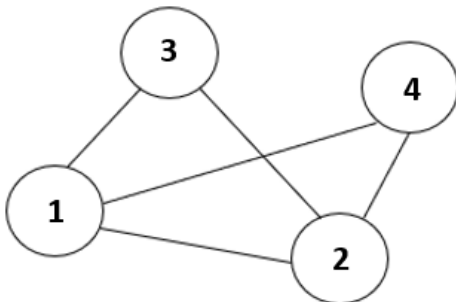
- $Adyacentes(1) = \{2, 3, 4\}$
- $Adyacentes(2) = \{1, 3, 4\}$
- $Adyacentes(3) = \{1, 2\}$
- $Adyacentes(4) = \{1, 2\}$

\_ Si estamos trabajando con grafos dirigidos, aquí se dice que un vértice  $V_1$  es adyacente a  $V_2$  mientras que  $V_2$  es adyacente desde  $V_1$ , esta dirección depende de la flecha ya que me dicen a cuáles puedo llegar desde el vértice que este.



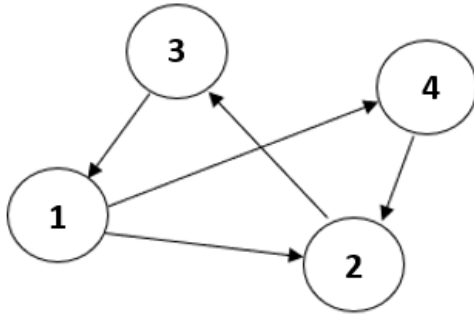
- $\text{Adyacentes}(1) = \{2, 4\}$
- $\text{Adyacentes}(2) = \{3\}$
- $\text{Adyacentes}(3) = \{1\}$
- $\text{Adyacentes}(4) = \{2\}$

Grado: en un árbol cada cantidad de hijos de cada nodo es el grado de cada uno de los nodos del árbol, donde el mayor de esos es el grado del árbol, o sea el máximo número de hijos que puede tener un nodo del árbol. Acá hay una pequeña diferencia con los grafos, en estos no se obtiene o no se dice que hay el grado de un grafo, únicamente se habla del grado de los vértices y hasta ahí no hay grado del grafo como tal. El grado de un vértice es el número de arcos incidentes a dicho vértice, entonces por ejemplo me fijo en el vértice 1 para saber cuál es su grado, solo veo cuantas líneas están conectadas a ese vértice.



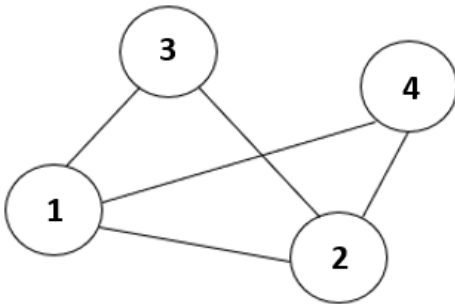
- $\text{Grado}(1) = 3$
- $\text{Grado}(2) = 3$
- $\text{Grado}(3) = 2$
- $\text{Grado}(4) = 2$

\_ Cuando se trabaja con grafos dirigidos, se habla de grado de entrada de un vértice como el número de arcos que llegan a ese vértice y el grado de salida como el número de arcos que parten de él.



- $\text{Grado}(1) = 3$  (1 entrada + 2 salida)
- $\text{Grado}(2) = 3$  (2 entrada + 1 salida)
- $\text{Grado}(3) = 2$  (1 entrada + 1 salida)
- $\text{Grado}(4) = 2$  (1 entrada + 1 salida)

Camino: se llama camino o trayectoria desde el vértice  $V_i$  que puede ser cualquiera, al vértice  $V_j$  que puede ser cualquiera también en un grafo  $G$ , a la secuencia de vértices  $V_i, V_{k1}, V_{k2}, \dots, V_{kn}, V_j$  tales que  $(V_i, V_{k1}), (V_{k1}, V_{k2}), \dots, (V_{kn}, V_j)$  pertenecen al conjunto de arcos del grafo  $G$ . La longitud de ese camino es el número de arcos que la forma. La idea es empezar en algún vértice y terminar en otro vértice que puede ser inclusive el mismo, aquí no hay restricciones y se pueden repetir los caminos.

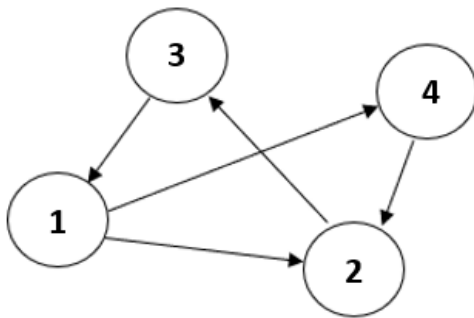


- $\text{Camino}(4-3) = \{4, 2, 1, 3\}$
- $\text{Longitud}(4-3) = 3$

\_ Un camino simple es un camino en el que todos los vértices excepto quizá el primero y el ultimo son distintos, el quizá hace referencia a que el primero y el ultimo son los únicos que si se pueden repetir, esto lleva a un ciclo. Para el caso de grafos dirigidos, tanto en caminos como en caminos simples se agrega el prefijo dirigido. Este camino es aquel en el que no se me permiten repetir elementos. Tomando el ejemplo anterior tenemos que:

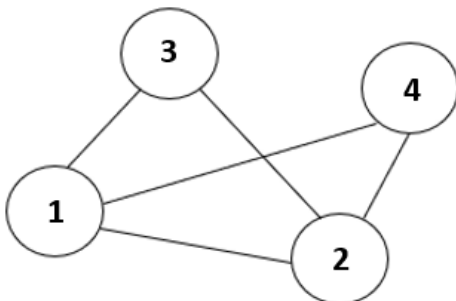
- $\text{Camino}(4-3) = \{4, 2, 1, 3\}$
- $\text{Longitud}(4-3) = 3$

\_ No podríamos hacer por ejemplo {4, 2, 1, 4, 2, 1, 3} porque se repiten. Ahora en el caso de los grafos dirigidos:



- Camino(4-1) = {4, 2, 3, 1}
- Longitud(4-1) = 3

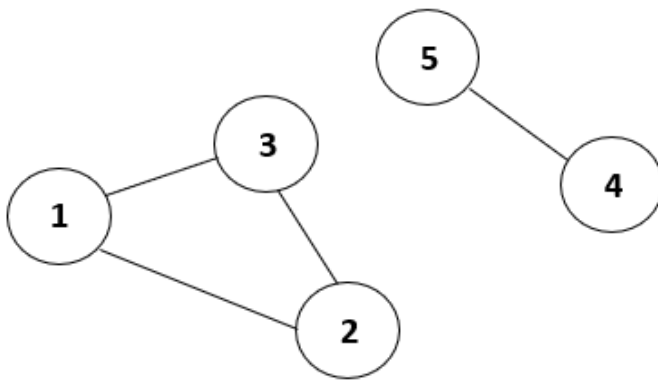
Ciclo: es un camino simple en la cual el primero y el último vértice coinciden, y además la longitud del camino es mayor o igual a 2 en los grafos dirigidos y mayor o igual a 3 en los no dirigidos. Obligatoriamente tiene que repetirse el primero y el último pero no se puede repetir ninguno de los intermedios. En estos caminos pueden ser infinitos pero ciclos tal vez hay un límite.



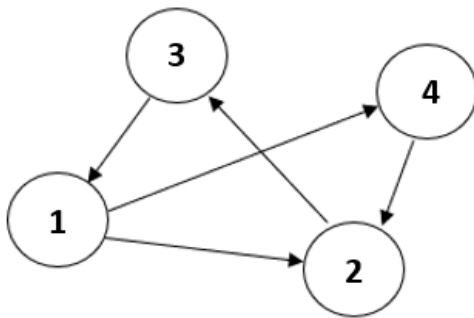
- Camino(4-4) = {4, 2, 3, 1, 4}
- Longitud(4-4) = 4

\_ En un grafo G no dirigido se dice que dos vértices  $V_i$  y  $V_j$  están conectados si existe un camino de  $V_i$  a  $V_j$ . Por ser G no dirigido obligatoriamente debe existir también un camino de  $V_j$  a  $V_i$ . Un grafo G no dirigido se llama conexo si para cada par de vértices  $V_i, V_j$  distintos, existe un camino que los une, es decir, todos los vértices están conectados entre sí. Se dice por ejemplo que el vértice 3 y el vértice 4 están conectados si es que hay manera de llegar de 3 a 4 cualquiera que sea. Todos los grafos dirigidos y sobre todos los no dirigidos que hemos visto hasta el momento son conexos, pero como vemos a continuación, tenemos un grafo no conexo el cual debemos aclarar en la definición del mismo que se trata de un grafo y no dos, cuando están separados a cada una se las conoce como componentes conexas por lo tanto el que tenemos tendría dos componentes conexas.

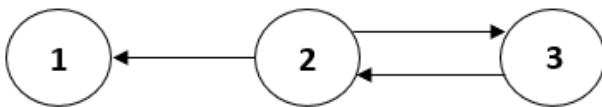




\_ Un grafo  $G$  dirigido es fuertemente conexo si para cada par de vértices distintos  $V_i, V_j$  existe un camino dirigido de  $V_i$  a  $V_j$  y de  $V_j$  a  $V_i$ . Entonces en un grafo dirigido se dice que es conexo que en la misma definición que en el grafo no dirigido. Tiene todas las posibles conexiones.



Subgrafo: un subgrafo  $G_1$  de un grafo  $G$  es un grafo tal que el conjunto de vértices de  $G_1$  es un subconjunto del conjunto de vértices de  $G$  y el conjunto de arcos de  $G_1$  es un subconjunto del conjunto de arcos del grafo  $G$ . O sea si tenemos un grafo  $G$ , un subgrafo es cualquier grafo que tenga al menos vértices similares o los mismos pueden ser todos o algunos de ellos, y también los arcos pueden ser todos o algunos de ellos. Por ejemplo  $(2, 3)$  es un subgrafo del grafo  $(1, 2, 3)$ :



Subgrafo máximo: es aquel que sigue las normas de un subgrafo y es el más grande que se puede tener, y el mayor de cualquier grafo es el mismo grafo. Un subgrafo  $G_1$  de un grafo  $G$  se dice que es máximo si no existe ningún subgrafo  $G$  que lo contenga (el subgrafo máximo de cualquier grafo, es el mismo grafo).

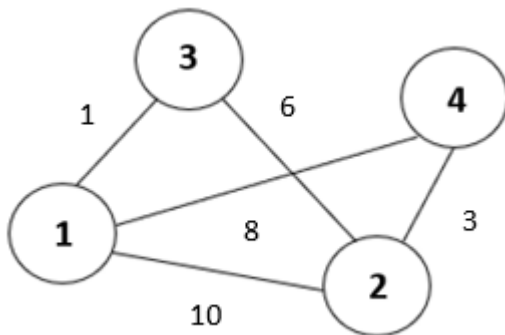
Subgrafo mínimo: de este no se puede hacer un gráfico directamente. Un subgrafo  $G_1$  de un grafo  $G$  se dice que es mínimo si no existe ningún subgrafo  $G$  diferente del vacío que este contenido en  $G_1$ . Cuando hablamos de diferente del vacío, no podemos decir que el subgrafo mínimo de un grafo es el vacío porque no se acepta. Es decir, si dividimos este

grafo en algunos subgrafos, si habrá alguno que sea mínimo pero mientras no hagamos eso no podemos saber cuál es el mínimo.

Componente conexa: una componente conexa de un grafo no dirigido es un subgrafo conexo máximo. Sabemos que es una componente conexa porque es lo máximo que podemos obtener de cada sección del grafo dividido.

Componente fuertemente conexa: de un grafo dirigido es un subgrafo máximo que es fuertemente conexo. Sección que me permite ir y volver por todos los nodos. En este grafico seria (2, 3).

Grafo valuado o etiquetado: es aquel en el que cada arco  $E(G)$  tiene asociado un valor (peso/coste) en el arco. El coste de un camino es la suma de costes de los arcos que la forman. Como vemos en el gráfico, están puestos números en cada arco, donde estos representan costes o pesos. A los grafos se los puede utilizar por ejemplo para calcular tendidos eléctricos entre ciudades, entonces suponemos que cada vértice es una ciudad y los tendidos eléctricos son los arcos que están entre las ciudades donde cada uno de los tendidos tiene su peso. También ese número podrían representar longitudes, entre otras cosas que se necesite. No confundir entre longitud y coste/peso.



- Camino(1-4)= {1, 3, 2, 4}
- Coste= 10

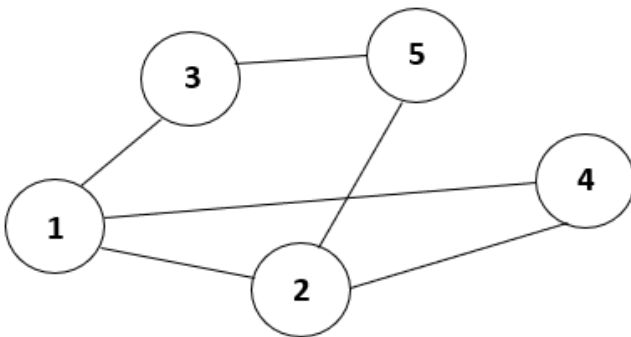
## Recorridos sobre grafos

\_ En arboles vimos los recorridos preorden, en orden, y postorden, pero en realidad hay dos recorridos principales en árboles que son en anchura y en profundidad, y en el de profundidad se divide en los tres que nombramos al comienzo.

\_ En grafos no están los tres recorridos en profundidad, sino que aquí están solo los dos principales. La operación de recorrer un grafo, consiste en partir de un vértice determinado y visitar todos aquellos vértices que son accesibles desde el, siguiendo un orden específico. Según se opere este orden, hablamos de los siguientes recorridos: en profundidad y en anchura.

### Recorrido en profundidad:

- 1)\_ Se visita el vértice V de partida, es decir, debo de tomar cualquier vértice para empezar.
- 2)\_ Se selecciona un vértice W adyacente a V y que no haya sido ya visitado.
- 3)\_ Se continua el recorrido en profundidad a partir de W.
- 4)\_ Cuando se encuentra un vértice cuyo conjunto de adyacentes ha sido visitado en su totalidad, se retrocede hasta el último vértice que tenga adyacentes no visitados, y se vuelve al segundo paso, donde justamente al regresar al vértice anterior es recursivo, ya que una de las razones de la recursividad es que se puede dejar procesos pendientes para tratarlos después.



- Los recorridos son:

- 1, 2, 4, 5, 3
- 1, 3, 5, 2, 4
- 1, 4, 2, 5, 3
- 1, 2, 5, 3, 4

\_ Esto quiere decir que a diferencia de los árboles donde cada recorrido es único, en los grafos los recorridos no necesariamente son únicos, sino que hay otras posibilidades.

Recorrido en anchura: es más corto el algoritmo, son solo dos pasos, pero es más difícil de implementar, y esto mismo sucede en árboles.

- 1)\_ Se visita el vértice V de partida. Segundo se visitan todos los adyacentes de V que no hayan sido visitados, antes de visitar otros vértices por ejemplo 1, vemos cuales son adyacentes a este y a esos los vamos a visitar antes de ir a otro, es decir, elegimos cualquiera de ellos y vemos los adyacentes que nos quedan.

