Algoritmos y Estructuras de Datos

dreams_eater

April 10, 2012

Abstract

El Documento es tan solo una parte de los temas que me dictaron en clase, de Algoritmos y Estructuras de Datos, en algunos lados (Algoritmos y Estructuras de Datos 2). Por lo que el verdadero autor son mis profesores, foreros y los autores de los libros donde copie descaradamente imágenes.

Contents

Part I
Algoritmos y Crecimiento de las Funciones.

1 Algoritmos.

Cuando ejecutamos un programa sobre una gran cantidad de datos, debemos estar seguros que el programa termina en plazo razonable. Esto es independiente del lenguaje de programación utilizado, compilador/interprete/maquina_virtual utilizada y la metodología utilizada (si es procedimental u orientada a objetos).

¿Que es un Algoritmo? A esta altura sabemos por intuición que es, pero hay que decirlo.

Algoritmo: Un algoritmo es aquel procedimiento computacional bien definido, que toma o setea valores como entrada y produce un valor como salida. De este modo **es una secuencia de pasos computacionales** que transforman una entrada en una salida. También se puede ver como **una herramienta para resolver un problema computacional bien definido**. La declaración del problema especifíca, en general, los términos que se desean para la relación Entrada/salida. Los algoritmos describen el procedimiento computacional específico para lograr que se relacionen la Entrada con la Salida.

El algoritmo tiene propiedades que lo hacen ver mejor/peor según el problema y circunstancia.

¿Que es un Problema? Es la definición de un objetivo de llevar una circunstancia actual a otra. **Por ejemplo:** La ordenación es un problema si se puede definir formalmente el objetivo de llevar la entrada a una salida deseada.

Definición formal del problema de ordenación:

Entrada: Una secuencia de n números $(a_1, a_2, ..., a_n)$.

Salida: Una permutación (reordenación) $(a'_1, a'_2, ..., a'_n)$ de la secuencia de entrada.

Tal que: $a'_1 \leq a'_2 \leq ..., \leq a'_n$.

Si el input es (31; 41; 59; 26; 41; 58), el algoritmo retorna (26; 31; 41; 41; 58; 59).

Esa entrada en especial, para el problema, se llama **instancia del problema de ordenación**.

Se llama **instancia de un problema** a una entrada particular del problema que satisface cualquier obligación impuesta en la declaración del problema.

Un **algoritmo es correcto,** si para toda instancia del problema, termina con la respuesta correcta como salida. El algoritmo correcto resuelve el problema computacional.

Un **algoritmo es incorrecto, si** no termina nunca para algunas instancias de entrada, o si terminar con un valor incorrecto como respuesta para algunas instancias de entrada.

Contrario a lo que usted espera, **un algoritmo incorrecto puede ser util, si controlamos su taza de error.**

Suponga que tiene la computadora con velocidad infinita y la memoria infinita. ¿Para que perder el tiempo leyendo esto?

Esto le sirve para demostrar que su método de solución termina y que además el resultado es correcto. Si posee la computadora con rapidez infinita cualquier método correcto basta. Probablemente quiera estar en los margenes de "las buenas practicas de la ingeniería del software" (diseñado, implementado y documentado, y cobrado por el trabajo realizado), pero implementando el método más sencillo casi siempre. Las computadoras pueden ser rápidas, pero no infinitamente rápidas. La memoria puede ser mucha, pero no infinita. El tiempo de computo es por lo tanto una limitación de la fuente, y ocupa espacio en memoria. Programe sabiamente para que los algoritmos sean eficientes en términos de tiempo y espacio.

2 Análisis de los Algoritmos.

Sabemos que nuestro algoritmo funciona. El Análisis de los Algoritmos es determinar los recursos que consume.

Los recursos que consume los algoritmos son:

- Tiempo = Tiempo usado para resolver el problema.
- Espacio = Cantidad de memoria utilizada para resolver el problema.

El tiempo y el espacio están en función del tamaño del problema, llamado también tamaño de entrada . El Análisis de los Algoritmos se concentra en el tiempo.

El tamaño de entrada(..es..) depende del tipo de problema que se estudie:

- Es el numero de elementos de entrada (por ejemplo en la ordenación).
- Es la tupla de variables, donde la cantidad de cálculos a realizar esta en función del valor de cada variable.

¿Cómo es el análisis del tiempo?

Análisis de los Algoritmos tiene que ser independiente de la tecnología usada. A lo que me refiero es que no vamos a decir: Sobre un pentium M, con X código en C, tardamos 15 minutos en clasificar un archivo de 1 MegaByte sobre una Tabla Hash de 65mil entradas.

Ser independiente de la tecnología usada porque:

- **Las instrucciones por segundo de los microprocesadores son diferentes.** Por eso no se puede medir en función con el tiempo que tarda. La medición **Wall time,** marca el tiempo transcurrido en el reloj de pared y se emplea solo para testing de requisitos no funcionales, etc.
- **Los microprocesadores tienen sets de instrucciones diferentes, arquitecturas diferentes, compiladores diferentes.** Con esto descartamos la cantidad de instrucciones realizadas en asembler.

Se usa un análisis matemático asintótico sobre el crecimiento del algoritmo.

El running time de un algoritmo es el numero de operaciones primitivas o pasos que se van a ejecutar. El running time depende de la magnitud del tamaño de la entrada. Este análisis nos permite predecir cuanto tardara nuestro algoritmo para un tamaño de entrada mayor, al conocer el factor de crecimiento del running time.

De esta forma podemos comparar la relativa performance de algoritmos alternativos, en caso de tenerlos y poder elegir en base a un poderoso fundamento, porque el running time depende de la eficiencia del algoritmo.

No se puede mejorar el running time de un algoritmo en la fase de implementación, sin cambiar al algoritmo. La filosofía de la materia dice: *Mejore el algoritmo, no el código.*

3 Crecimiento.

Consta de dos partes: Calculo del running time del algoritmo y establecer un limite para poder usarlo para la predicción factor de crecimiento del tiempo del algoritmo.

El calculo del running time es el conteo de pasos elementales, tomando cada paso como de tiempo constante y que vale el mismo tiempo cada paso. En tras palabras, podemos representar la cantidad de pasos que realiza un algoritmo en función del tamaño de la entrada. En la literatura es la función T(N).

3.1 Notación Asintótica

El calculo del running time no da una expreción matemática que nos interese.

Las causas son:

- Los coeficientes de los términos no se conservan al cambiar la maquina ejecutora.
- Generalmente no se trabaja para tamaños de problema pequeños, por lo que los coeficientes de los términos no son importantes, tampoco los términos no dominantes.
- Para un tamaño de entrada suficientemente grande, es el que más repercute en el valor final

El **índice de crecimiento** de las funciones nos indica quien converge al infinito de forma más rápida, en otras palabras establece un orden relativo entre las funciones, mediante la comparación de terminos dominantes.

El término dominante de la función, es el índice de crecimiento, los términos inferiores al dominante se ignoran, la constante multiplicativa del termino dominante también se ignora. Hacemos de cuenta que los términos inferiores valen cero y la constante multiplicativa del término dominante vale uno.

Algunos ordenes de crecimiento son:

Termino Dominnte	Crecimiento
1	Constante
$\log N$	Logaritmica
$\log^2 N$	Logaritmica al cuadrado
N	Lineal
$N \log N$	
N^2	Cuadratica
N^3	Cubica
2^N	Exponencial

3.2 Ejemplos de calculo de T(N)

Supongamos que en un lenguaje x tenemos:

for j=1 to n:
...a=a+1;
¿Cual es la función T(N) de nuestro corto algoritmo?
Si c_1 es lo que tarda *for j=1 to n:* (comprobación e incremento e incremento solo)
y c_2 es lo que tarda *a=a+1;*
Como *for j=1 to n:* se ejecutara n+1 veces, y
a=a+1; se ejecutara n veces, Tenemos:
$T(N) = c_1(N+1) + c_2N = (c_1 + c_2)N + c_1$
¿Cual es el orden de crecimiento de nuestro corto algoritmo?
Los términos no dominantes valen 0 y las constantes valen 1.
Reformulando T(n), queda:
$T(N) = N$: el orden de crecimiento es Lineal.