- Los recorridos son:

- 1, 2, 3, 4, 5
- 1, 3, 4, 2, 5
- 1, 4, 3, 2, 5

- 1, 2, 4, 3, 5
- 1, 4, 2, 3, 5
- 1, 3, 2, 4, 5

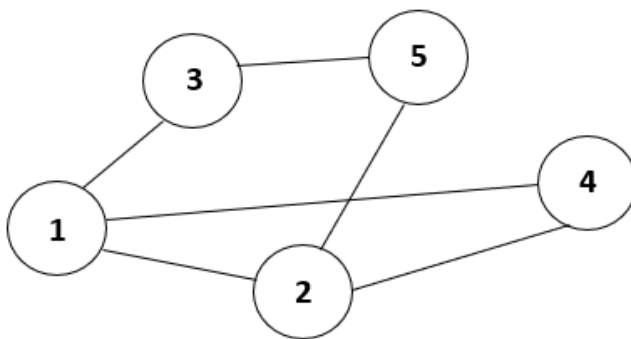
## Implementación de grafos

\_ Existe la posibilidad de implementar grafos como debemos de haber implementado los árboles, es decir, con dos referencias por cada clase por cada nodo, pero al ser grafos no es lo más recomendable porque se vuelve muy difícil trabajar con un arreglo de referencias o con una lista, posiblemente si existe ese tipo de implementación pero no es lo más recomendable. En ese sentido existen unas estrategias de implementación que nos permiten trabajar con grafos sin implementar directamente como de deben haber implementado los árboles, a continuación vemos las posibilidades:

Matriz de adyacencia: se utiliza una matriz cuadrada cuyo número de filas y columnas sea igual al número de vértices existente, la lectura siempre debería hacerse desde una fila hacia una columna. Esta es una implementación estática, por lo tanto no pueden cambiar de tamaño. Esta es una representación del grafo en donde yo puedo ir desde el grafo hacia la matriz o también desde la matriz hacia el grafo.

- Ventajas: su implementación es fácil, algunos algoritmos usan directamente la matriz de adyacencia, operaciones rápidas.
- Desventajas: número de vértices limitado, además hay desperdicio de memoria ya que hay muchas casillas en la matriz que no son utilizadas pero igual se las ha declarado.

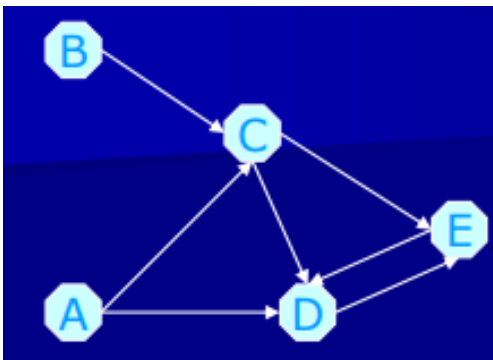
\_ A continuación trabajamos con un grafo no dirigido, donde ponemos tanto en las filas como en las columnas el nombre de los vértices, como no hay arcos recursivos no se puede ir de un vértice a sí mismo por eso en el gráfico ponemos F (false) para representar la unión de 1 y 1, 2 y 2 y así sucesivamente. Cuando tenemos vértices adyacentes al que estamos indicando colocamos T (true) de lo contrario F. La matriz que obtuvimos como resultado, es especial y tiene como un espejo a partir de la diagonal principal, y se llama matriz simétrica pero no todas las matrices que representan grafos son simétricas.



	1	2	3	4	5
1	F	T	T	T	F
2	V	F	F	V	V
3	V	F	F	F	V
4	V	V	F	F	F
5	F	V	V	F	F

\_ Estamos tratando de representar un grafo con un arreglo, lo hacemos con una matriz donde las filas y columnas son la cantidad de nodos, pero al representar un grafo como arreglo adquirimos todas los limitantes de una arreglo, pero podemos hacer algún cálculo matemático para obtener la información y esa sería nuestra ventaja siempre y cuando tengamos pocos nodos y medianamente ordenados. Además de T o F puede ser 1 o 0.

\_ La matriz de cierre transitivo representa la conexión de todos los nodos sin importar si es de orden uno o dos, sin importar como se llega hasta los nodos siempre y cuando se pueda llegar. Garantiza que hay una trayectoria entre los nodos i y j (de cualquier longitud).



	A	B	C	D	E
A	0	0	1	1	0
B	0	0	1	0	0
C	0	0	0	1	1
D	0	0	0	0	1
E	0	0	0	1	0

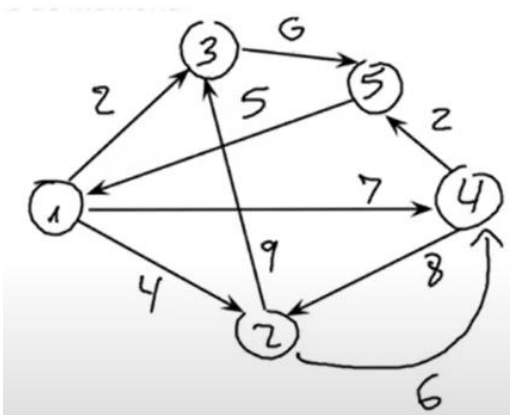
**Matriz**

	A	B	C	D	E
A	0	0	0	1	1
B	0	0	0	1	1
C	0	0	0	1	1
D	0	0	0	1	0
E	0	0	0	0	1

**Matriz de orden 2**

	A	B	C	D	E
A	0	0	1	1	1
B	0	0	1	1	1
C	0	0	0	1	1
D	0	0	0	1	1
E	0	0	0	1	1

\_ Trabajando con el grafo dirigido, hacemos la matriz, en este caso usamos los valores de los costos en vez de true y false. Como la diagonal principal no va a existir le colocamos 0 o el -1. Si tomamos la matriz resultante, podemos volver a dibujar el mismo grafo. Para un grafo valuado tenemos, colocamos sus costos en vez de las uniones:

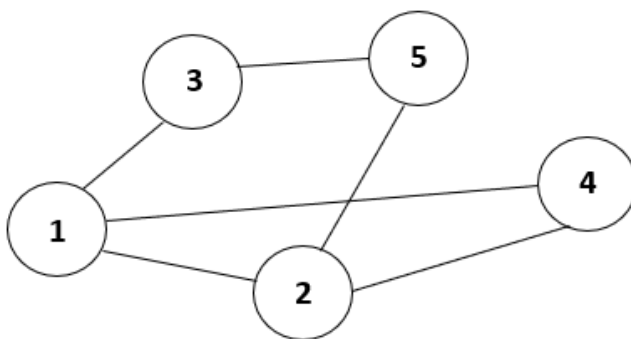


	1	2	3	4	5
1	0	4	2	7	0
2	0	0	9	6	0
3	0	0	0	0	6
4	0	8	0	0	2
5	5	0	0	0	0

Dos conjuntos: se implementa un conjunto de vértices y un conjunto de arcos. Si se desea representar conjuntos con todas sus restricciones, se puede usar el TDA (tipo de dato abstracto) Collection, en los lenguajes que dispongan de este tipo de datos, pero también se pueden usar otras estructuras teniendo cuidado de no romper las reglas que rigen a los conjuntos. Por ejemplo podríamos usar dos arreglos, uno para los vértices, y otro para los pares ordenados de los arcos pero teniendo cuidado de que no se me repitan los números, siempre y cuando sean dos conjuntos podemos usar cualquiera de las estructuras que conocemos que permitan representarlos.

- Ventajas: usando un TDA existente en el lenguaje, la implementación es fácil. Utiliza la definición matemática de grafo.
- Desventajas: puede volverse complicado tanto la implementación como verificar que no haya datos repetidos.

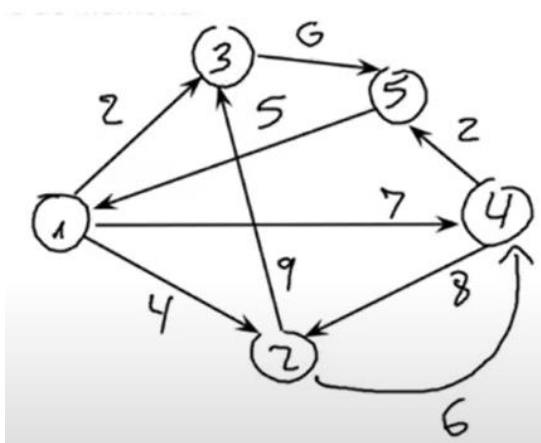
\_ Dado los grafos que usamos como ejemplo, en el caso del no dirigido usamos un arreglo de una dimensión para representar los vértices y otro arreglo de dos dimensiones para representar los arcos. Con esos dos podemos volver a dibujar el grafo.



Vértices
1
2
3
4
5

Arcos	
1	2
1	3
1	4
2	4
2	5
3	5

\_ Para el grafo dirigido aplicamos la misma lógica pero usamos un arreglo de tres columnas para representar los pesos, y luego con estos podemos dibujar el grafo.



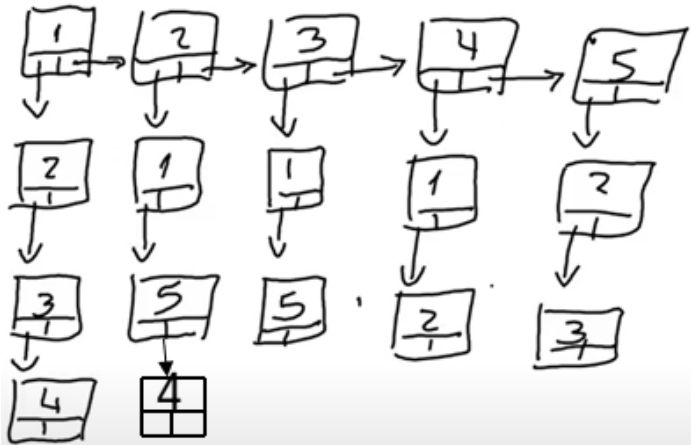
Vértices
1
2
3
4
5

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

Lista de listas: hace referencia a una lista de vértices y para cada uno de ellos otra lista con sus adyacentes.

- Ventajas: no hay límites de vértices, es una estructura dinámica por lo tanto la podemos hacer crecer lo que necesitemos. Solo se usa una estructura para la implementación (igual que con la matriz de adyacencia).
- Desventajas: la implementación de una lista de listas va a ser consecuentemente más difícil y complejo que solo una lista, y también a la hora de recrear los algoritmos sobre grafos. Va a ser más difícil hacer recorridos u operaciones.

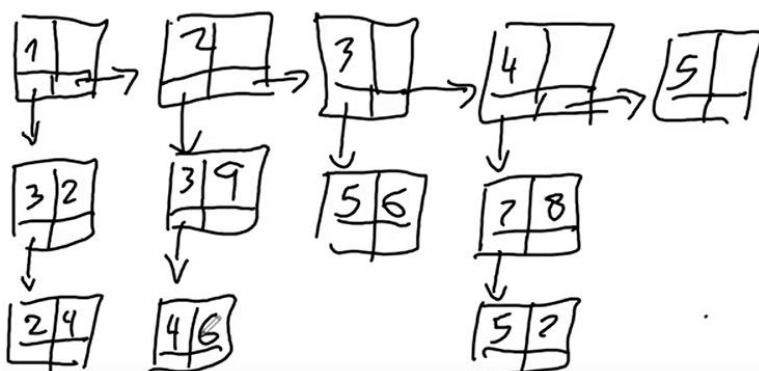
\_ Se utiliza la primera fila o la primera columna para la representación de los vértices. En una lista simple tenemos un puntero para el siguiente nodo, pero como esta es una lista de listas en la que en lugar de tener un solo puntero van a haber dos. En la primera fila o columna ponemos los vértices, ahora el segundo puntero o la segunda lista nos va a servir para poder indicar los adyacentes o los pesos que son los que estarían para abajo.



\_ Cuando trabajamos con el grafo dirigido, creamos la clase que representa a los nodos, donde ahora cada uno de los nodos además de tener dos apuntadores (horizontal y vertical) va a tener dos datos, uno para el vértice y el otro para el peso.

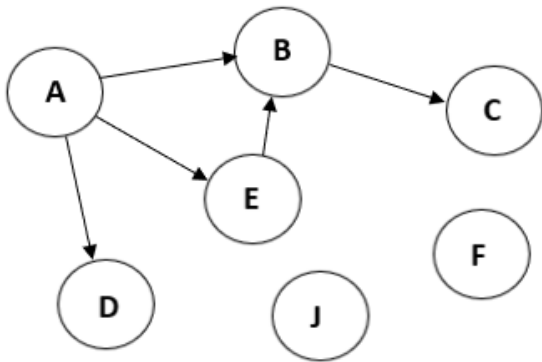
```

Class nodo{
    Float peso
    Int elemento;
    Nodo aph;
    Nodo apv;
}
  
```

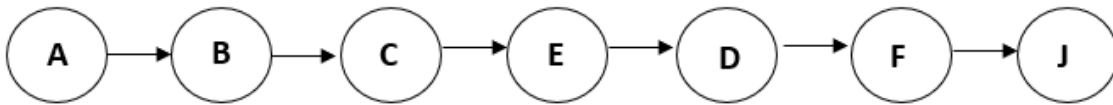


## Representación ligada de grafos

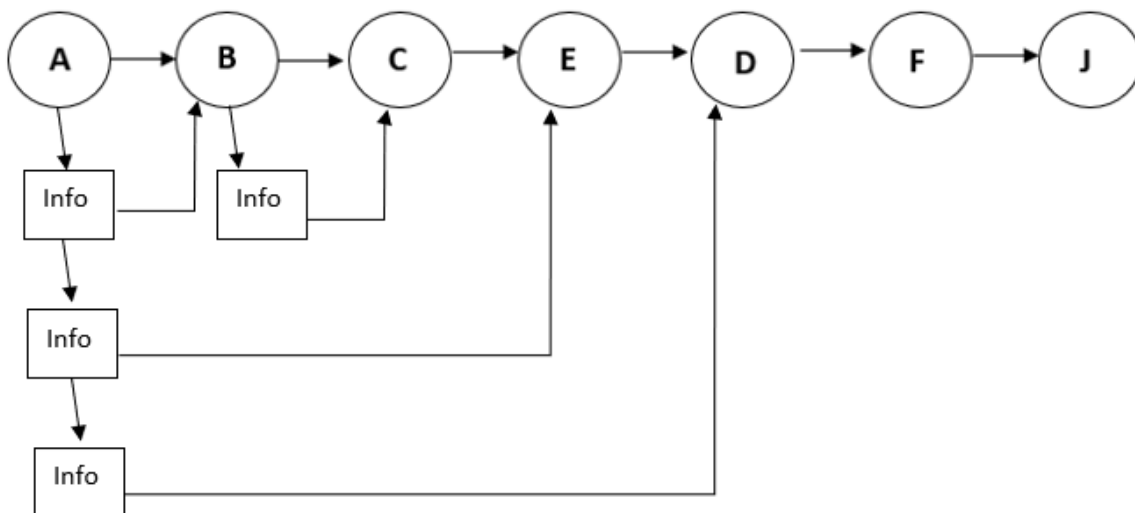
\_ La representación de un grafo más fuerte se da entre conexiones de algo que parecen listas enlazadas. Lo primero que pensamos es que los grafos son de orden  $N$ , no son binarios como vimos en árboles, y eso nos complica más. Nosotros deberíamos representar el siguiente grafo en alguna estructura de software, y para esto lo que hacemos es por un lado representar los nodos y por otro los datos.



\_ Para representar los nodos hacemos una especie de lista enlazada que nos permite conectar todos los nodos que hay para parar en el que queramos y así ir por ejemplo a un nodo que no esté conectado con nadie en el grafo. No hay orden. Aparece un nodo nuevo J por ejemplo y tenemos que conectar F con J, entonces todos los nodos que existan van a tener si o si conexión con los otros nodos. Los nodos contienen un monton de información y un puntero que me lleva a otro nodo.



\_ Ahora el nodo A va a tener otro puntero que nos indique como conectamos a este nodo con otro, pero esto no tiene nada que ver con el camino en el que recorremos el grafo. Conectamos el nodo A con el B indicando la información que hay entre el nodo A y B justamente. pero A además de conectarse con B se conecta con D y E, entonces deberíamos poner otro puntero de A con D, en donde este va a contener la información que hay entre ambos, y así también con E y el resto de nodos.





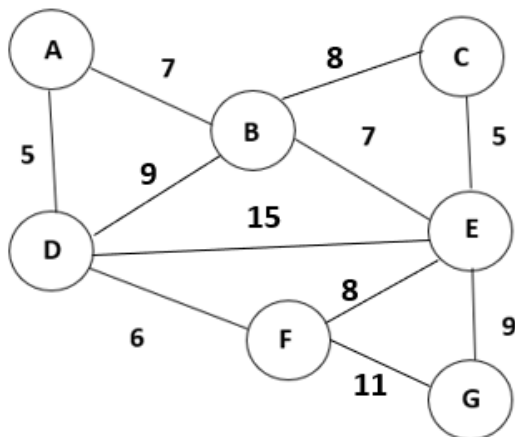
\_ Entonces de esta manera también armamos una lista enlazada entre los arcos de cada nodo. Cada nodo va a tener dos punteros, uno que conecta todos los nodos del grafo sin importar el orden, y otro puntero que conecta los arcos de cada nodo. Cada arco a su vez tiene dos punteros, uno entre los arcos y otro entre los nodos para poder establecer el vínculo

## Algoritmos aplicables a grafos

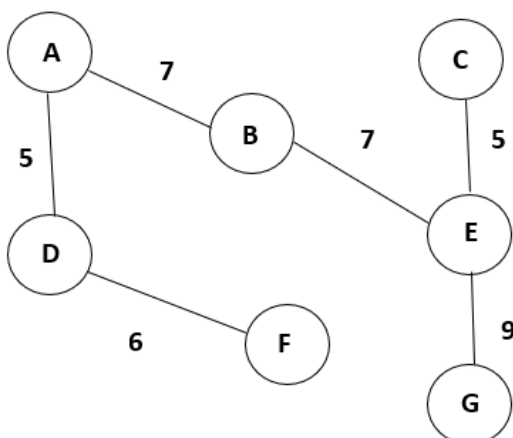
\_ Hay algunas cosas que se pueden hacer con los grafos, pero principalmente hay dos tipos de operaciones:

\_ La primera es encontrar el árbol de menor peso (árbol de expansión mínimo o árbol generador mínima) que este contenido en el interior de un grafo, para esto necesitamos trabajar con grafos valuados. Para este existen dos algoritmos:

- Algoritmo de Kruskal: lo primero que se hace es copiar la misma disposición de los vértices, a continuación se van dibujando los arcos desde el de menor peso en adelante siempre y cuando no formen ciclos, en nuestro grafo el A-D y C-E son de menor peso, seguimos uniendo pero en el caso de E-F que tiene un peso de 8 no lo hacemos porque estaríamos cerrando el camino generando un ciclo, directamente conectamos E-G y ya estarían conectados todos, por lo tanto hemos encontrado el árbol de menor peso que estaba en el grafo. Si sumamos los pesos nos va a dar el menor peso.



\_ Solución:



- Algoritmo de Prim: este funciona muy parecido al algoritmo de Dijkstra, es decir, comprendo Prim comprendo Dijkstra.

\_ Lo segundo que se puede hacer con grafos es encontrar el camino de menor coste entre dos vértices de un grafo, es decir, partimos de cualquier vértice y si es un grafo valuado usamos los siguientes algoritmos:

- Algoritmo de Dijkstra: que tiene la ventaja de que nos da todo el camino entre los dos vértices, pero tiene la desventaja que solo nos da el camino entre dos vértices y no entre todos los vértices y para eso debería aplicar una y otra vez este algoritmo.
- Algoritmo de Floyd/Warshall: nos va a dar todos los costes de todos los pesos de una matriz de adyacencia, pero no nos va a decir cuáles son los caminos. Este algoritmo es muy mecánico, utiliza la matriz de adyacencia y me da las menores distancias entre dos vértices.

\_ Existen otras aplicaciones como por ejemplo el problema del viajero. No es lo mismo encontrar la menor distancia entre dos vértices que recorrer todos los vértices del grafo encontrando el menor peso posible sin repetirlos, para esto no hay un algoritmo oficial sino que hay muchos planteamientos y uno de esos es este.

## Conclusión

\_ Cuando pensamos en un nodo pensamos en un monton de información dentro del mismo, es decir, no es un solo dato. Cuando intentemos meter esa información en un arreglo, estos se van a complicar porque deberíamos tener arreglos multidimensionales para cada información, pero con un nodo es mucho más fácil de administrar debido a sus propiedades y campos a diferencia de una arreglo. Vamos a encontrarle solución a los grafos con arreglos multidimensionales pero en casos muy simples, ahora cuando sea más complejo, la solución tiende a ser en otro tipo de construcción que no es un arreglo.

\_ El hecho de definir por separado los nodos y los arcos nos permite que cuando por ejemplo definimos un nodo X con la información que queramos independientemente de que esté conectado o no con alguien. Entonces el separar por un lado la información de un nodo y por otro la de los arcos, nos permite que uno exista sin que exista el otro, hablando de los nodos que podrían existir por sí mismos, en caso de los arcos no porque debe existir un nodo. Definimos el concepto de nodo con toda su estrategia y por otro el de arco con la suya diferente.

\_ En el caso del arco con un peso a almacenar, por ejemplo si un arco vale 6 que represente una distancia en km pero también nos interesa saber cuánto gas, agua y luz podemos transportar entonces en el arco deberíamos tener más datos por lo que una opción sería tener un arreglo para cada dato del arco respecto a los dos nodos, y esto sería algo así como entre el nodo A y B tenemos 6 km, de nuevo entre A y B tenemos tal

cantidad de gas y así. Pero otra estrategia mucho más flexible y dinámica es que el arco (A, B) tenga su propia información.

\_ Cada nodo puede contener mucha información en vez de un solo número, pero lo representamos con un número para poder simplificar el problema.

## Ordenamiento

\_ El concepto de conjunto ordenado de datos tiene un alto impacto en nuestra vida diaria. Es muy importante tener las cosas ordenadas, imaginémonos buscar el teléfono de una persona en una guía desordenada, esto sería muy difícil. Vamos a asociar dos cuestiones, ordenamiento y búsqueda, porque ordenamos si algo vamos a buscar, ya que cual sería el sentido de ordenar cosas si después no vamos a buscar algo, entonces hay un costo o esfuerzo de ordenar que tiene sentido en la medida que uno después va a buscar un monton de veces eso que ordeno. Ordenar algo que no vamos a buscar no tiene mucho sentido. Vamos a analizar cuánto cuesta ordenar y que beneficio nos trae después al momento de buscar.

\_ Un programador o diseñador debería pensar si le conviene o no ordenar los datos por algún criterio o dejarlos desordenados, y si en algún momento necesita buscar algo quizás insumir un poco más de tiempo en la búsqueda y no gastar todo el esfuerzo en tratar de mantener ordenado permanentemente un archivo que muy pocas veces se va a utilizar para buscar. Esta es una decisión a tomar en el momento de diseño de la base de datos.

\_ Entonces dada la relación entre búsqueda y ordenamiento, la primera cuestión es si debería o no ordenarse un archivo. Una vez decidido el programador si es conveniente o no ordenar, debe definir que ordenar y que método es más conveniente utilizar.

\_ La búsqueda es tan básica que no necesito tener todo ordenado para encontrar algo, pierdo tiempo de búsqueda en lugar de que si hubiese ordenado todo. Entonces siempre esta mezclado cuanto tiempo nos lleva ordenar y cuánto tiempo nos lleva buscar.

\_ Nosotros vamos a empezar a medir algo que hasta acá no hemos medido, que es la eficiencia en los procesos, vamos a pensar cuánto cuesta ordenar y buscar algo, pero vamos a empezar a medirlo en tiempos, en esfuerzo de tiempos, con lo cual se busca la mejor optimización posible. Entonces, vamos en dos secuencias, la primera es ordenar, cuanto es el esfuerzo y costo de ordenar algo, y la segunda es que beneficio nos trae si es que después vamos a buscar la información.

## Terminología

\_ Suponemos que tenemos un archivo de tamaño  $n$  que tiene un montón de  $n$  elementos que van desde  $r[0]$ ,  $r[1]$ ,  $r[2]$ , ...,  $r[n-1]$ . Cada uno de estos elementos se llaman registros, en estos registros podría haber por ejemplo nombres de alumnos, claves de la UCC, DNI, dirección, etc. Y ahora nos preguntamos por cual de estos atributos o elementos vamos a ordenar nuestro archivo. A cada registro  $r[i]$  le podemos asociar una llave o clave que se podría denominar  $k[i]$ , esta llave por lo general es un subcampo del archivo entero.

\_ Entonces lo que vamos a decir es que el archivo está ordenado de acuerdo a la clave  $k$ , si para todos los registros del archivo se cumple que si la posición  $i < j$ , la clave  $k[i] < k[j]$ . Esto nos dice que el archivo está ordenado de forma ascendente de menor a mayor, y las claves que podrían ser el DNI, la clave UCC, etc, o sea el archivo de tamaño  $n$  va a estar ordenado por una de las claves, es decir, si yo lo ordeno por DNI seguramente se nos va a desordenar por apellido y nombre, y si yo lo ordeno por apellido y nombre, lo más probable es que no va a estar ordenado por la clave UCC. Nosotros elegimos un atributo clave y ordenamos el archivo según ese atributo, entonces el archivo va a estar ordenado para esa clave  $k$  pero probablemente no va a estar ordenado para otras claves o atributos del registro.

\_ En el caso de un archivo, lo que vimos arriba es el criterio que nos da el orden, que dice que vamos a ordenarnos de acuerdo a una clave  $k$ , y según esa clave  $k$  elegida el archivo va a estar ordenado para esa clave si cumple que para todos los elementos  $i < j$  la clave  $k[i] < k[j]$ . El archivo va a estar ordenado para alguna clave pero seguramente no va a estar ordenado para otras claves o atributos del mismo registro del mismo archivo, ya que es muy difícil que esté ordenado por todos los criterios.

\_ Vamos a decir que existen los siguientes orden:

Ordenamiento interno: vamos a llamar ordenamiento interno si los registros que se están ordenando están en la memoria principal, es decir, si el ordenamiento se produce en la memoria principal.

Ordenamiento externo: vamos a llamar ordenamiento externo si los registros que se están ordenando están en un almacenamiento auxiliar, es decir, si el ordenamiento se produce en un disco duro por ejemplo.

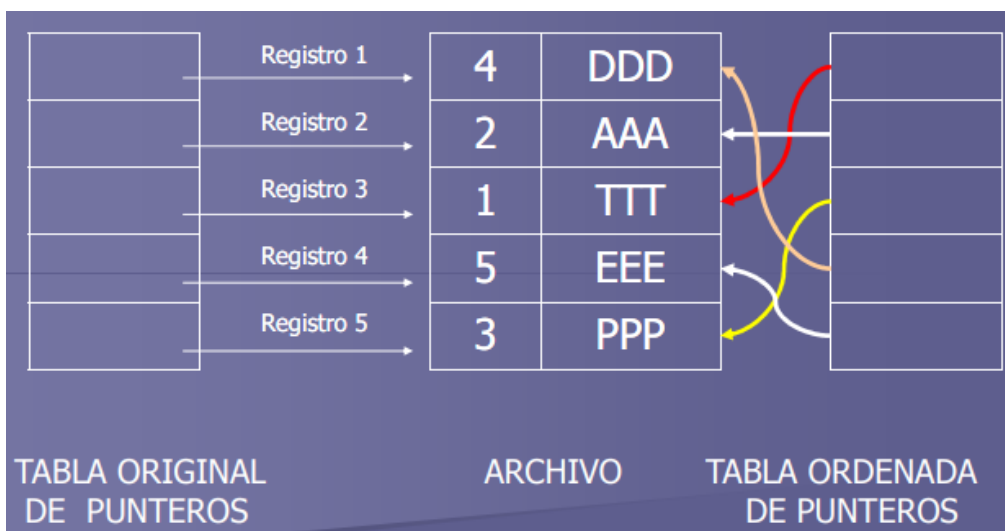
\_ Analizamos los siguientes casos:

Ordenamiento de registros reales: suponemos que tenemos un archivo que es el archivo original con sus elementos, y ahora lo queremos ordenar por alguno de sus atributos, por ejemplo podríamos ordenarlos por los números aunque no va a estar ordenado por el nombre o el texto que tengo porque no se puede ordenar por las dos cosas, pero tendríamos el archivo ordenado por los números.

Registro 1	4	DDD	1	TTT
Registro 2	2	AAA	2	AAA
Registro 3	1	TTT	3	PPP
Registro 4	5	EEE	4	DDD
Registro 5	3	PPP	5	EEE

ARCHIVO ORIGINAL      ARCHIVO ORDENADO

Ordenamiento usando una tabla auxiliar de punteros: podríamos decir, tenemos los registros, estos registros están desordenados físicamente en el archivo, pero podríamos generar una lista nueva o tabla ordenada de punteros que apuntan al archivo aunque el archivo mismo va a seguir desordenado, o sea el archivo se mantiene desordenado pero nosotros creamos una tabla que me hace dar la sensación de que si viéramos el archivo pero viéndolo desde esta tabla ordenada de punteros, nosotros nos podríamos creer que el archivo esta físicamente ordenado aunque no lo está. Las ventajas que esta produce son, primero que el registro podría ser muy grande y tener un monton de atributos (lo que queramos) y la tabla ordenada de punteros es muy pequeña donde solamente contiene el atributo clave k que yo elegí y el puntero que me lleva a la clave k, y donde acá nos creemos que el archivo esta ordenado por la clave aunque físicamente no lo está.



\_ Entonces solo deberíamos mantener ordenado el archivo de punteros, y esto me haría creer que el archivo esta físicamente ordenado por la clave k. Incluso podríamos tener un segundo archivo de punteros que apunte a los nombres y apellidos y creerme que esta ordenado por estos aunque físicamente tampoco lo está, solamente mantenemos

ordenado la tabla de punteros. Esto me permite generar diferentes tablas de punteros que apunten al archivo y que me hagan creer que el archivo esta ordenado por diferentes atributos cuando físicamente el archivo esta siempre desordenado.

\_ Esto es como operan la mayoría de los motores de bases de datos. La ventaja de esto es como el archivo físicamente se mantiene ordenado y las tablas de puntero son lo que yo debo ordenar, esto hace que yo gaste poco tiempo porque el archivo físico está en el disco duro desordenado y estas tablas de punteros ordenados pueden estar en la memoria principal donde nos encontramos con un archivo de punteros ordenado fácil y rápido de recorrer, fácil de reordenar si aparecieran nuevos elementos o registros en el archivo. Consumimos menos tiempo y esfuerzo, no movemos los datos en el disco duro, solamente movemos los datos en la memoria y solamente mantenemos ordenada una tabla de punteros que nos hace creer que el archivo esta ordenado.

## Consideraciones de eficiencia

\_ Existen varios métodos que pueden usarse para ordenar un archivo. Los elementos a considerar para la decisión del método a utilizar para el ordenamiento de un archivos son tres tipos de esfuerzos:

- Tiempo de programación, cuanto tiempo nos lleva programar el programa justamente que ordena.
- Tiempo de máquina para ejecutarlo.
- Memoria requerida para la ejecución, si gasta poca memoria quizás sea lerdo o difícil de programar, si es fácil de programar seguro anda rápido pero consume mucha memoria, etc.