Supongamos que en un lenguaje x tenemos:

for i=1 to n:
...for j=1 to n:
.....a=a+1;

¿Cual es la función T(n) de nuestro corto algoritmo?
Si c_1 es lo que tarda *for i=1 to n:* (comprobación e incremento e incremento solo)
, c_2 es lo que tarda *for j=1 to n:* (comprobación e incremento e incremento solo)
y c_3 es lo que tarda *a=a+1;*
for i=1 to n: se ejecutara n+1 veces,
for j=1 to n: se ejecutara (n+1)*n veces,
a=a+1; se ejecutara n*n veces
por lo que $T(N) = c_1(N+1) + c_2(N+1)N + c_3NN = (c_3+c_2)N^2 + (c_1+c_2)N + c_1$
¿Cual es el orden de crecimiento de nuestro corto algoritmo?
Los términos no dominantes valen 0 y las constantes valen 1.
Reformulando T(n), queda $T(N) = N^2$, el orden de crecimiento es cuadrático.
Supongamos que en un lenguaje x tenemos:
for i=1 to n:
...for j=i to n:
.....a=a+1;
¿Cual es la función T(n) de nuestro corto algoritmo?
Si c_1 es lo que tarda *for i=1 to n:* (comprobación e incremento e incremento solo) ,
$c_2$es lo que tarda for *j=i to n:* (comprobación e incremento e incremento solo) y
$c_3$es lo que tarda *a=a+1;*
for i=1 to n: se ejecutara n+1 veces,
for j=i to n: se ejecutara $\frac{(n+1)n}{2}$ veces,
a=a+1; se ejecutara $1 + n + \frac{(n+1)n}{2}$ veces
por lo que $T(N) = (c_1+c_2) + (c_1 + \frac{3}{2}c_2 + \frac{1}{2}c_3)N + \frac{1}{2}(c_2+c_3)N^2$
¿Cual es el orden de crecimiento de nuestro corto algoritmo?
Los términos no dominantes valen 0 y las constantes valen 1.
Reformulando T(n), queda $T(N) = N^2$, el orden de crecimiento es cuadrático!
-fin de ejemplos- :-)

3.3 Notación asintótica (eficiencia asintótica)

Recapitulando la matemática, teníamos algo llamado asíntotas, las cuales nos imponían un limite horizontal o vertical en el plano cartesiano. Del cual nos acercábamos mucho pero nunca lo tocábamos. Aquí la asíntota es una función, de esta manera logramos un estudio de la eficiencia mediante asíntotas. Mejor asíntota, mejor certeza.

El siguiente gráfico fue copiado desde el libro de Cormen, porque ilustra como una función $g(n)$ trabaja como asíntota de la función $f(n)$.

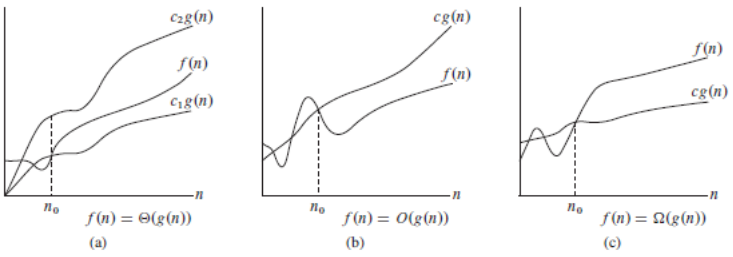


Figure 3.1 Graphic examples of the Θ, O, and Ω notations. In each part, the value of n_0 shown is the minimum possible value; any greater value would also work. **(a)** Θ-notation bounds a function to within constant factors. We write $f(n) = \Theta(g(n))$ if there exist positive constants n_0, c_1, and c_2 such that at and to the right of n_0, the value of $f(n)$ always lies between $c_1 g(n)$ and $c_2 g(n)$ inclusive. **(b)** O-notation gives an upper bound for a function to within a constant factor. We write $f(n) = O(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or below $cg(n)$. **(c)** Ω-notation gives a lower bound for a function to within a constant factor. We write $f(n) = \Omega(g(n))$ if there are positive constants n_0 and c such that at and to the right of n_0, the value of $f(n)$ always lies on or above $cg(n)$.

La notación-O (big o), sirve para dar una cota superior para el crecimiento de nuestro algoritmo, esta afirma que hay un punto n_0, tal que para los valores n después de este punto, el valor múltiplo de la función $g(n)$ supera o esta en el mismo orden que nuestro algoritmo $f(n)$, es decir que:

Existe un c y un n tal que $0 \leq f(n) \leq cg(n)$ para todo $n_0 \leq n$.

Y lo representamos como: $f(n) = O(g(n))$.

Matemáticamente podemos acotar $f(n)$ con un orden muy superior, pero no hacemos eso, porque queremos saber cuanto tardará nuestro algoritmo con el calculo más justo que podamos.

La notación-Ω (notación omega), sirve para dar una cota inferior para el crecimiento de nuestro algoritmo, esta afirma que hay un punto n_0, tal que para los valores n después de este punto, el valor múltiplo de la función $g(n)$ esta en un orden inferior o esta en el mismo orden que nuestro algoritmo $f(n)$, es decir que:

Existe un c y un n tal que $0 \leq cg(n) \leq f(n)$ para todo $n_0 \leq n$.

Y lo representamos como: $f(n) = \Omega(g(n))$.

La notación-Θ (notación theta), sirve para dar una cota superior e inferior para el crecimiento de nuestro algoritmo, esta afirma que hay un punto n_0, tal que para los valores n después de este punto, un valor múltiplo de la función $g(n)$ esta en el mismo orden que nuestro algoritmo $f(n)$, es decir que:

Existe un c_1 , c_2 y un n tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n_0 \leq n$.

Y lo representamos como: $f(n) = \Theta(g(n))$.

La notación o-chica, sirve para dar una cota estrictamente superior para el crecimiento de nuestro algoritmo.

esta afirma que hay un punto n_0, tal que para los valores n después de este punto, un valor múltiplo de la función $g(n)$ esta en un orden superior a nuestro algoritmo $f(n)$, es decir que:

Existe un c y un n tal que $0 \leq f(n) < cg(n)$ para todo $n_0 \leq n$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$

Y lo representamos como: $f(n) = o(g(n))$.

La notación ω, sirve para dar una cota estrictamente inferior para el crecimiento de nuestro algoritmo.

esta afirma que hay un punto n_0, tal que para los valores n después de este punto, un valor múltiplo de la función $g(n)$ esta en un orden inferior a nuestro algoritmo $f(n)$, es decir que:

Existe un c y un n tal que $0 \leq cg(n) < f(n)$ para todo $n_0 \leq n$.

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Y lo representamos como: $f(n) = \omega(g(n))$.

Aclaración de notación: En notación O, el igual '=' se debe leer como 'ES'. $T(n) = O(F(n))'$ se debe leer como $T(n) es O(F(n))$.

3.4 ejemplo de notación asintótica

El algoritmo insertion-sort crece $T(n) = n^2$, para ordenar n elementos hace n^2 operaciones.

El algoritmo merge-sort crece $T(n) = nlog(n)$, para ordenar n elementos hace $nlog(n)$ operaciones.

Esto significa que para n pequeños combine insertion-sort pero para casos grandes es mejor merge-sort.

Si n=1000, insertion-sort hace 1 millon de pasos, y mere-sort 3 mil.

Tenemos 2 computadoras:

- A (la rápida) 10 billones $\frac{instrucciones}{segundo}$.
- B (la lenta) mil millones $\frac{instrucciones}{segundo}$.

A, esta ordenando un arreglo de 10 millones de números mediante insertion-sort. Con el mejor programador del mundo , por lo que implementa al algoritmo insertion-sort con un running time $T(n) = 2n^2$. B, esta ordenando un arreglo de 10 millones de números mediante merge-sort. Con pepe, que aprendió ayer a programar con un compilador que esta seteado para no optimizar, por lo que lo implementa mere-sort con dificultad con un running time $T(n) = 50nlog(n)$.

¿Quien ganara? A es 1000 veces más rapida que B, en poder de computo. A tiene mejor programador que B.

Pero el ganador es B:

A tarda: $\frac{2(10^7)^2}{10billones} = 20$segundos.
B tarda: $\frac{50(10^7)\log 10^7}{milmillones} = 11.62674$ segundos.

Part II
Recursividad

4 Probar matemáticamente que un bucle funciona:

loop invariant:La invariante de un bucle es una declaración de las condiciones necesarias que se deben cumplir a la entrada de un bucle, esa condición se cumplirá para cada iteración del bucle.

(aqui los arreglos y vectores comienzan en 1, no en 0)
INSERTION-SORT(Array)
1 for j=2 to A.length
2...key = A[j]
3...// hay que insertar A[j] en la secuencia ordenada A[1,2,...,j-1].
4...i = j-1
5...while i > 0 and A > key
6.....A[i+1] = A
7.....i = i - 1
8 ...A[i+1] = key

Aquí la invariante del bucle es que la secuencia A[1,..,j-1] debe estar ordenada antes de entrar en el bucle en 5.

Debemos probar que el bucle funciona, esto es parecido a probar una propiedad matemática funciona y se mantiene mediante inducción matemática.

Tiene 3 reglas para saber si el algoritmo es correcto sobre el invariante (loop invariant):

- R1-Inicialización: El invariante es verdadero antes de la primera iteración del bucle.
- R2-Mantenimiento: El invariante es verdadero antes de una iteración y verdadera antes de la próxima iteración. La propiedad funciona para el inicio y funciona durante cada iteración del bucle.
- R3-Terminación: Cuando termina el bucle, el invariante verdadero nos muestra si el algoritmo es correcto.

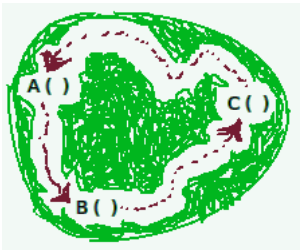
Ejemplo:

- R1-para INSERTION-SORT: vale porque hay un solo elemento el cual si o si esta ordenado.
- R2-para la INSERTION-SORT: vale porque se va ordenando de a sub-arreglos gracias al for principal que preserva el invariante.
- R3-para la INSERTION-SORT: cuando el bucle for termina, j es mayor al largo del arreglo. El sub-arreglo A[1,2...,n] consiste en sus elementos originales pero ordenado .

5 Recursividad:

Un **método recursivo** es aquel que esta definido en términos de si mismo. Este método, directa o indirectamente hace una llamada a si mismo**, pero con instancia del problema diferente.**

La **recursión** es el uso de métodos recursivos.
Esta se basa en los principios de la inducción matemática.
¿Cómo es posible resolver una instancia de un problema llamándose a si mismo?
La clave principal, es que se llama a si mismo, pero con diferentes instancias del problema.
Veamos la siguiente imagen:



El método A, puede llamar al método B. El método B, puede llamar al método C. El método C, puede llamar al método A. Independientemente a cual se llama primero para resolver un problema, ¿Cuáles son los métodos recursivos? Todos son métodos recursivos.

Ventajas de la recursividad:

- Claridad al expresar ideas.
- La implementación iterativa y la recursiva poseen el mismo orden de complejidad.
- Fácilde implementar.

5.1 Repaso de inducción matemática

La técnicaica de demostración de hecho que trabaja en 2 pasos:

- Primero: Se demuestra que el es cierto para casos sencillo, demostrándolos de forma directa, estos se los llama **caso base**. Se indica que de ser cierto para ciertos casos (**hipótesis inductiva**)se puede extender el siguiente caso.
- Segundo: Al demostrar como extender el caso podemos extender el caso indefinidamente (**paso inductivo**).

Por ejemplo queremos demostrar que la suma de los n primeros números es $\frac{n(n+1)}{2}$, con inducción:

- Primero mostramos a mano que vale para 1, 2 y 3
- Suponemos que vale para k, (si o si), $1 + 2 + 3 + ... + k = \frac{k(k+1)}{2}$
- Entonces suponemos que vale para k+1 , pero se comprueba. Si es verdad, la hipótesis es verdadera, sino es falsa. $1 + 2 + ... + k + (k + 1) = (k + 1) + \frac{k(k+1)}{2} = \frac{(k+1)(k+2)}{2}$ ==>es verdadera.

Para las comprobaciones de los algoritmos:

- Las comprobaciones, "hechas a mano", se denominan caso base, son sencillas y se comprueban fácilmente.
- Si suponemos cierto para un k arbitrario, suposición llamada hipótesis de inducción, por lo tanto también es cierto para un k+1.
- Ahora debemos ver una convergencia al caso base, para casos sencillos es suficiente. Si es cierto para el caso sencillo, es cierto para el llamador.

Caso base es una instancia, que se puede resolver sin emplear la recursión. Toda llamada debe progresar al caso base.

Podemos ver algunas características:

- Solo la ultima instancia llamada del problema esta activa, el resto no puede hacer trabajo hasta que retorne.
- No se manejan tareas de bajo nivel.
- Tenemos un o varios caso base, y llamadas progresivas a un caso base.
- El buen uso de la recursión no agota la memoria de la computadora.

Tip: Se puede tener una rutina guía que evalué las condiciones necesarias de la entrada para simplificar la función recursiva. (por ejemplo que los números sean mayores que 0, entonces llamo a la rutina guía y no a la recursiva. Sino la función recursiva va estar constantemente evaluando que los números sean mayores a 0.)

5.2 ¿Cómo funciona la recursión?

Cada lenguaje implementa las llamadas de una forma, pero esto no significa que sea así para los demás. La implementación de un método requiere tareas de bajo nivel. Se podría implementar tranquilamente una función con un lugar en memoria fijo para cada método, un lugar para las variables locales, otro que retorna al llamador, otro lugar para el valor de retorno, etc. Pero esto trae serios problemas, ya que impide la recursión, ya que los datos serian sobreescritos. Es necesaria la estructura de la pila. La implementación de métodos recursivos se hace mediante una estructura de dato llamada **pila de registros de activación**, donde un elemento del registro de activación, contiene información sobre el método: valores de parámetros, variables locales, etc. ¿Porque usamos la pila? la usamos porque la terminación de la función se realiza en sentido inverso a la invocación. La naturaleza de la pila es invertir la secuencia. Si se llama se pone en la pila, si se retorna se desapila.

Las variables locales están dentro del registro de activación (van en apiladas), porque de otra forma, las instancias de la función se compartirían y no se podría usar la recursividad. En lenguajes como Java o C/C++, cuando uno marca a una variable como *static*, esta pidiendo, en otras palabras que esa variable sea compartida entre todas las instancias, por lo tanto no es almacenada en la pila de registros de activación, es decir, no es tratada como una variable local.

5.3 4 Reglas:

- **R1) Regla del caso base:** Se debe tener al menos un caso base.
- **R2) Regla del progreso:** Las llamadas recursivas deben progresar al caso base.
- **R3) Regla del Puede creerlo:** Cuando se hace un algoritmo recursivo, asuma que la llamada retornara una respuesta correcta. La llamada recursiva internamente funciona correctamente por la hipótesis inductiva. Por lo tanto, ya no seguirá un largo y tortuoso camino hasta el caso base, ahora ya se cuenta con una herramienta matemática que facilita el diseño.
- **R4) Regla del interés compuesto (demasiada recursividad es mala):** Nunca duplique el trabajo resolviendo, la misma instancia de un problema, en llamadas recursivas separadas una misma instancia de un problema. Se puede converger a la solución, pero esto puede implicar la repetición de instancias ya resueltas en otras instancias de la función. Esto producirá mucho trabajo redundante. La recursividad en algunos casos, consume mucha memoria.

5.4 Divide-and-conquer

Con divide-and-conquer, podemos resolver un problema recursivamente, aplicando 3 pasos en cada nivel de la recursión:

- **Divide:** Dividir el problema en subproblemas disjuntos, que sean instancias menores al problema.
- **Conquer:** Resuelvo en forma recursiva.
- **Combine:** Combino las soluciones obtenidas a partir de las soluciones de los subproblemas.