\_ Cuando nosotros ordenamos y hacemos un programa de ordenamiento por ejemplo el método burbuja que es muy simple a nivel de programación, consume probablemente poca memoria pero el tiempo de ejecución de la computadora es muy lento y por ende ineficiente. Nosotros vamos a medir estas tres variables para después poder entender si los métodos son buenos o malos, que método conviene o no, y que en que instancias me conviene uno u otro.

\_ Si el archivo es pequeño las técnicas utilizadas, por más que yo mejore u optimice, disminuyen la eficiencia en valores pequeños, pero si el archivo es muy grande ya la cosa cambia. Así mismo si un proceso se correrá sólo una vez no tiene sentido un gran esfuerzo en técnicas de programación total lo ordenamos solo una vez, pero si vamos a estar ordenando constantemente mi archivo, o se me va a desordenar y lo debemos reordenar, habría que ver cuanto lleva este esfuerzo de ordenamiento y si este tiempo es muy lerdo empezamos a gastar un monton de esfuerzo de la computadora. Entonces ahora empezamos a medir eficiencia.

\_ El programador debe conocer diferentes técnicas de ordenamiento.

\_ En un proceso de computadora, uno de los elementos más lentos al momento de una función de un programa de un código, es cuando hacemos un if, si nosotros queremos comparar cosas (números, palabras, nombres, cosas) para ver cuál es mayor o menor, el tiempo que lleva el esfuerzo de hacer un if es una de las instancias con mayor esfuerzo y mayor consumo que impacta a una computadora. Entonces vamos a empezar a medir cuantos if tiene el código de un programa, porque la cantidad de if es lo que va a hacer que el código se vuelva más lento o más rápido.

\_ Estamos midiendo tiempos que demora el proceso, y la forma de medir el tiempo, es contando cuantos if, dependiendo de cuantos if haga va a ser más lento o más rápido, y contamos los if porque es la instrucción más pesada para la computadora. Pensamos que todo esto sucede en la memoria de la computadora y no en el disco duro porque sería más lento todavía.

## Método de burbuja

\_ Es quizás el método más sencillo de entender, aunque puede ser el menos eficiente. Supongamos que tenemos un archivo que es una lista de valores con: 25 57 48 37 12 92 86 33, y nosotros queremos ordenarlo ya que no está ordenado ni en orden ascendente ni descendente. Entonces el método consiste en pasar a través del archivo varias veces en forma secuencial. En cada paso se compara  $x[i]$  con  $x[i+1]$  e intercambiarlos en caso de que estén desordenados. Para el ejemplo comparamos 25 con 57, y si 25 es menor que 57 lo dejamos en su lugar, luego comparamos 57 con 48 y como 57 es más grande que 48 los intercambiamos de lugar, luego comparamos el 57 con el 37, y así voy comparando uno por uno cada uno de los elementos para irlos intercambiando. En cada iteración vamos viendo el cambio.

X[0]	Con	x[1]	(25 con 57)	No intercambio
X[1]	Con	x[2]	(57 con 48)	Intercambio
X[2]	Con	x[3]	(57 con 37)	Intercambio
X[3]	Con	x[4]	(57 con 12)	Intercambio
X[4]	Con	x[5]	(57 con 92)	No intercambio
X[5]	Con	x[6]	(92 con 86)	Intercambio
X[6]	Con	x[7]	(92 con 33)	Intercambio

Iteración 0	25	57	48	37	12	92	86	33
Iteración 1	25	48	37	12	57	86	33	92
Iteración 2	25	37	12	48	57	33	86	92
Iteración 3	25	12	37	48	33	57	86	92
Iteración 4	12	25	37	33	48	57	86	92
Iteración 5	12	25	33	37	48	57	86	92
Iteración 6	12	25	33	37	48	57	86	92
Iteración 7	12	25	33	37	48	57	86	92

\_ Antes de codificar el método, observemos que:

- Luego de la iteración 1, el número 92 quedo en su lugar.
- Luego de la iteración 2, el número 86 lo hizo.
- Al cabo de la iteración 5, la lista estaba ordenada.

\_ Si escribiéramos en código:

```
Funcion Burbuja ( Arreglo )
Largo = len( Arreglo );
Repetir desde pasada= 0 hasta Largo-1
    pasada++;
    Repetir desde j=0 hasta Largo-1
        j++;
        Si Arreglo[j] > Arreglo[j+1]
            comodin = Arreglo[j];
            Arreglo[j] = Arreglo[j+1];
            Arreglo[j+1] = comodin;
        FIN
    FIN
FIN
```

\_ Analizando el código tenemos que primero pasamos por parámetro un arreglo o lista. Determinamos la longitud del arreglo, ejemplo 50 elementos. Entramos en un bucle desde 0 hasta 49. Contamos las pasadas. En cada pasada vamos de vuelta de 0 a 49 en otro bucle, incrementado j. Y preguntamos si  $\text{Arreglo}[j] > \text{Arreglo}[j+1]$ , en caso de que si tengo que dar vuelta los números, usando una variable aux para hacer el intercambio, de lo contrario lo dejo en su lugar. Repetimos esta pasada hasta tener todos los elementos ordenados.

\_ Si tenemos un arreglo que ya está ordenado, vamos a estar recorriendo el arreglo con los ciclos sin hacer nada. Podríamos mejorarlo para que se dé cuenta de esto, y el peor de los casos seria de que este ordenado al revés porque debería mover todos los números todas las veces. El código mejorado seria:

```
Funcion Burbuja ( Arreglo )
Largo = len( Arreglo )
Repetir desde pasada= 0 hasta Largo-1 mientras Intercambios = TRUE
    pasada++;
    Intercambios = False;
    Repetir desde j=0 hasta Largo-pasada-1
        j++;
        Si Arreglo[j] > Arreglo[j+1]
            Intercambios = True;
            comodin = Arreglo[j];
```



```
    Arreglo[j] = Arreglo[j+1];  
    Arreglo[j+1] = comodin;  
FIN  
FIN  
FIN
```

\_ Analizando el código ahora creamos la variable pasada. Ponemos una bandera en false. A medida que damos más pasadas el largo se va haciendo más chico. Cuando entramos al ciclo cambiamos la bandera a true. Entonces si no intercambio nada es porque están todos ordenados. Preguntar por true o false, es más fácil que preguntar si es más grande o más chico.

\_ Las condiciones de repetir son costosas pero no tanto como un if, ya que comparar cuesta más que ver si j llegó a n por ejemplo.

\_ A nivel de programación este es el más fácil, por ende fácil de programar. Hay fórmulas matemáticas para medir el nivel de complejidad del código. Con respecto al consumo de memoria, tomando el código mejorado, este consume 3 o 4 variables de memoria más allá del arreglo, esto hace que podamos presumir que el consumo de memoria es poco, es decir, consume muy poca memoria ya que trabajamos con punteros pequeños y muy fáciles de mover en memoria, donde los vamos a ordenar, y no en el disco duro donde seguirán desordenados. Pero el defecto de este método es la eficiencia ya que insume mucho tiempo, consume mucho tiempo de ejecución, para medir el tiempo de ejecución lo que hacemos es contar cuantos if tiene mi proceso ya que es la instrucción más cara y más pesada del proceso.

- Tiempo de ejecución: Volviendo al código no mejorado, determinamos la cantidad de if que tiene el proceso, si el arreglo tiene una longitud de 1000, recorreremos el primer bucle 1000 veces y el segundo bucle 1000 veces también por ende el if más importante que es el de adentro se repite 1 millón de veces. El defecto del método de burbuja se nota cuando yo empiezo a hablar de muchos elementos, es decir, a medida que el arreglo aumenta su tamaño y empezamos a tener muchos elementos aparece el concepto de orden  $n^2$ , cuando hablamos de orden hablamos de un orden de consumo de tiempo de  $n^2$ , donde n es la cantidad de archivos y  $n^2$  es  $n^2$  ifs, ya que es la más crítica de todas las instrucciones. Entonces a medida que el arreglo, su longitud, sea más grande, la cantidad de elementos n del arreglo sea más grande, la cantidad de ifs va a crecer  $n^2$ , por lo tanto más grande sea el archivo más tiempo al cuadrado va a demorar el proceso en terminar.

\_ En el caso del código mejorado, si el arreglo está ordenado, tenemos chances de que termine rápido, pero si los elementos no están tan desordenados, esto puede llegar a demorar de vuelta la misma cantidad de longitud de pasadas. Entonces si nosotros creemos que el archivo está relativamente ordenado el método de burbuja mejorado

parece no ser tan malo, pero si creemos que el archivo esta desordenado, el método de burbuja es de orden  $n^2$  en el peor de los casos. Entonces el método burbuja mejorado demora  $n^2/2$  ifs y encima si llega a estar medio ordenado capaz que sale antes pero más allá de esto no me ahorro los  $n^2$  ifs.

\_ Este no es de los mejores métodos, es fácil de programar y consume poca memoria, pero es ineficiente y malo al cuadrado, ya que si medimos la eficiencia en tiempos este método no sirve.

Repaso del análisis de los códigos: hay Largo-1 pasadas y Largo-1 comparaciones. Entonces el total de comparaciones es  $(\text{Largo}-1) * (\text{Largo}-1) = \text{Largo}^2 - 2 \text{Largo} + 1$  que es del orden  $n^2$ . Con las consideraciones realizadas antes de iniciar podemos observar que el número de comparaciones (Largo-pasada) o sea  $(\text{Largo}-1) + (\text{Largo}-2) + \dots + 1 = (\text{Largo} * (\text{Largo}-1) + (\text{Largo}-1)) / 2$  con lo cual si bien el multiplicador es menor que antes, se sobrecarga el tiempo de trabajo con la variable Intercambio que se inicializa y revisa una vez por pasada. Además la función sigue siendo del orden  $n^2$ .

\_ Lo más interesante del método es que no requiere espacio adicional de almacenamiento de datos y que si la tabla está ordenada en una revisión de Largo-1 comparaciones se detecta. Estas son las virtudes.

## Método Quicksort

\_ Este es un método un poco mejor en cantidad de tiempos, es un ordenamiento rápido y uno de los más eficientes. Pasamos del peor método en eficiencia como el método burbuja a un método muy eficiente como el Quicksort. Este método es más difícil de programar, es recursivo así que va a consumir mucha memoria, pero tiene la ventaja de ser uno de los métodos más rápidos de ordenamiento que tenemos para un arreglo.

\_ La estrategia de solución es la siguiente, tomamos un arreglo de  $n$  elementos, elegimos un elemento cualquiera y lo que hacemos es poner a la izquierda todos los que sean más chicos que ese elemento y ponemos a la derecha todos los que sean más grandes, aunque sigan desordenados de los dos lados, o sea que cuando terminemos el primer ciclo lo único que logramos fue poner a la izquierda los más chicos y a la derecha los más grandes de ese número pero no van a estar ordenados todavía. Una vez logrado esto, llamamos recursivamente a la función, pero ahora con la mitad izquierda y la mitad derecha, donde en la mitad izquierda va a elegir un número y poner los más chicos en la izquierda y los más grandes a la derecha de este, luego se hace con el lado derecho anterior y así sucesivamente hasta que todo se ordene.

\_ Comparando este método con un árbol, como eficiencia en tiempo va a ganar el Quicksort, pero este tiene una desventaja enorme contra los árboles, ya que los árboles demoran casi lo mismo que el Quicksort y tienen como gran ventaja la memoria dinámica.

Quicksort es super eficiente pero necesita un arreglo, y este es de longitud fija. Esto convierte a los árboles en una super herramienta, ya que el ordenamiento en un árbol es casi tan eficiente como un Quicksort y encima en una estructura de datos dinámica.

\_ Volviendo al tema nuestro objetivo seria, sea un Arreglo y Largo el número de elementos del arreglo que debe ser ordenado. Elegir un elemento A de una posición específica en el arreglo (ej.  $A = \text{Arreglo}[0]$ ). Suponiendo que A está en la posición j entonces:

- Cada uno de los elementos en las posiciones de 0 a j-1 es menor o igual que A.
- Cada uno de los elementos en las posiciones j+1 a n-1 es mayor o igual que A.

\_ Con el mismo arreglo del método burbuja {25, 57, 48, 37, 12, 92, 86, 33}, elegimos un numero cualquiera por ejemplo el 25, luego tenemos que buscar todos los números menores a 25 ponerlos a la izquierda el mismo, y todos los que sean más grande los ponemos a la derecha. El programa que vamos a programar funciona de la siguiente forma, arrancamos en el número que elegimos en este caso 25 y preguntamos a los números siguientes si son más chicos, cuando encontremos un número que lo sea en este caso 12 deberíamos pasarlo del otro lado del 25, cuando encontremos un número que sea más grande que 25 debería estar del lado derecho, entonces arrancamos preguntando de abajo a la derecha y vamos a buscar números más grandes que 25, preguntamos y encontramos a 57 que es más grande y deberíamos buscar un número que sea más chico que 25 para traerlo al otro lado, entonces una vez terminada la secuencia preguntamos del lado derecho hacia el izquierdo si es más chico que 25 hasta que encontramos el 12, entonces intercambiamos 12 con 57 porque como 57 es más grande lo llevo a la derecha y al 12 como es más chico lo llevo a la izquierda. Una vez intercambiados seguimos revisando y nos encontramos con el 12 de nuevo y el proceso termino, lo único que hacemos es intercambiar el 12 con el 25 con lo cual quedaron todos los más chicos de 25 a la izquierda y todos los más grandes a la derecha pero absolutamente desordenados.

Abajo →	25	57	48	37	12	92	86	33	Arriba
	25	57	48	37	12	92	86	33	Arriba
	25	57	48	37	12	92	86	33	← Arriba
	25	57	48	37	12	92	86	33	← Arriba
	25	57	48	37	12	92	86	33	← Arriba
	25	57	48	37	12	92	86	33	← Arriba
	25	57	48	37	12	92	86	33	← Arriba
	25	12	48	37	57	92	86	33	← Arriba

	Abajo →			Arriba			
25	12	48	37	57	92	86	33
		Abajo		Arriba			
25	12	48	37	57	92	86	33
		Abajo		← Arriba			
25	12	48	37	57	92	86	33
		Abajo	← Arriba				
25	12	48	37	57	92	86	33
		← Arriba - Abajo					
25	12	48	37	57	92	86	33
	Arriba	Abajo					
25	12	48	37	57	92	86	33
	Arriba	Abajo					
12	25	48	37	57	92	86	33

\_ Entonces con un cursor vamos a la derecha buscando números más grandes y vamos con otro cursor hacia la izquierda buscando números más chicos, cuando estos dos cursores se encuentran en el lugar en el que se encuentran, en ese lugar es en el que debería estar el 25, y de ahí hacemos el intercambio para que queden los más chicos a la izquierda y los más grandes a la derecha. Este proceso sigue recursivamente para hacer lo mismo que se hizo pero con el arreglo reducido que nos quedó en este caso que sería 12 25, donde comparamos con todos los más grandes, eligiendo un numero cualquiera, y repetimos el método de forma recursiva hasta que ya no se pueda ordenar más. Vamos a gastar un poco más de tiempo en que el Quicksort debería elegir un buen pivot (en este caso mejor que 25), tratando de que nos quede la mitad de números a la derecha e izquierda. Si logramos esto el método es óptimo y es el mejor en velocidad.