Una vez que los sub-problemas se volvieron lo suficientemente pequeños como para no usar recursividad, diremos que la recursión toco fondo ("bottoms out") y habrá tendido a un caso base. Aveces, por la adición de sub-problemas que son instancias más pequeñas que el problema original, Debemos resolver sub-problemas que no están completamente en el problema original.

La diferencia entre divide-and-conquer y la recursión simple, es que divide-and-conquer requiere que las instancias generadas al dividir el problemas sean totalmente diferente una de otra. En cambio la recursión simple no lo requiere.

5.4.1 Cota superior para el algoritmo divide y vencerás

Mediante un análisis matemático especifíca que, el running time de un algoritmo que posee una ecuación con la forma:

$T(N) = AT(\frac{N}{B}) + O(N^k)$

Donde:

- En cada nivel del Algoritmo, se generan A subproblemas.
- En cada nivel del Algoritmo, se reduce un factor B el tamaño de problema original, al pasar al siguiente nivel.
- En cada nivel del Algoritmo, se tiene un conste $O(N^k)$ para dividir y resolver.

$$T(N) = \begin{cases} O(N^{\log_B A}) & si....A > B^k \\ O(N^k \log N) & si...A = B^k \\ O(N^k) & si...A < B^k \end{cases}$$

Un teorema parecido es el teorema master, el cual plantea un limite theta para todo caso recursivo

5.5 Tail-Recursion

En algunos algoritmos recursivos, se pueden implementar en un caso especial de recursividad llamado Tail Recursion (recursividad por cola), la cual es una técnica para optimizar la recursividad, eliminando las constantes llamadas recursivas.

¿Cuándo una función es Tail recursion? Es cuando la llamada recursiva es la última instrucción de la función; con la restricción que en la parte que realiza la llamada recursiva, no exista otra expresión alguna.

¿Cuándo hago los cálculos lógicos? Se realizan los cálculos primero, y luego se realiza la recursión.

Propiedad: El valor que retorna la instancia del caso base, es el que retorna la función. Esto nos da la ventaja de poder realizar llamadas recursivas, sin la necesidad de tener un stack frame más. Es decir, podemos evitar la sobrecarga de cada llamada a la función y nos evitamos el gasto de memoria de pila. Con el compilador adecuado, una función tail recursive se puede evitar lo que se conoce como stack overflow, que ocurre cuando la pila de llamadas (call stack) consume mucha memoria, porque simplemente estaremos reutilizando el mismo stack frame en la próxima instancia de la función. **La cantidad de información que debe ser almacenada durante el cálculo es independiente del número de llamadas recursivas.** El compilador es el responsable de lograr la optimización. El compilador trata una función tail recursive como si fuera una función iterativa.

Por ejemplo, tenemos el enfoque del lenguaje Lisp. Si la recursión no es la única parte de la expresión return, la maquina retornara en la evaluación, por lo tanto no estamos en el fin (en el tail) de la función, pero si en el medio de la expresión. No obstante esto no se

aplica a los parámetros de la llamada recursiva, todo esta permitido aquí. Si un lenguaje no implementara la iteración, podría usar tail recursión como Haskell, porque esta recursión se comporta como una iteración. La optimización de calls por jumps es únicamente posible para esa llamada exterior.

Por Ejemplo, el calculo del factorial no tail-recursive:

```
int factorial_recursivo(int n){
    if(n == 1)
       return n;
    else
       return n * fact_recursivo(n-1);
}
```

Por Ejemplo, el calculo del factorial con tail-recursive:

```
int fact_tail_sum(int n, int sum){
    if(n == 1) {
       return sum;
    } else {
       return fact_tail_sum(n - 1, sum*n);
    }
}
int factorial_recursivo(int n){
    return fact_tail_sum(n, 1);
}
```

5.6 Coroutines

Donald Knuth, al tratar de resolver un problema de generación de tareas de cooperación, se encontró sin herramientas de alto nivel para resolverlo de forma elegante.

Él abrió un esquema totalmente diferente, sobre las funciones, dejo de pensar en funciones como llamadoras y funciones llamadas. El parte de la idea que ambas funciones cooperan por igual.

Las Corrutinas son componentes de programas informáticos que permiten generalizar las subrutinas, a una función con múltiples puntos de entrada, permitiendo la suspensión y la reanudación de la ejecución en ciertos lugares. Corrutinas son adecuadas para la generación de los componentes iterables que requieren tener almacenado el estado anterior. En otros términos: remplaza la tradicional primitiva de llamada y retorno con una parecida.

La nueva primitiva de llamada, guardará el valor retornado de donde fue llamado, en un lugar diferente a la pila, y luego saltará a una ubicación específica dentro de la función llamada, en un valor de retorno guardado.

Esta es una teoría muy buena, pero en la época de Knuth no había lenguaje de alto nivel compatible con la llamada corrutina primitiva. La funciones destinatario tendrá todos los problemas. Para que cada vez que la llamemos retomemos el control justo después de que retornamos.

Las subrutinas son casos especiales de las corrutinas. Cuando subrutinas se invocan, la ejecución comienza desde el principio y una vez que sale de subrutina, esta terminada.

Las corrutinas son similares, excepto que también puede salir rindiéndose o “yielding”, que es el equivalente al “return”, para luego retomar desde este punto la próxima vez que sea llamado, o salir llamando a otros, lo que les permite volver a entrar en ese punto una vez más, desde el punto de vista de la corrutina , no es salir, sino simplemente llamar a otra corrutina. La implementación de un lenguaje con corutinas requeriría la creación de stacks adicionales cuando usa corutinas.

5.6.1 Las Coroutinas de python desde la versión 2.5:

La documentación de python dice:

Caundo la sentencia yield es ejecutada, el estado se congela y el valor es retornado a la función llamadora. Por congelar, entendemos, que todos los estados locales están retenidos, hasta la próxima vez que la función vuelva a ser invocada. La función procederá como si hubiese efectuado una llamada a una función externa.

El yield es el punto de retorno y de re-ingreso a la corutina. La corutina es fácil de crear e implementar.

En python una corutina retorna un Objeto iterable, que es este el que otorga realmente los valores retornados.

```
def rota(people):

    _people = list(people)
    current = 0
    while len(_people):
    yield _people[current]
    current = (current + 1) % len(_people)

if __name__ == "__main__":


    people = ["Ant", "Bernard", "Carly", "Deb", "Englebert"]
    r = rota(people)
    for i in range(10):
       print "It's %s's turn." % r.next()
```

La corutina de python se puede combinar con la recursividad y abandonar y retomar la rutina recursiva en un punto clave, sin necesidad de volver por cada nivel donde la función fue llamada, hasta llegar al llamador. Solo ejecuta "yield", y listo.

5.6.2 Las corutinas en C

Este lenguaje no cuenta con este tipo de componente, aunque hay macros implementadas que simulan de forma simple y elegante la corutina, impide la combinación con la recursividad, puesto que no es una corutina usada en forma natural.

6 Recursividad y el Problema de la Solución Óptima

Podríamos clasificar a los problemas en tres clases (solo se tratan las dos primeras clases):

- Problemas que poseen solución única, por lo tanto, es la mejor solución.
- Problemas que poseen varias soluciones correctas, pero solo un subconjunto de las soluciones, es una solución óptima.
- Problemas que tienen forma de maximización o minimización de una variable objetivo y poseen restricciones de valores para cada variable, tomando forma de una serie de ecuaciones lineales con estas variables. Esta clase se denomina **problema de programación lineal.**

Definición: Problema de optimización: Es cuando existen diferentes soluciones correctas. Pero solo algunas de ellas es una solución óptima, Y se requiere la solución óptima para la vida real.

Ejemplo: El problema de cambio de monedas: Para una divisa con monedas de $C_1, C_2, ..., C_n$ unidades, ¿Cual es el mínimo numero de monedas que se necesitan, para devolver K unidades de cambio? Es un problema de optimización porque podemos dar de varias maneras el dinero a la persona, pero solo una solución es la óptima.

Por ejemplo: La emisión de billetes de un cajero-automático a un cliente del banco, asumimos, que siempre hay cambio y la emisión óptima es dar la mínima cantidad de billetes posible.

Una solución es: Mientras el monto a entregar sea diferente de cero, Iterar desde la mayor unidad hasta la menor unidad, Restar al monto a entregar un múltiplo de la unidad iterada, contabilizar dichas monedas que se le restaron y entregarlas. **Pero no sería una solución óptima:** sean las monedas: \$50, \$25, \$21, \$10, \$5 y \$1 y hay que dar \$42: la solución óptima es 2 de \$21. Pero la otorgada por el algoritmo es \$25, \$10, \$5 y \$1, el cambio es correcto pero no óptimo.

Definición: Un problema que posee una Subestructura Optimal: Implica que el problema puede dividirse en subproblemas y que las soluciones óptimas al subproblema son soluciones óptimas al problema original.

Por ejemplo Si para ir de Córdoba a Carlos Paz. Y el camino más corto implica partir de Córdoba pasar por el Puente 15, luego pasar por El Tropezón y finalmente ir a Carlos Paz. La subestructura optimal implica que si quiero ir de Córdoba a El Tropezón, deberé pasar por el Puente 15.

6.1 Algoritmo de vuelta atrás o Algoritmos de Backtracking

Es el algoritmo que usa la recursión para probar sistemáticamente todas las posibilidades en soluciones, almacenando la mejor solución hasta el momento. Se basa en el procesamiento de una lista de candidatos a la mejor solución, que a la vez puede generar nuevos candidatos. El recorrido en la búsqueda en el espacio de soluciones puede ser en profundidad o en anchura. **Por ejemplo** MiniMax.

6.2 Algoritmos Greedy o Algoritmos Devoradores o codiciosos.

Los algoritmos Greedy, son algoritmos que toman decisiones óptimas en cada instancia, sin pensar en el futuro, por ejemplo si la instancia del problema es parte de un problema mayor, Por eso estos algoritmos dan soluciones óptimas cuando el problema tiene subestructura optimal, pero no necesariamente al revés. Cuando no tiene, puede o no dar solución óptima.

¿Problemas que pueden pasar? En algún momento puedo recalcular los problemas y subproblemas porque estos se superponen. Cuando esto pasa, el algoritmo no es eficiente.

Soluciones:

- Establecer una profundidad global para el camino: Es decirle al algoritmo, que no se pase de x profundidad porque entonces no sera buena la solución otorgada.

- Emplear Programación dinámica almacenando en tablas los cálculos ya hechos.

7 Recursividad y Programación dinámica.

La programación dinámica es un tipo de recursividad que resuelve subproblemas generados por un planteamiento recursivo de forma no recursiva. Esto lo logra mediante el almacenamiento de las instancias ya resueltas dentro de una tabla.

La programación dinámica ofrece una solución tabulada, en la cual arma una tabla de resultados óptimos de los subproblemas. La tabla de resultados contiene, la instancia del problema, información al respecto del problema y su correspondiente solución óptima. Se emplea la tabla porque la instancia es probable que fuese resuelta en una ocasión anterior, además los resultados alamacenados son óptimos y se pueden reutilizar para tener el resultado optimo de un problema superior.

Hay dos formas de realizar el llenado de la tabla, son **dos enfoques:**

- **Top-down:** Es el enfoque que resuelve el problema de la generación de instancias repetidas al dividir. Se llena la tabla cuando se llega a resolver una instancia.
- **Bottom-up:** Empieza siempre llenando el caso base para luego incrementar el problema y converger al problema inicial. En la tabla, no almaceno la solución, sino una forma de construir la solución. La filosofía de "bottom-up" (desde el principio hacia delante): si vamos calculando los valores en el orden correcto, tendremos en cada momento todos los valores que necesitamos.

Por Ejemplo podría atacar el problema del cambio de moneda con buttom-up, almacenando el cambio para $1, luego para $2, y así hasta llegar al monto inicial. y retornar la solución óptima.

Programación dinámica se realiza con los siguientes pasos:

1. Encontrar la subestructura optimal y usar esta para construir una solución óptima al problema a partir de los subproblemas.
2. Mediante recursividad debemos definir el costo en términos de la solución óptima.
3. Computar el costo de una solución optima actual, es decir calcular el costo de mi solución añadiendo el costo le le tomo a una anterior instancia que resolvió el subproblema que combine.
4. Construir la solución óptima en base a la información que fue computada.

Patrones comunes para descubrir subestructuras optimales:

1. Si el problema muestra que la solución es crear una elección y la creación de esta elección, genera uno o más subproblemas a resolver.
2. Supongamos que para el problema dado, toma la solución que conduce a la solución optimal. No nos importara como determinamos esa elección. Solo asumimos que la obtuvimos.
3. Con la elección realizada, determinamos cuales subproblemas seguirán y cómo caracterizara el resultante espacio de subproblemas.
4. Se muestra que las soluciones de los subproblemas usados en la solución optimal deben por si solos ser optimales . Si supone que cada solución de un subproblema no es óptima y luego deriva en una contradicción, entonces deberá demostrar que puede obtener una mejor solución para el problema original, ya que por contradicción usted supone tener una óptima solución. Si se tiene una solución optimal a partir de varios subproblemas, estos se podrán unir con bajo esfuerzo.

7.1 Memoización enfoque Top-down de la programación dinámica.

Problema de la Obtención del i-esimo numero de la secuencia de fibonacci:

Definicióndel problema: Se requiere el iesimo numero de la secuencia de fibonicci, donde : f(i) = f(i-1)+f(i-2), donde f(0)=0 y f(1)=1

Solución en python:

```
def fibonacci(n):
    if n < 2:
       return n
    return fibonacci(n-1) + fibonacci(n-2)
```

Esta es la peor implementación de fibonacci, ya que se realiza en forma un crecimiento exponencial de problemas.

f(5) llamara a f(3) y f(4)
f(4) llamara a f(3) y f(2)
f(3) llamara a f(1) y f(2)
f(2) llamara a f(0) y f(1)
Se llamara 3 veces a f(0)
Se llamara 5 veces a f(1)
Se llamara 3 veces a f(2)
Se llamara 2 veces a f(3)
Se llamara 1 veces a f(4)
Las llamadas de fibonacci crecen como su secuencia :)

Se ve claramente que se vuelven a resolver instancias del problema que ya fueron resueltas.

Se podrían dar circunstancias de que el único algoritmo (fibonacci no es un ejemplo valido, porque esta mal implementado) se presenten viejas instancias del problema ya resueltas.

Existe una estrategia, para atacar estas circunstancia:

La memoización: Que es un tipo de recursividad en donde se cuenta con una cache de instancias del problema ya resueltas (una cache de funciones realizadas). En donde basta con buscar en la cache primero antes de intentar resolverlo (no invocar en forma recursiva a las ya realizadas). No es bueno, que la cache sea una variable global, Se puede tener una cache en recursividad sin variables globales. Con memoización ganamos tiempo a costa de perder espacio en memoria.

Ahora supongamos que hacemos:

```
def fibonacci(n):
    if n < 2:
       return n
    return fibonacci(n-1) + fibonacci(n-2)

def memoize(fun):
    cache = { }
    def f(*args):
       if args not in cache:
          cache[args] = fun(*args)
       return cache[args]
    return f

#retorna una version memoizada de f!
fibonacci = memoize(fibonacci)
```

La llamada a fibonacci(50), sin memizacion parece interminable, pero con memoizacion retornara rápidamente.

Paso a explicar el código,

- Primero creo una función de fibonacci común.(en la linea 1,2,3,4).
- Luego, creo una función que contiene una lista vacía llamada cache. (linea 6 y 7).
- La tabla llamada cache, no desaparece cuando la función *memoize* termina (linea 12).
- La función *memoize* retorna una función que fue escrita en términos del argumento. Cache no es una variable global, tampoco local, sino es algo llamado **closure**, que puede accederla función que estaba dentro del scope de cache al ser escrita.