\_ El proceso es difícil de entender, difícil de programar, y encima recursivo, pero es rápido. Acá tenemos algo de lo que sería el código:

Funcion Quicksort ( Arreglo, lb, ub )

Si lb <= ub FIN

Pivote = Arreglo[ lb ]

Arriba = ub;

Abajo = lb;

Repetir mientras Abajo < Arriba

Repetir mientras Arreglo[ abajo ] <= Pivote y abajo < ub

Abajo ++;

```

Repetir mientras Arreglo[ arriba ] > Pivote
    Arriba - -;
Si Abajo < Arriba n if
    temp = Arreglo[ abajo ];
    Arreglo[ abajo ] = Arreglo[ arriba];
    Arreglo[ arriba ] = temp;
Arreglo[ lb ] = Arreglo[ arriba ] ; n if
Arreglo[ arriba ] = Pivote;

```

```

Quicksort ( Arreglo, lb, arriba-1 );
Quicksort ( Arreglo, arriba+1, ub );

```

\_ Analizando el código, primero ingresamos un arreglo, pos inicial y pos final, de 0 a n. Si 0 es más chico o igual a n, termino el proceso, caso contrario hacemos todo el recorrido. Elegimos un pivote cualquiera (se puede optimizar buscando un mejor pivote). Primero vamos con un puntero avanzando, buscando números más grandes que el pivote. Buscamos para la izquierda **n if**. Buscamos todos los más chicos que el pivote, viniendo desde el otro lado. Buscamos para la derecha **n if**. Cuando los encuentro hago un intercambio, y de ahí volvemos a repetir. Repetimos hasta que abajo y arriba se crucen, y cuando esto pasa hacemos de nuevo un intercambio para poder asegurarnos que el pivote quede en el medio. Una vez dividido en dos, vuelvo a entrar a la función pero desde 1 hasta el lugar donde quedo ubicado el pivote. Y volvemos a entrar a la función desde la posición n+1 hasta el final del archivo. Tenemos dos llamadas de vuelta al Quicksort para que vuelva a hacer el mismo ciclo dentro del código.

\_ Lo que nos interesa de acá es que el programa es difícil de programar, si lo midiéramos en cuantos bucles tiene, cuantos for tiene, cuantas variables y encima recursivo, el programa es difícil de programar, y ya con ser recursivo todos sabemos que consume mucha memoria. Nosotros recorremos el archivo de izquierda preguntando si los números son más grandes que el pivote, y lo recorremos de derecha preguntando si los números son más chicos, cuando encontramos uno que sea más grande y otro que sea más chico que el pivote los intercambiamos, y volvemos a preguntar hasta que me cruzo, la cantidad de if que hicimos fue n (el tamaño del archivo). Si elegimos bien el pivote, la cantidad de if sería n/2 de izquierda y derecha, luego n/4 del sub lado izquierdo y derecho del lado izquierdo y lo mismo para el derecho y así, pero caso contrario no sería de esta forma sino que la suma de todos sería de vuelta n. Entonces tendríamos los **n if** hasta que todos estuvieran ordenados, esto tiene matemáticamente en el mejor de los casos una profundidad que se llama M que es igual al  $\log_2(n)$ , esto se da si elegimos perfectamente los pivotes, pero si los elijo muy mal esto podría llegar a ser n y demoraría lo mismo que el método burbuja, porque cada vez que tengo n if sería n \* n veces, sino en el mejor de los casos  $n * \log_2(n)$ . Para calcular el mejor pivote podríamos calcular el promedio del arreglo,

podría ser la media, mediana, etc, con alguna función matemática que no son tan lentas como hacer un if, para no gastar tanto tiempo en profundidad.

- Tiempo de ejecución: entonces si elegimos bien el pivote el tiempo de ejecución será  $n * \log_2 n$ . Si elegimos el peor pivote gastaríamos lo mismo que el burbuja, o sea  $n * n$ . Si eligiéramos los pivotes de forma excelente, el recorrido se iría dividiendo en dos mitades. Esto se parece a los árboles, ya que dependiendo el valor de la raíz el resto se va dividiendo en dos, pero además la virtud del árbol es que se puede elegir mucho más fácil quien va a ser la raíz.

\_ El método de Quicksort si lo miramos en extensión hacia delante, es un método que demora aproximadamente  $n + 2 * (n / 2) + 4 * (n / 4) + 8 * (n / 8) \dots$ , tantas veces  $n + n + n + n \dots$  ( m veces), es del orden m del  $\log_2 (n)$ , con lo cual este método es del orden  $n * \log_2 (n)$ . Esto es muy eficiente, más que  $n^2$ . Por ejemplo ordenar un arreglo de 1000 elementos mediante el método de Quicksort me llevaría aproximadamente 10000 if, y si lo hacemos con el método burbuja me llevaría 1 millón de if, el método burbuja optimizado me llevaría 500000 if. Este método favorece los tiempos de ordenamiento. Si los pivote no son bien elegidos vamos a tener el mismo problema en un árbol y un Quicksort, caso contrario la profundidad sería óptima y el tiempo de ordenamiento sería el mejor de todos. Así mismo por más que elija malos pivotes los tiempos de un algoritmo Quicksort son mucho mejor a los del método burbuja, al menos que todos los pivotes sean todas las veces mal elegidos donde en vez de parecerse a un árbol sería como una lista enlazada.

\_ El tiempo de procesamiento en unidades de segundo dependerá de la complejidad de cada computadora.

Resumen análisis código: esta rutina puede llamarse de manera recursiva, aunque por la cantidad de requerimiento de memoria no es conveniente para el caso de funciones de ordenamiento. Una técnica interesante para optimizar el proceso es hacer una recorrida por el arreglo calculando el valor promedio del arreglo, e ingresando este valor como elemento pivot para la función de ordenamiento. Supongamos que tenemos un archivo de tamaño  $n = 2^m$ . Si dividimos el archivo justo en mitades cada vez que encontramos el pivot, entonces haremos en la primer recorrida n comparaciones, luego haremos 2 veces  $n/2$  comparaciones y luego para 4 subarchivos  $n/4$  comparaciones. Así en cada recorrida se realizarán n comparaciones tantas veces como m.

$$n + 2 * (n / 2) + 4 * (n / 4) + 8 * (n / 8) \dots$$

$$n + n + n + n \dots ( m \text{ veces})$$

\_ De manera que la función es de orden  $n * m$ , y como  $m = \log_2(n)$ , se concluye que la función Quicksort es de orden  $n * \log_2(n)$ . Lo que la hace muy eficiente.

Repaso ordenamiento por árboles binarios: para medir la eficiencia debemos considerar el caso en que los datos ingresen ordenados, en ese caso se realizarán las siguientes comparaciones:

$$2 + 3 + \dots + n = n * (n + 1) / 2 \quad 2 + 3 + \dots + n = n * (n + 1) / 2 - 1$$

\_ O sea un proceso de orden  $n^2$ . En caso de que el árbol este con los datos balanceados, (caso óptimo) las comparaciones son del orden  $n * \log_2(n)$ . Además debemos considerar para recorrer el árbol la necesidad de punteros y luego la necesidad de una pila para almacenamiento del orden de recorrido que se lleva del árbol. El árbol es un poco más lento que el Quicksort porque cuando el Quicksort termina el arreglo esta ordenado, en cambio el árbol cuando termina debemos hacer un proceso más que es el de recorrerlo para poder cerrar la secuencia.

\_ Entonces Quicksort, consume tiempo de programación, es complejo, encima es recursivo, consume mucha memoria, pero la cantidad de if que requiere para resolver un problema es del orden  $n * \log_2(n)$ , y eso es mucho menor que  $n^2$ .

\_ De la misma forma en la que podemos encontrar buenos pivotes para el Quicksort, podemos encontrar también buenas estrategias de balanceo para que el árbol se mantenga balanceado. Como los árboles logran acomodarse en situaciones complejas, tienden al orden  $n * \log_2(n)$  también. La gran ventaja de un árbol con un Quicksort es que son dinámicos.

## Método Shellsort

\_ Es un método de ordenamiento por disminución del incremento, este método ordena subarchivos separados del archivo original. Se parte de un valor de incremento k, que es el paso que se da entre los elementos, entonces el método ordenará k subarchivos donde cada subarchivos contiene  $n/k$  elementos a ordenar. Luego de ordenar los k subarchivos por inserción simple, se vuelve a elegir un valor menor de k, se particiona el archivo el en k subarchivos hasta que k toma el valor 1, que es la última acción.

\_ En la definición de Joyanes este método se suele denominar también ordenación por inserción con incrementos decrecientes. Se considera que es una mejora del método de inserción directa. En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo, pero el algoritmo de Shell modifica los saltos contiguos por saltos de mayor tamaño y con ello consigue que la ordenación sea más rápida. Generalmente, se toma como salto inicial  $n/2$  (siendo n el número de elementos), luego en cada iteración se reduce el salto a la mitad, hasta que el salto es de tamaño 1.

\_ Los pasos a seguir por el algoritmo para una lista de n elementos, según Joyanes:

1. Se divide la lista original en  $n/2$  grupos de dos, considerando un incremento o salto entre los elementos de  $n/2$ .
2. Se clasifica cada grupo por separado, comparando las parejas de elementos y si no están ordenados se intercambian.
3. Se divide ahora la lista en la mitad de grupos ( $n/4$ ), con un salto entre los elementos también mitad ( $n/4$ ), y nuevamente se clasifica cada grupo por separado.
4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.
5. El algoritmo termina cuando el tamaño del salto es 1.

Ejemplo: haciendo el método desde el primer punto de vista, hacemos un ejemplo con el archivo {25, 57, 48, 37, 12, 92, 86, 33}. En este caso suponemos un  $K=5$ , entonces los subarchivos serían:

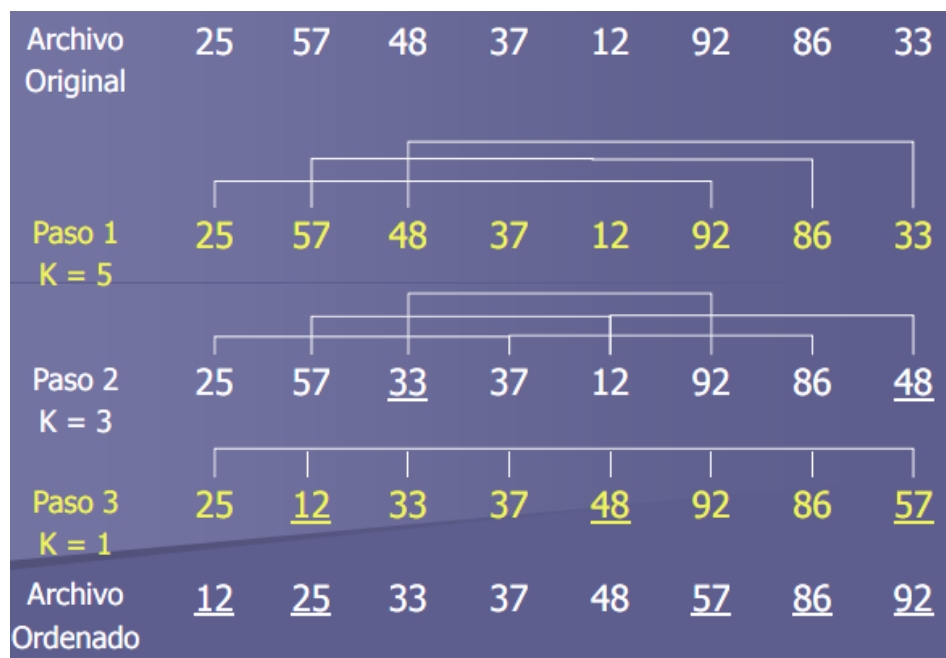
$x[0], x[5] - x[1], x[6] - x[2], x[7] - x[3] - x[4]$ , donde comparamos e intercambiamos en el caso de que uno sea mayor que el otro.

\_ Luego la segunda iteración con  $k = 3$ , y quedaría entonces:

$x[0], x[3], x[6] - x[1], x[4], x[7] - x[2], x[5]$ , seguimos haciendo intercambios.

\_ Por último si selecciona  $k = 1$ , se debe ordenar el siguiente archivo:

$x[0], x[1], x[2], x[3], x[4], x[5], x[6], x[7]$ , se compara uno por uno por que al estar casi ordenado esto lo hace con rapidez para terminar de ordenar el archivo.





\_ La versión en código sería:

ShellSort ( Arreglo, largo, incrementosdecrecientes, numincr )

paso = 0

Repetir mientras paso < numincr

    k = incrementosdecrecientes[ paso ]

    j = k

    Repetir j < largo

        a\_ubicar = x[ j ]

        temp = j - k

        Repetir mientras temp >= 0 y a\_ubicar < x[ temp ]

            x[ temp + k ] = x[ temp ]

            temp -= k

        x[ temp + k ] = a\_ubicar

        j ++

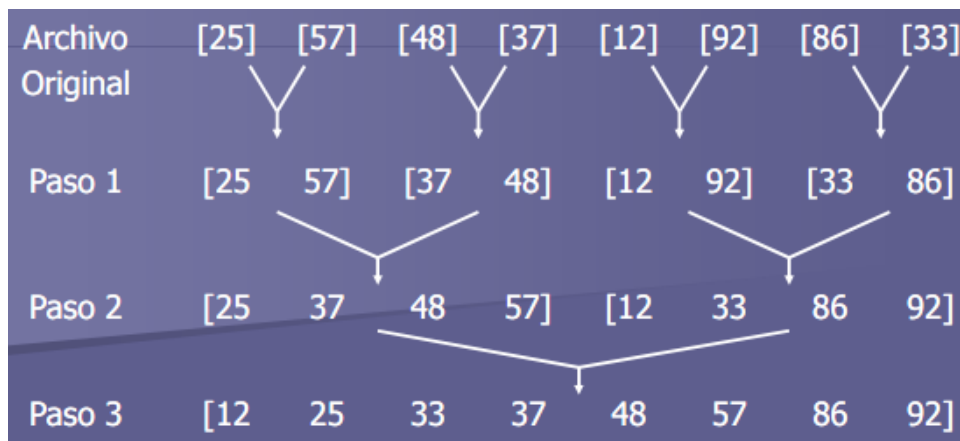
    incre ++

\_ Observar que si k = 1 entonces el ordenamiento es por inserción simple.

\_ La idea de ShellSort es muy simple. El ordenamiento por inserción simple es muy eficiente cuando el archivo está casi ordenado. Además cuando el archivo es pequeño a veces es más eficiente un ordenamiento del tipo  $n^2$  que un ordenamiento del tipo  $n \cdot \log(n)$ . Como el primer incremento de ShellSort es grande, los subarchivos son pequeños, de manera que el ordenamiento por inserción simple es bastante rápido. A medida que los incrementos decrecen el archivo se encuentra más ordenado y entonces los subarchivos que son grandes están ordenados y por ende el método de ordenamiento también es muy eficiente. El cálculo matemático es complejo, pero se ha demostrado que ShellSort se aproxima al orden  $n \cdot (\log n)^2$ , en general el método de ShellSort, se recomienda para archivos de tamaño moderado del orden de los mil elementos.

## Método de Intercalaciones

\_ Intercalación es el proceso de combinar 2 o más archivos ordenados en un tercer archivo ordenado. La idea que se nos ocurre consiste en dividir el archivo tomando elementos de a pares adyacentes, entonces tenemos  $n/2$  archivos de tamaño 2. Por ejemplo tomamos el archivo 25 57 48 37 12 92 86 33.



\_ El método de intercalación lo que hace son muchas intercalaciones, y asumimos que estas son más rápidas que los if, pero si el archivo es muy grande para ordenarlo, estamos suponiendo que tenemos todo el archivo en la memoria, pero si no logramos tener todo el archivo en la memoria porque es muy grande y realmente este estaría en el disco rígido, en este caso los métodos de intercalación moverían datos que están en el disco rígido, donde esos movimientos se convertirían en muy pesados porque acceder al disco rígido para leer o grabar cosas consume mucho tiempo. Entonces estos métodos que parecen muy rápidos, si el archivo es muy grande, se vuelve lento al momento de tener que mover los datos dentro del disco rígido. Este método no es recursivo.

\_ El ordenamiento por intercalación no hay más de  $n \cdot \log_2(n)$  pasos, y cada uno implica  $n$  o menos comparaciones. Así este método requiere no más de  $n \cdot \log_2(n)$  comparaciones. Esto se asemeja bastante al método de Quicksort, lo que pasa es que requiere muchas más intercalaciones de cambio de variables. Además se requiere más espacio de memoria para el armado de los subarchivos de trabajo. Ambos métodos implican la división de los archivos en partes, luego el ordenamiento de ambos y luego la unificación de ambos archivos.

## Resumen

\_ Estas son las consideraciones, para elegir que método nos puede servir en cada momento dependiendo del problema que tenemos que resolver. Existen un monton de métodos más ademas de estos.

Método	Esfuerzo de Programación	Tiempo de Ejecución	Consumo de memoria	Comentarios
Burbuja	Poco	$O(n^2)$	Bajo	Positivo si esta ordenado, es para archivos chicos.
Quicksort	Alto	$O(n \cdot \log_2(n))$	Alto	No detecta si el archivo esta ordenado, es solo para arreglos.
Árbol	Medio	$O(n \cdot \log_2(n))$ + recorrer	Alto	Flexible, rápido si se puede balancear el archivo.
Shellsort	Medio	$O(n \cdot (\log_2(n))^2)$	Bajo	Conviene para archivos hasta 1000 registros por ejemplo.
Intercalación	medio	$O(n \cdot \log_2(n))$ + intercambios	Medio	Hace muchas intercalaciones

# Búsqueda

\_ Vamos a empezar a pensar en cuanto tiempo nos insume buscar un dato en un archivo. Nosotros tenemos dos tareas, ordenar el archivo y luego buscar. La combinación del esfuerzo que me lleva ordenar más el esfuerzo que me lleva buscar produce un insumo de tiempo. Lo que nosotros vamos a medir es si vale la pena ordenar para después buscar, por ejemplo si buscamos un alumno en una lista y lo buscamos por única vez, mi tiempo de buscar es una sola vez, ahora si estamos buscando muchos alumnos en la lista hacemos un esfuerzo de ordenar pero después disminuyen todos mis intentos de búsqueda, entonces el esfuerzo de haber ordenado costo pero como después vamos a buscar un monton de veces y eso me va a simplificar el tiempo de búsqueda, el esfuerzo de ordenar vale la pena contra todos los intentos posteriores de buscar. No nos sirve de nada ordenar si después no vamos a buscar. El tiempo de búsqueda baja muchísimo si el archivo esta ordenado.

## Terminología

\_ Vamos a buscar en una tabla o archivo que es un grupo de elementos, donde cada elemento se denomina registro. Hay una llave asociada a cada registro, que se usa para diferenciar unos de otros, donde la asociación entre un registro y su llave puede ser simple o compleja. Nosotros usamos esa llave para buscar lo que queremos dependiendo del caso. En la forma más simple la llave está contenida dentro del registro en un tramo a una distancia específica del principio del mismo, esto se llama llave interna o incluida. En otros casos hay una tabla de llaves diferentes que incluye apuntadores a los registros, estas se llaman llaves externas y no son directamente el registro.

Llave primaria: para todo archivo existe por lo menos un conjunto de llaves que es único, que asegura que hay un numero distinto para cada uno de los atributos de la base de datos. Es una clave que asegura que el archivo tiene un conjunto de llaves único, o sea no hay repetidos. Es una clave que me permite identificar un elemento de otro sin que haya nunca dos repetidos. Por ejemplo la clave de la UCC es un dato que nos garantiza que no hay alumnos repetidos.

Llaves secundarias: además pueden los registros estar ordenados por otro conjunto de datos.

Algoritmo de búsqueda: es un algoritmo que acepta un argumento a y trata de encontrar un registro cuya llave sea a. El algoritmo puede dar como resultado un apuntador al registro encontrado, o un apuntador a registro nulo. Es decir, una función de búsqueda es un algoritmo al que yo le digo que busque tal persona o número, y me debe responder dos cosas, o me dice si está en tal ubicación o me dice no está.

Recuperación: llamamos así a la búsqueda exitosa.

Búsqueda e inserción: muchas veces al no encontrar el registro se desea que el algoritmo inserte una nueva llave con el valor indicado. Cuando buscamos un dato debemos construir una función que busca, cuando nosotros hablamos de principios de programación dijimos que nosotros llamamos a una función que tiene que hacer una sola cosa exclusiva, si queremos buscar a alguien esa función busca, si queremos insertar a alguien nuevo en la base de datos deberíamos llamar a una función que diga insertar(persona) y la agregamos. Cuando los archivos están ordenados buscar en alguna estructura de datos alguna persona y detectamos que no está, ese es el momento en el que lo insertamos en donde debería estar. La función de búsqueda puede hacer dos tareas en vez de una, puede buscar e insertar, y es conveniente inyectarlo en la misma función. La función que busque y si no llega a encontrar en el mismo momento inserte me sirve porque justo la inserción va en el lugar donde estaba buscando. Este es un caso particular donde una función puede hacer dos cosas a la vez.

\_ Entonces, las funciones de búsqueda pueden tener esta doble característica que es buscar e insertar, porque justo donde busque y no encuentro es en ese lugar donde te tengo que insertar.

Búsqueda interna y externa: si la tabla de búsqueda está contenida totalmente en memoria se llama búsqueda interna, si la misma está almacenada en memoria auxiliar se llama búsqueda externa. Vamos a usar la memoria para simplificar el análisis.

## Método de búsqueda secuencial

\_ Es la búsqueda uno a uno, por ejemplo suponemos que tenemos una base de datos con 1000 alumnos y queremos buscar a Santi, la cantidad de if que vamos a usar en promedio es 500 ( $n/2$ ). Si ordenamos el archivo gastábamos  $1000 * \log_2(1000)$  para ordenarlo, y después para buscar a alguien usamos otra estrategia para buscar. Pero si no ordenamos nada, nos ahorramos todo el tiempo de ordenar y si estamos buscando a Santi una sola vez, por más de que no ordenamos el archivo, en  $n/2$  if lo vamos a encontrar. Entonces si la única persona que buscamos es Santi quizá no vale la pena todo el esfuerzo de ordenar porque ya me lleva  $n * \log_2(n)$  ordenar en el mejor de los casos, y además debo buscarlo, donde todo es más tiempo que  $n/2$  en una búsqueda secuencial. Ahora si voy a buscar un monton de veces, lo que significa repetir  $n/2$  (tiempo de ejecución) un monton de veces, ahí me conviene ordenar para que ahora empiece a demorar mucho menos.

\_ Normalmente lo que nos pasa con la información, es que esta se ordena una vez pero se busca miles de veces. La forma más simple de búsqueda es la secuencial, esta búsqueda se puede utilizar sobre una tabla ordenada o no, o sobre una lista ligada.

\_ Si  $k$  es un arreglo de  $n$  llaves, y  $r$  es un arreglo de  $n$  registros, de manera tal que  $k(i)$  es la llave del registro  $r(i)$ , y si  $key$  es el argumento de búsqueda.

\_ A continuación vemos un código:

Para  $i$  que va de 0 hasta  $n$ , de a 1

Si  $key == k(i)$  entonces  $FIN(i)$

$Fin(-1)$

\_ Analizando el código voy desde el elemento 0 al  $n$  de uno en uno, preguntamos si es el que buscamos. Si lo encontró termina, sino sigue preguntando.

\_ El algoritmo revisa cada llave, si logra la coincidencia devuelve el apuntador  $i$ , sino obtiene coincidencia devuelve el valor  $-1$ . Podría suceder que busco a alguien en la base de datos, que no está, y si la base de datos tampoco está ordenada voy a demorar en ese caso  $n$  (1000 veces si tomamos el ejemplo anterior). También nos ayuda el hecho de tener la tabla ordenada para ahorrarnos y agilizar tiempo.

\_ Entonces la eficiencia del método de búsqueda, es  $O(n/2)$  para el caso de una búsqueda exitosa, y de  $O(n)$  en caso de una infructuosa. Para un archivo ordenado o no, en ambos casos el orden es  $O(n/2)$ . Si la búsqueda es secuencial parece no tener sentido que el archivo esté ordenado, pero la ventaja de que este ordenado es que va a dejar de preguntar cuando un dato que buscamos no esté en el archivo. Es normal que haya un grupo de registros que son más accedidos que otros, ejemplo facultad o infractores de ley. También es muy difícil conocer cuál va a ser el registro o grupo de registros más accedidos.

\_ Tenemos dos estrategias:

1)\_ Mover al frente: podría ser que al archivo no lo ordenemos para nada, y cuando busco a alguien automáticamente cuando lo encuentro me lo traigo al principio de la lista, donde solo hacemos un solo cambio, que es la persona que encontramos con la que está primero y tiro todos un lugar para abajo, entonces esta persona que buscamos la ponemos al inicio de la lista para que la próxima vez que la queramos buscar la encontremos más rápido, por ejemplo el historial de Google, para compartir archivos por Wpp aparecen las personas con más frecuencia y sino aparecen a continuación los últimos chats. Este método implica mover el último puntero localizado en una búsqueda exitosa al primer lugar.

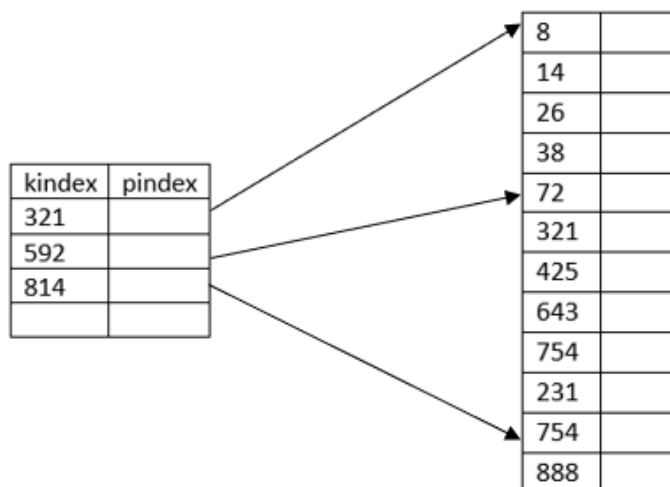
2)\_ Transposición: es por ejemplo busco a Santi y cuando lo encuentro no lo llevo al principio sino que lo subo un lugar o avanzo un casillero, es decir, intercambio un solo lugar con el que está arriba o delante, si volvemos a buscar a Santi vuelve a ir para arriba, y así sucesivamente cada vez que lo busquemos hasta que llegue al principio. Se ha demostrado que el método de transposición es más eficiente en un gran número de búsquedas en las que no cambie la distribución de probabilidad, en cambio el método de moverse al frente reacciona mejor ante los cambios de distribución de probabilidades.

\_ Estos dos métodos que vimos tienen la siguiente característica, que es que son métodos que aunque no ordenemos el archivo, movemos algunas variables cuando vamos

buscando y encontrando, y tengo chances de encontrarte muy rápido aunque no gaste el esfuerzo de ordenar. Estos métodos se utilizan en la vida real y hacen que más allá de que ordene o no el resto de una documentación por ejemplo, puedo usar transposición o mover al frente para ganar tiempo y encontrar lo que busco rápidamente sin tener que ir a buscar en el mismo lugar en el que está siempre. Esto ahorra tiempos de búsqueda sin gastar esfuerzos en ordenar.

\_ Si la tabla se encuentra ordenada la búsqueda secuencial no mejora mucho buscar o no de esta forma ya que el tiempo sigue siendo  $n/2$ . Pero de todas formas si la tabla esta ordenada (por ej. en orden ascendente) pueden usarse varias técnicas para mejorar la eficiencia y me ahorra los tiempos de buscar a alguien que no está en la tabla, pero en el caso de buscar a alguien que si esta demoro lo mismo estando ordenada o no la tabla. Entonces un caso simple es por ej. si el elemento a buscar no se encuentra en la tabla en el caso ordenado hacen falta  $n/2$  comparaciones para detectarlo, en cambio si la tabla está desordenada se necesitan  $n$  comparaciones. Si juntamos un grupo de peticiones de búsqueda a la tabla ordenada, y ordenamos las peticiones, la cantidad de comparaciones a la tabla se reduce considerablemente. Si la tabla de peticiones es más grande que la tabla de búsqueda, el método secuencial de búsqueda es el más eficiente de todos

Búsqueda secuencial indexada: suponemos que tenemos los números de las claves en un arreglo de 1000 personas, hacemos una búsqueda secuencial por ejemplo para el 409, demoro  $n/2$  if, a veces lo encuentro rápido otras veces no tan rápido. Esta búsqueda nos ahorra mucho tiempo porque entramos rápido en el índice que fabricamos, y una vez que encontramos nos metemos en el índice apropiado y ya no buscamos entre tantos elementos sino que entre menos, y en vez de demorar  $n/2$  vamos a demorar la cantidad de claves en el índice dividido dos, y después va a demorar la cantidad de elementos que haya entre clave y clave dividido dos. Por ejemplo si tenemos un índice de 10 y dentro de cada uno tenemos 100 claves, reduciendo a la mitad tendríamos 5 y 50 y en total 55. Si el índice nos parece grande podemos hacer un índice del índice y así. Esta búsqueda secuencial indexada que empieza a bajar mucho los tiempos se empieza a parecer a un árbol, y logrando de que el archivo fuera de longitud variable donde los índices también lo fueran tendríamos un árbol de  $n$  dimensiones que probablemente sea muy rápido para encontrar a alguien.



\_ Repasando esta entonces se requiere un adicional de memoria para almacenar una tabla auxiliar llamada INDEX. La cantidad de comparaciones se reduce de manera considerable. Además en caso de ser muy grande la tabla, y crecer la tabla INDEX, se puede adicionar una nueva tabla INDEX Secundario, que acceda sobre la tabla INDEX y luego esta sobre la tabla secuencial.

## Método de búsqueda binaria

\_ Hacemos mención al ejemplo de la guía telefónica. Donde los tiempos de búsqueda para encontrar a alguien se disminuyen a un orden de  $\log_2(n)$ .

\_ Esta búsqueda nos obliga si o si a que ordenemos, tenemos que gastar el tiempo de ordenar y una vez que ordenemos, la búsqueda binaria dice que si estamos buscando tal alumno por ejemplo no lo busques uno por uno en toda la secuencia, sino que podría entrar al medio y fijarte si está el alumno, sino esta se elige buscar en la mitad más chica o la mitad más grande, entonces entramos al medio y así sucesivamente hasta que encontrar a la persona o te das cuenta que no está. La búsqueda binaria no demora  $n/2$  sino que demora  $\log_2(n)$  if. Esta es mucho más rápida que la búsqueda secuencial. Se parece a la búsqueda indexada, se parece a un árbol binario porque los tiempos son aproximadamente iguales, siempre y cuando para que un árbol binario funcione bien debería estar bien balanceado, es decir, tener justo la misma cantidad de hijos en todas las ramas, deberíamos haber elegido muy bien que elemento va al medio para dividir en dos mitades del mismo tamaño, y que elemento va al medio de cada árbol hijo para dividir en dos árboles del mismo tamaño y así sucesivamente.