- En la linea 14, *memoize(fibonacci)* hace reescribir a la función f, en términos de *fibonacci*, en la linea 10 , cambiando allí dentro a la dirección de fun por la de *fibonacci* (la dirección de la linea 1).
- En el caso de "*def f(*args)*" el asterisco consume todos los argumentos posteriores y los une en una tupla única. En el caso de "*cache[args] = fun(*args)*", el asterisco desempaqueta la tupla.
- Para finalizar La linea 14. Setea a *fibonacci,* por lo que sobrescribe a las futuras llamadas que llamen a *fibonacci*, en realidad estarán llamando a la función f incluido a fibonacci mismo que llamara a f en la linea 4.

¿Mucho lío?

```
@memoize
def fibonacci(n):
    if n < 2:
       return n
    return fibonacci(n-1) + fibonacci(n-2)
```

El decorador memoize, indica que queremos memoizar dicha función.

7.2 Un ejemplo de Bottom-Up

Para que quede claro, lleno la tabla del problema del cambio de moneda. Mi divisa tiene las monedad $1,$2,$5 y $10

Puesto que el problema es más complicado que el de fibonacci, la tabla tiene más elementos en las entradas.

Cambio a Entregar ($)	0	1	2	3	4	5	6	7	8	9	10	11
Lo que entrego (resta)	0	1	2	2	2	5	1	2	2	5	5	5
Trabajo realizado(costo)	0	1	1	2	2	1	2	2	3	3	2	3

Paso a explicar como llene mi tabla de lo simple a lo complejo:

- El algoritmo asume que un hombre quiere sacar $0, entonces me sitúo en la entrada 0 (rojo), no tengo que restar nada así que el trabajo es 0.
- Ahora asume que quiere sacar $1, entonces me sitúo en la entrada 1 (rojo). Resto una unidad, quedando el monto en cero, no hace falta continuar, el trabajo fue 1.
- Ahora asume que quiere sacar $2, entonces me sitúo en la entrada 2 (rojo). Resto 2 unidades, quedando el monto en cero, no hace falta continuar, el trabajo fue 1.
- Ahora asume que quiere sacar $3, entonces me sitúo en la entrada 3 (rojo). Leo las entradas anteriores y escojo La primer entrada con moneda directa que minimize el conto del trabajo realizado al final. (En este caso da lo mismo, escojo el 2)Resto 2 unidades, quedando el monto en uno, hace falta continuar, entonces me sitúo en la entrada 1 (rojo), resto la unidad y calculo el trabajo, que fue 2.
- Ahora asume que quiere sacar $4, entonces me sitúo en la entrada 4 (rojo). Leo las entradas anteriores y escojo la primer entrada con moneda directa que minimice el costo del trabajo realizado al final. Escojo el 2, puesto que escojer el 1, aumentaria el trabajo. Resto 2 unidades, quedando el monto en dos, hace falta continuar, entonces me sitúo en la entrada 2 (rojo), resto 2 , queda 0 y calculo el trabajo, que fue 2.
- Ahora asume que quiere sacar $5, entonces me sitúo en la entrada 5 (rojo). Resto 5 unidades, quedando el monto en cero, no hace falta continuar, el trabajo fue 1.
- Ahora asume que quiere sacar $6, entonces me sitúo en la entrada 6 (rojo). Leo las entradas anteriores y escojo la primer entrada con moneda directa que minimice el costo del trabajo realizado al final. Escojo el 5 o el 1, da lo mismo, pero pongo el 1 para diferenciarme de greedy. Resto 1 unidad, quedando el monto en 5, hace falta continuar, entonces me sitúo en la entrada 5 (rojo), resto 5 , queda 0 y calculo el trabajo, que fue 2.

- Asume $8=1+2+5
- En síntesis, el resultado de la resta es el índice del próximo lugar a ser visitado. En la parte del trabajo se podría dejar una indicación del próximo lugar a ser visitado, en caso de ambigüedad.

7.3 Resumen

¿Posee Subestructura optimal?	¿Se generan instancias diferentes?	Enfoque
NO	NO	Backtraking
—–	SI	Dividir y reinar
SI	NO	Programación dinamica
SI	SI	Algoritmos Greedy

Part III
Algoritmos de Ordenación

8 Ordenación mediante comparación

Los algoritmos de Ordenación mediante comparación, son aquellos que ordenan usando solo comparaciones. Por ejemplo: Insert-sort es basado en comparaciones, pero Radix-sort no es basado en comparaciones.

Los algoritmos de Ordenación in-place, son aquellos que ordenan sin usar espacios de memoria adicionales. Por ejemplo: Quick-Sort es in-place, pero Merge-sort no es in-place.

Los algoritmos de Ordenación estables, son aquellos que ordenan sin cambiar el orden de aparición de elementos repetidos. Por ejemplo: Merge-sort es estable, pero Shell-sort no es estable. Solo con algoritmos de ordenación estables se puede conseguir ordenar una lista de nodos en cascada (Ordenar por nombre y aplellido por ejemplo).

¿Con que derecho nos atrevemos a decir que tan ordenado esta un arreglo al verlo? El grado de desorden de un arreglo nos lo da el número de inversiones que posee un arreglo.

Una inversión en un vector A, es todo par (i, j) que cumple $i < j$ y $A_i > A_j$.

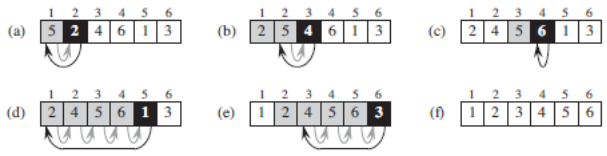
Entonces podemos decir que **la salida de un algoritmo de ordenación** es la permutación de la entrada que no contiene inversiones.

Suponiendo que recibimos en forma constante arreglos llenados al azar, **el numero medio de inversiones de un vector de tamaño N es** $\frac{n(n+1)}{4}$. Este numero se deduce Pensando que la cantidad de permutaciones posibles de un vector es $\frac{n(n+1)}{2}$, donde $\frac{n(n+1)}{2}$ es el máximo numero y 0 es el mínimo como es en promedio, el resultado es la mitad.

El algoritmo de ordenación eficiente, es el que más inversiones resuelve por intercambio de posiciones. Los algoritmos que solucionan una inversión adyacente, eliminan exactamente una inversión, por lo que en promedio tendrán $\frac{n(n+1)}{4}$ inversiones a realizar. Por lo que tendrán un running time promedio $T(N) = \Omega(N^2)$.

8.1 Insert-Sort

Esta basado en comparaciones, es in-place y es estable. Es más facil imaginarse que nos estan entregando cartas y las vamos añadiendo a nuestro mazo ordenado que esta en nuestra mano.



La parte gris son las cartas que tenemos en las manos y la negra es la ultima carta otorgada. **Como vamos en el sentido de máximo a mínimo en el recorrido, mantenemos el algoritmo estable.**

```
/*
* Ordena los elemntos de 'p' a 'r'
* en el arreglo A[...,p,..,r,...]
*/
void InsetSort(int p,int r){
int key;
int i;
for (int j = p+1; j <= r ;j++){
key = A[j];
i = j-1;
while( i >= p &&  A[i]>key){
A[i+1] = A[i];
--i;
}
A[i+1]=key;
}
return;
}
```

El insert-sort soluciona en promedio inversiones adyacentes, y tiene un running time promedio de $T(N) = \Theta(N^2)$.

La memoria auxiliar usada es el tamaño de un nodo $MEM = O(1)$, por eso se lo considera in-place.

8.2 Shell-Sort

Creado por Donal Shell, pero estudiado en profundidad por Gonnet. Esta basado en comparaciones, es in-place y no es estable. Trata de eliminar la mayor cantidad de inversiones al comparar dos elementos que se encuentran distanciados, en el principio, luego compara entre elementos más cercanos, hasta finalmente comparar elementos adyacentes.

Las distancias estan puestas en una serie llamada **secuencia de incrementos**: $h_{t,}, ..., h_2, h_1$ donde va desde el incremento inicial al incremento final. El incremento final siempre vale 1 $h_1 = 1$.

Después de ordenar en la fase de incremento h_k para todo i vale: $a[i] \leq a[i + h_k]$, los elementos a esa distancia están ordenados.

Shell sugirió que la secuencia de incrementos sea $h_t = \frac{N}{2}$ y $h_{k-1} = \frac{h_k}{2}$.