\_ El método más eficiente para buscar en una tabla secuencial sin usar tablas auxiliares, es el de búsqueda binaria. Este método reduce en un factor de 2 luego de cada comparación la cantidad de elementos a comparar. Este método se puede aplicar tanto para la tabla secuencial como para la tabla INDEX. Como dificultad podemos indicar que este método solo puede utilizarse para tablas organizadas como arreglos. Además en este tipo de tablas se hace muy dificultoso el manejo de inserciones y eliminaciones de elementos, lo cual hace muy poco común su uso. Es decir, es muy potente pero solo funciona en estructuras de tipo arreglo y ordenados.

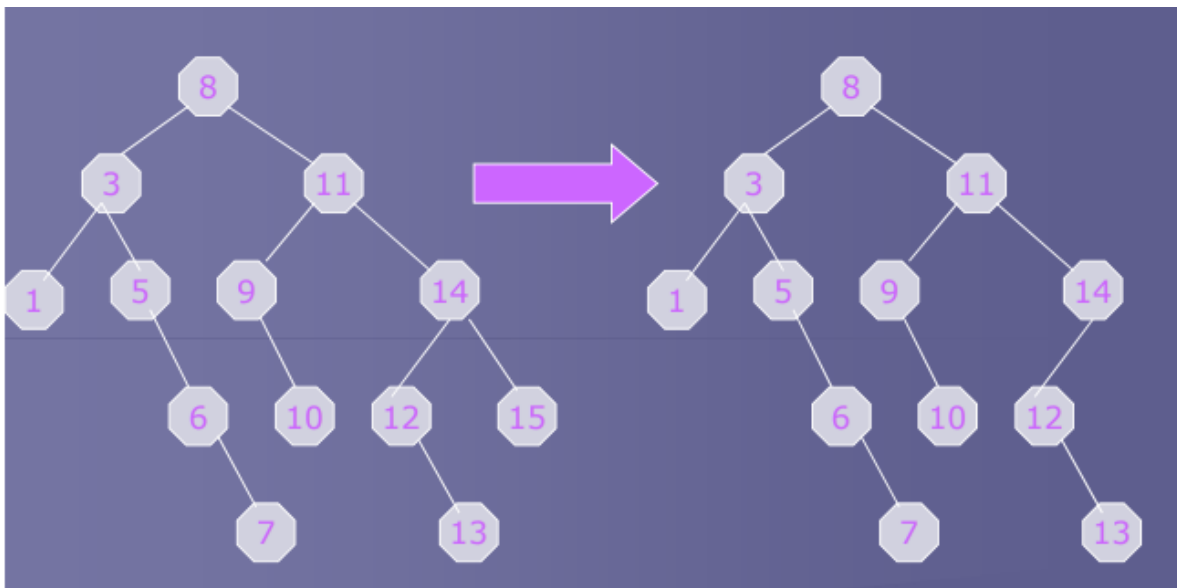
\_ Un árbol se parece a la búsqueda binaria si elegimos bien los elementos que están en el medio, si no elegimos bien los elementos del medio el árbol va a ser más ineficiente. Entonces si hacemos un árbol parejo el tiempo va a ser parecido al de una búsqueda binaria encima como el árbol es flexible puede crecer y achicarse rápidamente a diferencia de la búsqueda binaria que tiene un arreglo de longitud fija para poder entrar justo en la mitad. Los árboles a pesar de que cambien su tamaño van a seguir funcionando de manera ágil.

## Método de búsqueda en arboles

\_ Entonces como ventaja los árboles tienen que son estructuras dinámicas y pueden crecer o decrecer fácilmente, son óptimas para el ordenamiento si el árbol se mantiene en un formato balanceado y serían óptimas para la búsqueda porque tienen al tiempo  $\log_2(n)$  como la búsqueda binaria siempre y cuando el árbol este balanceado, la programación sería recursiva, el consumo de memoria sería alto, y logran buenos tiempos de ejecución.

Eliminación de una llave: suponemos que se elimina algún elemento del árbol, pero el árbol sigue estando ordenado y no se rompe, o sea si yo saco un nodo pero que no tiene hijos ni izquierdo ni derecho no hay problema de sacarlo y no pasa nada. Ahora suponemos que queremos sacar un nodo que no tiene hijos del lado izquierdo pero si tiene del lado derecho, lo que demos hacer es cambiar el puntero derecho del nodo padre del que queremos eliminar y que apunte al hijo derecho del que eliminamos en este caso para que el árbol siga ordenado. Si queremos sacar un nodo que tiene hijos de ambos lados buscamos en su hijo izquierdo al hijo derecho, o buscamos en su hijo derecho al hijo izquierdo para que cualquiera de esos dos lo reemplace. Cuando queremos reemplazar un nodo con otro debemos ver si el nodo reemplazante tiene o no un árbol abajo, ya que si es así debemos hacer toda la acción de nuevo de forma recursiva las veces que haga falta hasta que se acomoden todos los nodos.

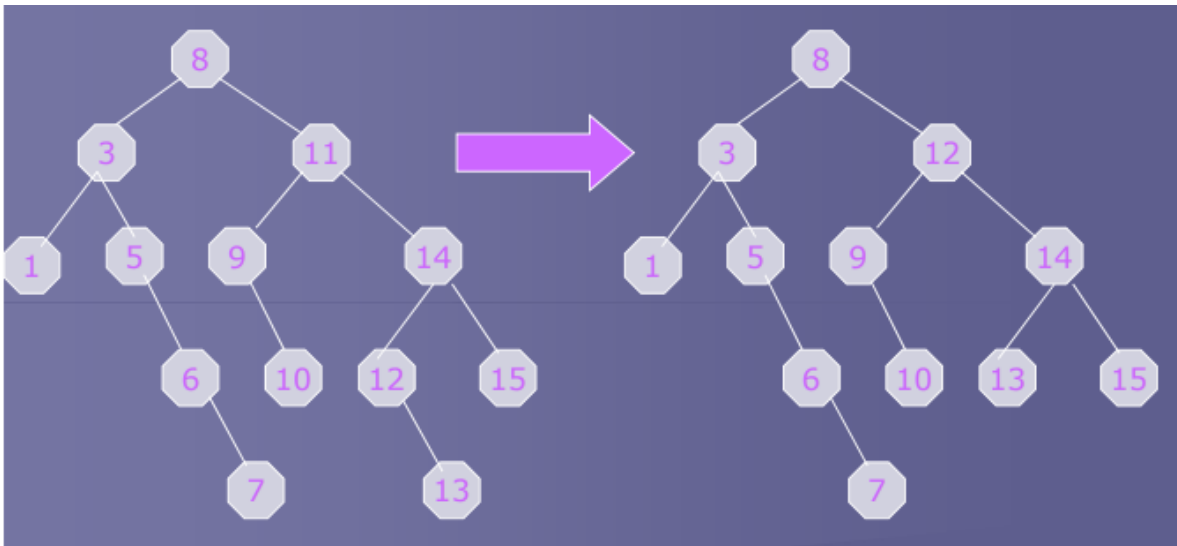
\_ A continuación eliminamos el nodo 15:



\_ Como no tiene ni hijos a la izquierda ni a la derecha, su eliminación no desordena el árbol.

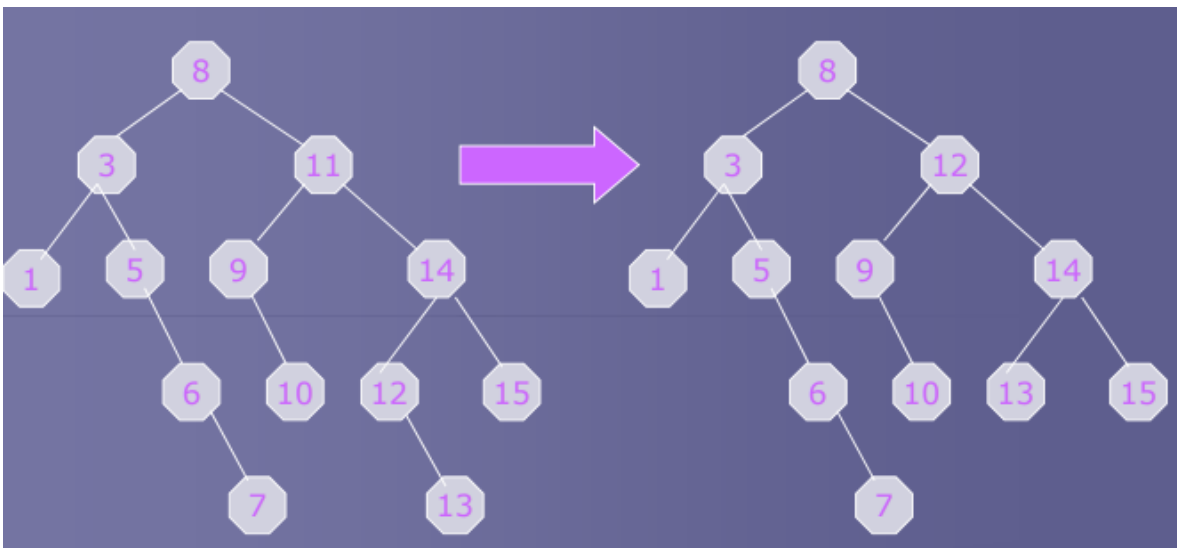


\_ A continuación eliminamos el nodo 5:



\_ En este caso, donde solo tiene hijos a la derecha y no a la izquierda, la eliminación se puede hacer automáticamente que el nodo raíz del que eliminamos apunte al hijo derecho de este último, sin producir desorden. Pasaría lo mismo si solo tuviese hijos a la izquierda.

\_ A continuación eliminamos el nodo 11:



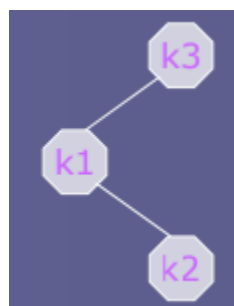
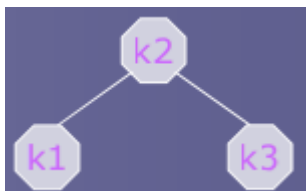
\_ En este caso se produce conflicto ya que tiene hijos de los dos lados. Para resolver esto tenemos dos opciones, la primera es o busco en el hijo derecho el hijo izquierdo o al revés en el hijo izquierdo el derecho para reemplazar el que eliminamos para mantener el árbol ordenado. En este caso puede ser tanto el 12 como el 10. Como pasamos el 12, también lo eliminamos del lugar donde estaba antes, por ende aplicamos una de las estrategias.

\_ Entonces para eliminar el nodo de un árbol tengo tres casos de riesgo, que no tenga hijos, que tenga hijo derecho, que tenga hijo de los dos lados. Programo cada una de las secuencias, y llamo de forma recursiva las veces que haga falta para poder acomodar las cosas. Esto permite que se dé de baja un dato sin que el árbol se desordene, con lo cual el tiempo de mantener ordenado un árbol no se complica tanto por más que yo elimine gente adentro del árbol. Con métodos bastante sencillos el árbol puede crecer y achicarse sin desordenarse, el único problema es que el árbol se puede desbalancear, y si esto ocurre empieza a tardar más tiempo en encontrar, si el árbol se me desbalancea en el peor de los casos tendríamos un árbol que parece una lista tardaría  $n/2$  en encontrar ya que se me transforma en una búsqueda secuencial. Pero si logramos tener el árbol balanceado vamos a lograr un tiempo de búsqueda de  $\log_2(n)$ .

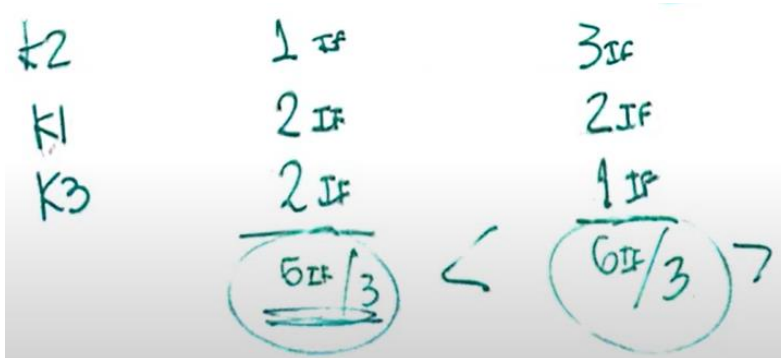
\_ Si logramos un árbol balanceado con poco esfuerzo, habremos encontrado un método que me garantiza soluciones óptimas de búsqueda para poder buscar en el menor tiempo posible. Los árboles son una estructura de datos que es muy útil porque permite ordenar y buscar muy rápido porque se puede mantener el balanceo, y si logramos tener un árbol balanceado tenemos la solución completa en una estructura de datos muy amigable para ordenar y buscar información; es óptimo el tiempo de ordenamiento y de búsqueda, es flexible la estructura, es potente, es rápida, es recursiva por lo que ocupa mucha memoria, pero gana por todos los otros lados.

Búsqueda: como vimos anteriormente, el ingresar los datos ordenadamente en un árbol es una operación del orden  $n \cdot \log_2(n)$ . Aquí debemos considerar dependencias como el orden en que se ingresan los datos. (si los datos están ordenados, el ingreso es más lento y la búsqueda es de tipo secuencial). Si los datos se ingresan en forma aleatoria, lo más probable que el árbol se encuentre balanceado.

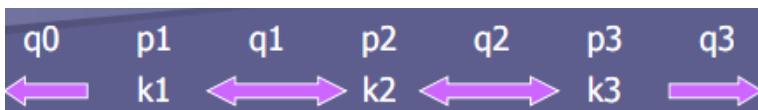
Eficiencia en búsqueda en árboles: Supongamos 2 árboles, en el primer caso recuperar  $k_3$ , implica 2 comparaciones, en el segundo sólo implica una comparación. Si nos preguntamos cual árbol es mejor y decimos el de la izquierda porque está balanceado. Entonces a continuación demostramos matemáticamente porque el árbol de la izquierda balanceado es mejor que el de la derecha. Lo importante a considerar, es cuantas veces se va a buscar  $k_1$ ,  $k_2$ , o  $k_3$ , y cuantas veces la búsqueda será infructuosa.



\_ Por ejemplo si buscamos K2, en el árbol de la izquierda vamos a tener un if y en el de la derecha tendremos tres if para encontrarlo. Hacemos lo mismo para K1 y K3 en los dos árboles, entonces lo que podemos hacer es calcular el promedio y nos da que es mas chico en cantidad de ifs el árbol izquierdo, por ende por eso es mejor.



\_ Ahora tenemos un análisis más preciso. Supongamos  $p_1, p_2$  y  $p_3$  las probabilidades de buscar a  $k_1, k_2, k_3$ , y  $q_0, q_1, q_2, q_3$  las probabilidades de buscar valores diferentes de  $k_1, k_2, k_3$  en los rangos correspondientes.



\_ Entonces sumamos las probabilidades  $p_1 + p_2 + p_3 + q_0 + q_1 + q_2 + q_3 = 1$ . El número esperado de comparaciones está dado por la probabilidad de acontecimiento multiplicada por las comparaciones a realizar para encontrar la llave. Entonces para árbol1:  $2p_1 + p_2 + 2p_3 + 2q_0 + 2q_1 + 2q_2 + 2q_3$ , en cambio para árbol2:  $2p_1 + 3p_2 + p_3 + 2q_0 + 3q_1 + 3q_2 + q_3$ . Este número esperado de comparaciones puede usarse como una medida de cuán “bueno” es un árbol para un conjunto dado de llaves y para un conjunto de probabilidades. Así para las probabilidades de la izq es mejor el árbol1, y para las de la derecha es mejor el árbol2.

$p_1 = 0,1$	$q_0 = 0,1$	$p_1 = 0,1$	$q_0 = 0,1$	$p_1 = 0,3$	$q_0 = 0,025$
$p_2 = 0,3$	$q_1 = 0,2$	$p_2 = 0,1$	$q_1 = 0,1$	$p_2 = 0,3$	$q_1 = 0,025$
$p_3 = 0,1$	$q_2 = 0,1$	$p_3 = 0,3$	$q_2 = 0,1$	$p_3 = 0,3$	$q_2 = 0,025$
	$q_3 = 0,1$		$q_3 = 0,2$		$q_3 = 0,025$
Valor esperado arb1 : 1,7		Valor esperado arb1 : 1,9		Valor esperado arb1 : 1,7	
Valor esperado arb2 : 2,4		Valor esperado arb2 : 1,8		Valor esperado arb2 : 2,03	

\_ Entonces un árbol balanceado es mejor que uno que no parece estar balanceado. Y esto se refleja además si las búsquedas son aleatorias.

Árbol de búsqueda óptimo: un árbol de búsqueda binaria que minimice el número esperado de comparaciones para un conjunto dado de llaves y probabilidades se llama óptimo. Es decir, se llama óptimo a un árbol que minimice el tiempo que demora en buscar. Fabricar y mantener árboles que sean perfectamente balanceados es muy caro y llevan mucho esfuerzo. El algoritmo más rápido conocido para producir un árbol de búsqueda binaria óptimo es  $O(n^2)$  en el caso general. Esto es muy costoso a menos que el árbol se mantenga sin cambiar durante un largo número de búsquedas. Sin embargo, aunque no existe un algoritmo eficiente para construir un árbol óptimo en el caso general, hay varios métodos de construcción de árboles cercanos a óptimos en tiempo de  $O(n)$ . Tenemos los siguientes métodos para tratar de mantener el árbol balanceado:

- Método de balanceo: tenemos que:  
 Si  $p(i) = a$  la probabilidad de buscar la llave  $k(i)$   
 Si  $q(i) = a$  la probabilidad de búsqueda infructuosa entre  $k(i-1)$  y  $k(i)$   
 Defino  $s(i, j) = q(i) + p(i+1) + \dots + q(j)$   
 \_ Este método intenta encontrar  $i$  que minimice el valor absoluto de  $s(0, i)$  o el absoluto de  $s(0, i-1) - s(i, n)$  y entonces establece  $k(i)$  como raíz del árbol, donde  $k(1)$  a  $k(i)$  donde  $k(1)$  a  $k(i-1)$  sean el subárbol izquierdo y  $k(i+1)$  a  $k(n)$  sean el subárbol derecho. Este proceso se aplica de manera recursiva para construir los subárboles izquierdo y derecho. Puede demostrarse que el orden de realizar este trabajo es del orden  $n \cdot \log(n)$ .
- Método exhaustivo: en lugar de construir el árbol de arriba hacia abajo como en el método anterior, este método construye el árbol de abajo hacia arriba. Este método utiliza una lista doblemente ligada, con 4 punteros (izq y der dobles), un valor llave y 3 valores de probabilidad (izq, der y de la llave).
- Método de división: este método contiene 2 llaves en cada nodo, una con el valor de éxito (si coincide la búsqueda culmina) y otra con el valor de división, que utiliza la mediana como llave del nodo más frecuente para subdividir los árboles. Este requiere un tiempo de construcción del orden de  $n \cdot \log(n)$ .

## Método de árboles balanceados

\_ El método más usado para mantener los árboles equilibrados es el de árboles balanceados en el que vamos a intentar es que la probabilidad de buscar una clave en la tabla sea la misma o aproximadamente la misma para todas. Si esto es así la búsqueda más eficiente se efectúa en un árbol binario balanceado. Tenemos las siguientes definiciones:



Altura o profundidad: de un árbol está dada por el nivel máximo de sus hojas.

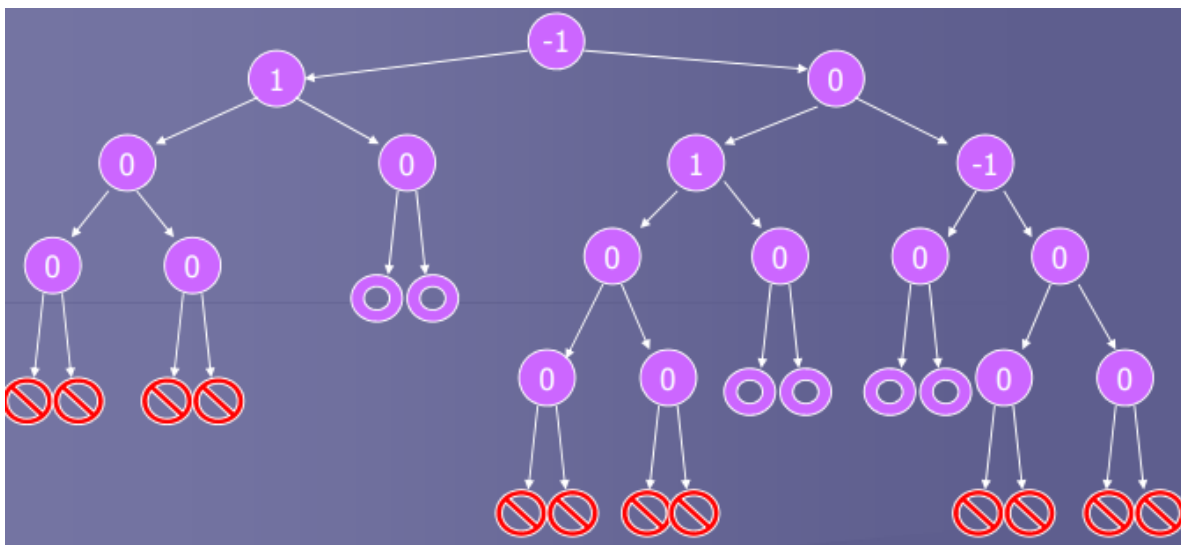
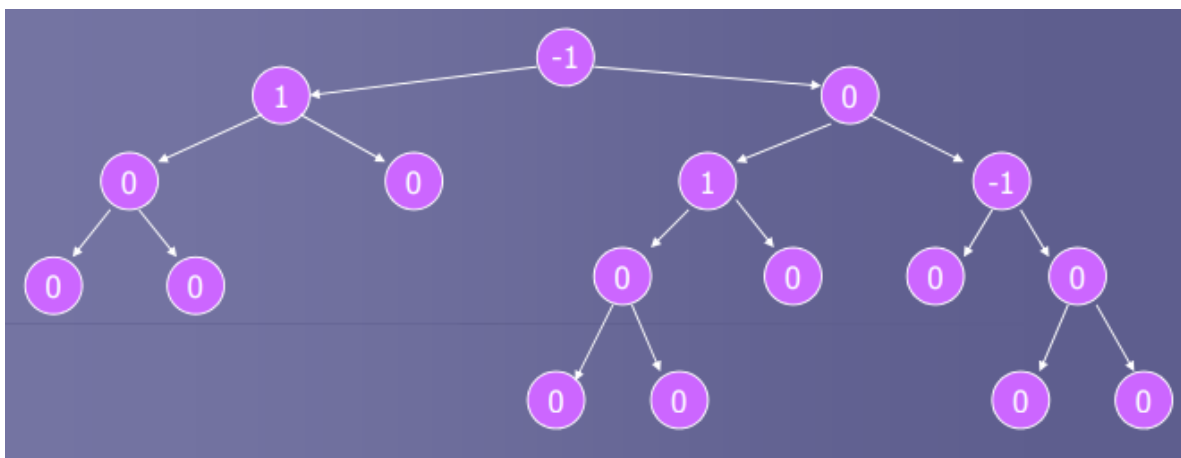
Altura de un árbol nulo: se define como -1.

Árbol binario balanceado: es un árbol en el cual las alturas de los dos subárboles izquierdo y derecho de todo nodo difieren a lo sumo en 1.

Balance de un nodo: se define como la altura del subárbol izquierdo menos la altura del subárbol derecho.

\_ Entonces, cada nodo de un árbol balanceado tiene balance igual a 1, 0 o -1 dependiendo de las alturas de los subárbol izquierdo o derecho.

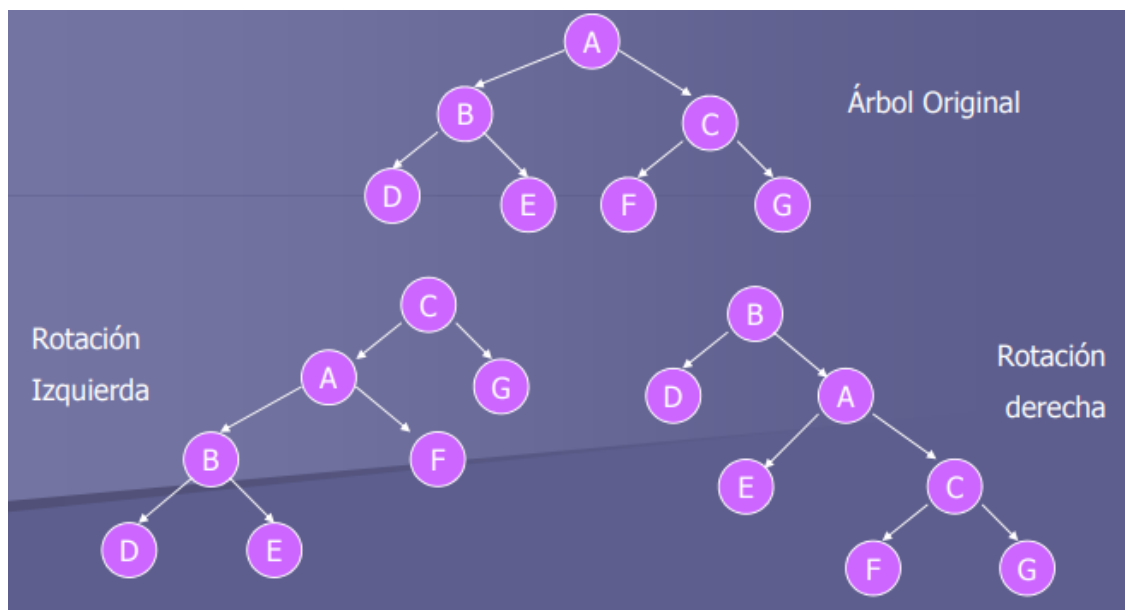
\_ Por ejemplo, como vemos en el grafico si ha este árbol balanceado agregamos un nuevo nodo (o elemento), puede que el árbol permanezca balanceado o que se desbalancee. En el paso siguiente los nodos con el símbolo  producen que el árbol se desbalancee, los indicados con  mantienen el árbol balanceado. Si se ponen en -2 el árbol se rompe.



\_ Cada vez que entra alguien en la base de datos, lo agregamos en el lugar que le toque, nos fijamos si el árbol se desbalanceó, si es así lo acomodamos y se da el alta. Esto nos garantiza que esa alta mantiene el árbol ordenado, cuando damos de baja, es decir, quitar

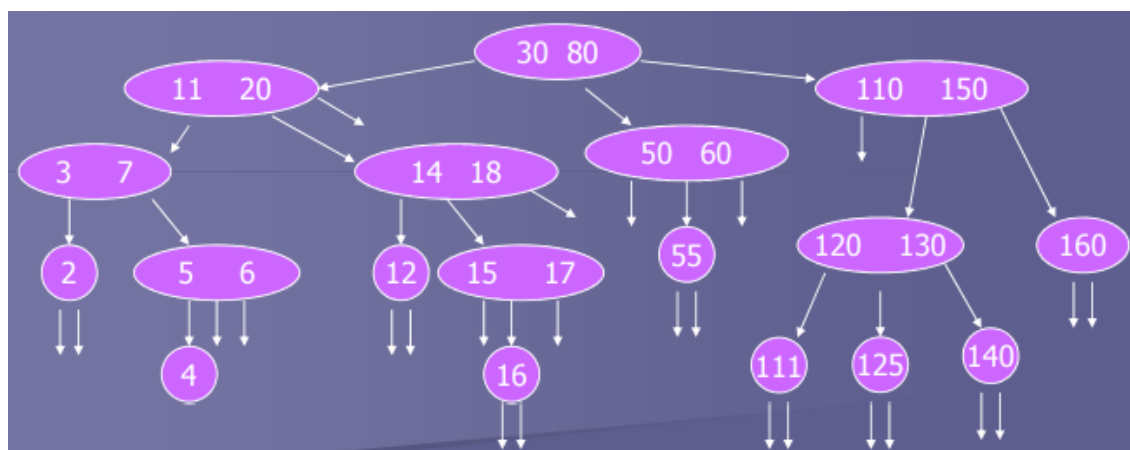
un elemento, pasaría lo mismo, y se fija si el árbol esta balanceado o no, y si no se lo balancea.

Técnica de rotación: para mantener los árboles balanceados al agregar nuevos nodos, la técnica a manejar es la de rotación derecha e izquierda. Esta técnica permite que un árbol que se encuentra balanceado y que al ingresar un nuevo nodo se va a desbalancear, el realizar una rotación izquierda o una derecha o ambas, pueda mantenerse balanceado. En la práctica, los árboles de búsqueda binaria balanceados se comportan mejor, produciendo tiempos de búsqueda de orden  $\log_2 n$  para valores grandes de  $n$ . Además se requiere una rotación en el 46,5% de las inserciones. El algoritmo para eliminar un nodo de un árbol balanceado es más complejo, y puede requerir una rotación doble o simple en cada nivel del árbol.



### Árboles multivias

\_ Un árbol de búsqueda de accesos múltiples de orden  $N$ , es un árbol general en el cual cada nodo tiene  $n$  o menos subárboles y contiene una llave menos que la cantidad de subárboles. Uno o más subárboles de un nodo pueden estar vacíos.



## Árboles B

\_ En los árboles B la técnica de inserción es más compleja, aunque esto se compensa con el hecho de crear árboles balanceados, de manera de reducir el número de nodos acusados para encontrar la llave es menor. Un árbol de búsquedas multivías balanceado de orden  $n$  en el cual cada nodo raíz contiene al menos  $n/2$  llaves se llama árbol B de orden  $n$ . Para mantener el árbol balanceado se intenta que los nodos de cada subárbol mantengan un número similar de llaves para cada subnivel, de manera que cuando el árbol tiende a desbalancearse se deben subdividir los nodos, y realizar rotaciones izquierdas y derechas de los elementos para mantener el árbol balanceado.

## Resumen

\_ Cualquier estrategia que elijamos, vamos a intentar que los esfuerzos sean los menores posibles y la más barata de todas. Siempre hay costos, tanto para ordenar como para buscar. Sino voy a buscar nunca o muy pocas veces no nos conviene ordenar porque es caro, y si es por una búsqueda simple quizás nos convenga que sea secuencial, antes de gastar todo el esfuerzo en un método complicado.

Método	Esfuerzo de Programación	Tiempo de Ejecución	Consumo de memoria	Comentarios
Secuencial	Bajo	$O(n/2)$	Bajo	Forma más simple, sobre tabla o lista ordenada o no. Ineficiente. Si esta desordenado, el orden será $O(n)$ .
Secuencial ordenado	Bajo	$O(n/2)$	Bajo	Como el arreglo ordenado, el orden es $O(n/2)$ .
Secuencial indexado	Bajo	$O(n/2)$	Bajo	Logramos un poco más de afinidad.
Binaria	Alto	$O(\log_2(n))$	Alto	Recursivo y lo combinamos con los árboles binarios.