Gonnet demostró en la practica que la secuencia de incrementos seria mejor: $h_t = \frac{N}{2}$ y $h_{k-1} = \frac{h_k}{2,2}$.

Se realiza ordenación por inserción entre los elementos a distancia h_k.

Original	32	95	16	82	24	66	35	19	75	54	40	43	93	68	
After 5-sort	32	35	16	68	24	40	43	19	75	54	66	95	93	82	6 swaps
After 3-sort	32	19	16	43	24	40	54	35	75	68	66	95	93	82	5 swaps
After 1-sort	16	19	24	32	35	40	43	54	66	68	75	82	93	95	15 swaps

```
/*
* Ordena los elementos del 0 al n-1
* en el arreglo int A[..]
*/
void shellsort(int n) {

    int i,aux;
    //itrero los incrementos
    for(int h = n/2; h>0 ; h = (h==2? 1 : h/2.2) ) {
    //insertsort adaptado
       for(int j = h; j < n; j++) {
          key = A[j];
          i = j;
          while(i>=h && key<A[i-h]){
            A[i] = A[i-h];
            i-=h;
          }
          A[i]=key;
       }
    }

}
```

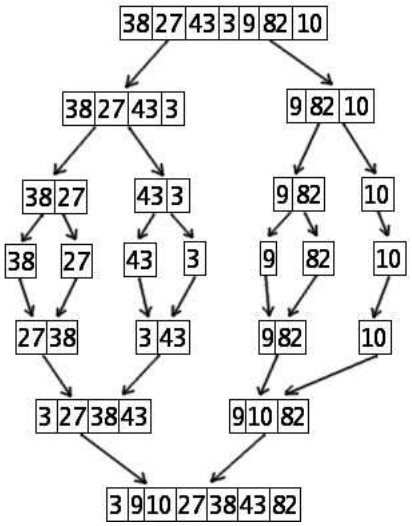
Los casos medio del algoritmo aun nadie lo sabe. El peor caso del algoritmo es $T(N) = \Omega(N^2)$. Se considera subcuadratico.

La memoria auxiliar usada es el tamaño de un nodo $MEM = O(1)$, por eso se lo considera in-place.

8.3 MergeSort

Es un algoritmo basado en comparaciones, es estable, pero no es in-place. Esta escrito bajo la filosofía divide y vencerás. Conceptualmente hace:

- Divide: Un arreglo desordenado en 2 arreglos desordenados con la mitad de tamaño.
- Reinar: Ordenar las sublistas.
- Combinar: Unir o mezclar (de la traducción de la palabra *merge*) dos listas ordenadas, en una lista ordenada.

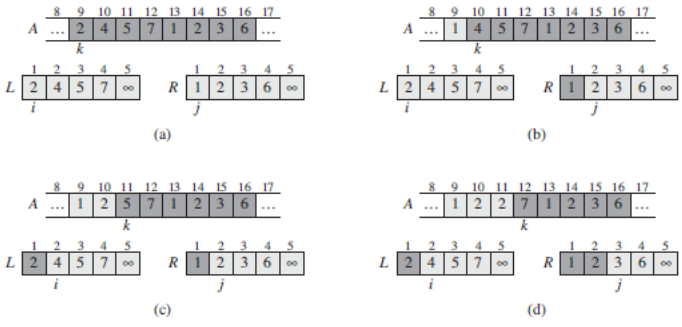


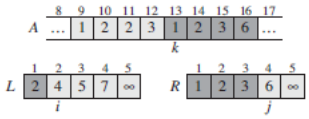
¿Cuál es el costo de mezclar dos arreglos en uno? Es de orden lineal, Porque el nuevo arreglo puede ser creado en N operaciones de costo constante cada una. Lo que nos da un running time de $T(N) = 2T(\frac{N}{2}) + O(N)$ que por la cota de los algoritmos divide y vencerás (o empleando el teorema master) sabemos que el running time es $T(N) = N \log N$.

Teniendo cuidado al cambiar, logramos que sea una ordenación estable.¿Cuanta memoria extra usa? La memoria auxiliar usada es el tamaño de todos los nodos $MEM = O(N)$, NO ES in-place.

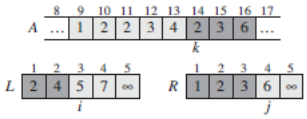
8.3.1 Operación Merge

La mezcla parte de de dos vectores de entrada L y R, Produciendo al vector A. El algoritmo tiene 3 índices en el principio de cada vector i, j y k respectivamente. Comparamos L_i contra R_j, después almacenamos el menor en A_k, después incrementamos k, después incrementamos i si L_i era el menor, o incrementamos j si R_j era el menor. (ignoren esos infinitos de la imagen)

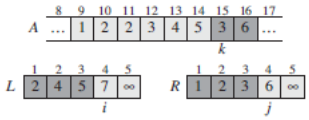



8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 1 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

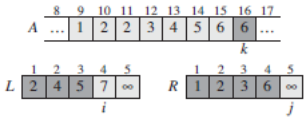
(e)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 2 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

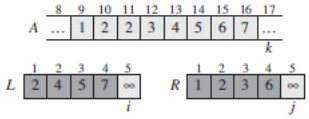
(f)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 3 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(g)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 6 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(h)


8 9 10 11 12 13 14 15 16 17
A … 1 2 2 3 4 5 6 7 …
k
1 2 3 4 5
L 2 4 5 7 ∞
i
1 2 3 4 5
R 1 2 3 6 ∞
j

(i)

```
/*
 * Ordena los elemntos de 'p' a 'r'
 * en el arreglo A[...,p,..,r,...]
 */
void MergeSort(int p, int r){
int q;
 if(p < r){
q = (p + r)/2;
MergeSort(p, q);
MergeSort(q + 1, r);
Merge(p, q, r);
}
}
/*
 * Mezcla los elementos Desde A[p] y A[q+1]
 * hasta Desde A[q] y A[r]
 */
void Merge(int p, int q, int r){
int i = p;
int j = q+1;
int k = 0;
int* mem = (int*) malloc((p+r+1)*sizeof(int));
while(i<=q && j<=r){
if(A[i] <= A[j])
mem[k++] =A[i++];
else
mem[k++] =A[j++];
}
while(i<=q) mem[k++] =A[i++];
while(j<=r) mem[k++] =A[j++];
for(i=0;i<=r-p;i++){
A[p+i]=mem[i];//Lleno A
}
}
```

Este es el algoritmo que usa python para la ordenación de listas, pero esta implementado en C++. ¿Por qué no usa Quick-Sort? Por que Quick-Sort NO ES ESTABLE, y piensa que el usuario razono con algoritmos estables!!!

8.4 Quick-Sort

Es un algoritmo basado en comparaciones, NO ES estable, pero ES in-place. Se considera que en la practica, en promedio es el más rápido de los de comparaciones, pero es discriminado por no ser estable. Esta escrito bajo la filosofía divide y vencerás. Conceptualmente hace:

- Divide: Divide un arreglo desordenado en 2 arreglos desordenados con la mitad de tamaño, salvo un elemento que se encuentra en la posición final, que esta situado entre los 2 subarreglos.
- Reinar: Escoge un elemento del arreglo y lo coloca en la posición final.
- Combinar: Retorna un arreglo ordenado de 2 subarreglos con elemento en el medio que se coloco en ese nivel.

El running time en el peor caso es $T(N) = O(N^2)$. Pero el rendimiento del caso promedio es $T(N) = O(N \log N)$.

Funcionamiento:

1. Si el arreglo tiene 1 o 0 elementos es un arreglo ordenado y retorna.
2. En caso contrario, Se escoge un elemento v del arreglo S, al cual ***llamaremos pivote.***
3. Colocar v en la posición final. Particionar $S - \{v\}$en dos subarreglos disjuntos I y D, tal que todo elemento $x \epsilon I$ sea menor a v y todo elemento $y \epsilon D$ sea mayor a v. (En solo uno de esos arregos van elementos iguales a v).
4. Retornar el arreglo ordenado I seguido de v, seguido de D.

La selección del pivote es el corazón del algoritmo, hacerlo mal equivale a tener un algoritmo de orden cuatrático.

8.4.1 Selección Del Pivote

¿Cómo es la selección ideal del pivote? El mejor caso se presenta cuando el pivote divide al conjunto en 2 subconjuntos de igual tamaño. $T(N) = O(N \log N)$.

¿Cómo es la selección pésima del pivote? El peor caso se presenta cuando reiteradamente el pivote genera un subconjunto vacío. $T(N) = T(N-1) + O(N) = O(N^2)$.

El caso medio es $T(N) = O(N \log N)$.

Esto nos dice de forma indirecta que las entradas que degeneran al algoritmo son, un arreglo ya ordenado y un arreglo con todos los elementos repetidos.

¿Cómo elegimos, de forma rápida y sin conocer el elemento del medio al pivote?

- **Escoger el primer elemento:** Es la forma más popular, puesto asume que el arreglo esta muy desordenado, cuando en la realidad no es así. Y este mal comportamiento se repetirá nivel por nivel, por eso se desaconseja usarlo. Nunca use más el primer elemento.
- **Escoger el ultimo elemento:** Por iguales razones estadísticas, no use el ultimo elemento.
- **Escoger el elemento al azar:** Es una buena elección, y se tendrá el caso promedio.
- **Escoger el elemento del medio:** Es una elección razonable, evita los casos que degeneran al algoritmo, pero se considera una elección pasiva, intenta evitar elegir un pivote malo.
- **Escoger la mediana de tres:** Escoge un elemento mejor que el del medio. Primero seleccionamos el primer elemento, el $\frac{N}{2}$-esimo y el ultimo elemento, finalmente escogemos la mediana de los 3, como elemento pivote. Implementar la mediana de tres requiere ordenar los tres valores y colocar el pivote al final.

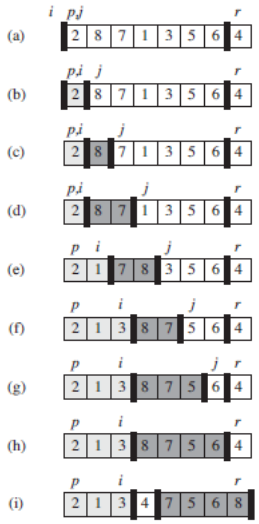
8.4.2 Estrategia de la Partición

Pasos para la partición:

1. Intercambiar al pivote contra el ultimo elemento.
2. Agrupar los elementos menores a la izquierda y los mayores a la derecha.
3. Intercambiar el primer elemento del grupo de los mayores contra el pivote.

Hay varias formas de hacer el segundo paso:

- Una es recorrer con dos índices desde los extremos hasta el centro y comparar contra el pivote. Se busca un elemento mayor que el pivote en la primera mitad, Se busca un elemento menor que el pivote en la segunda mitad, Se intercambian dichos elementos y se vuelve a buscar hasta llegar a que se junten los dos índices. Cuando eso pasa el indice que viaja de izquierda a derecha esta seleccionando un elemento mayor al pivote, se intercambian ese índice contra el pivote, y se retorna el índice del pivote para que se vuelva a dividir en quicksort.
- Otra es ir intercambiando al pivote contra un elemento mayor encontrado en la primera mitad y luego intercambiarlo contra uno menor el la segunda mitad, hasta que no se pueda hacer más.
- Otra es es recorrer con dos índices desde la izquierda hacia la derecha agrupando un primer sector con los menores y un segundo sector con los mayores. (versión implementada). Como vemos en el grafiquito, el índice r señala al pivote y al final del subarreglo, y p al principio. El índice i señala el ultimo elemento menor al pivote encontrado, El índice j señala el elemento que se esta analizando, por lo que podemos decir que: Los elementos con los índices desde p a i (salvo al principio que vale $(p-1)$) son la zona de los elementos menores. Los elementos con los índices desde $i+1$ a $j-1$ (salvo al principio) son la zona de los elementos mayores. Cuando en la posición j se descubre un elemento mayor al pivote simplemente solo se incrementa j . Cuando en la posición j se descubre un elemento menor al pivote se incrementa incrementa i y luego se se intercambia contra el elemento de la posición j , después se incrementa j.



```
/*
 * Eleccion del pivote mediante la mediana de tres.
 * Finaliza llamando a la funcion que particiona
 */
int PrePartition(int p,int r){
int aux;
int m = (r+p)/2;//indice que esta entre p y r
//si el primero es el mayor, lo paso al ultimo
if(A[p] > A[m] && A[p] > A[r]){
aux = A[r];
A[r] = A[p];
A[p] = aux;
}
else {
//si el medio es el mayor lo paso al final
if(A[m] > A[p] && A[m] > A[r]){
aux = A[r];
A[r] = A[m];
A[m] = aux;
}
}
//si medio es menor lo paso al principio
if(A[m] < A[p]){
aux = A[p];
A[p] = A[m];
A[m] = aux;
}
//a los del medio los hago penultimos
aux = A[r-1];
A[r-1] = A[m];
A[m] = aux;
return Partition(p+1,r-1);
}
/*
 * Selecciona el ultimo como pivote
 * y encuentra su lugar particionando
 * en mayores y menores al pivote
 * por ultimo intercambia al pivote
 *  con el primer numero mayor
 * de 0 a i son los elementos menores al pivote
 * de i+1 a j-1 son los mayores al pivote
*/
int Partition(int p,int r){
int x, aux;
int i;
x = A[r];// x es el pivote
i = p-1;
//j de p a r-1
for(int j = p ; j <= r-1 ; j++){
if(A[j] <= x){//si es menor
i++;
aux = A[i];
A[i] = A[j];
A[j] = aux;
}
}
//se intercambaran los lugares con el primer mayor al pivote.
aux = A[i+1];
```

```
A[i+1] = A[r];
A[r] = aux;
return i+1;
}
/*
 * Ordena los elemntos de 'p' a 'r'
 * en el arreglo A[...,p,..,r,...]
 */
void QuickSort(int p,int r){
int q;
if(p<r){
if(r-p <= 7){
//si es menor a 7->InsetSort
InsetSort(p,r);
}
else{
//q = Partition(p,r);
q = PrePartition(p,r);
QuickSort(p,q-1);
QuickSort(q+1,r);
}
}
return;
}
```

8.5 Cota inferior para los Algoritmos basados en comparaciones

Suponiendo algoritmos basados en comparaciones mejores aun no descubiertos ¿Cuánto mejor son?

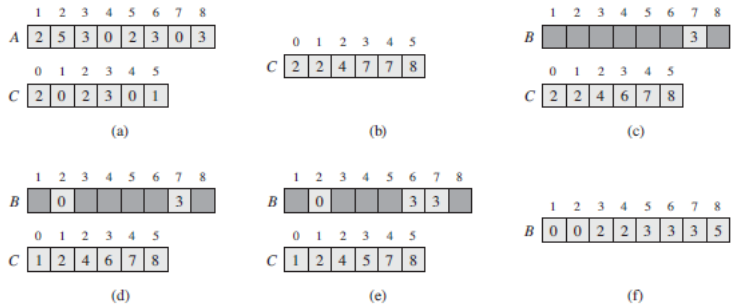
Teorema: Cualquier Algoritmo que emplee únicamente comparaciones para ordenar elementos, requiere al menos emplear $\log(N!)$ comparaciones, por lo que tendría un running time de $T(N) = \Omega(N \log N)$.

Prueba: Si tenemos como entrada un arreglo de N, elementos , como nos interesa el orden, El numero de permutaciones posibles es $N!$, como logaritmicamente bajamos el numero de permutaciones posibles al eliminar inversiones (una mitad de permutaciones posibles, y otra segunda mitad de permutaciones imposibles por la eliminación de inversión), queda claro es es $\log(N!)$.

8.6 Counting-Sort

NO ES un algoritmo basado en comparaciones, ES estable, pero NO ES in-place. Basicamente se contabiliza la cantidad de apariciones de los elementos en una tabla, la cual es indexada por el mismo elemento. El resultado es que se muestran en orden las apariciones.

Se asume que la entrada de los elementos están dentro del conjunto $C = \{0, ..., k\}$, usando este arreglo para determinar el reativo orden, se puede ordenar una entrada de N elementos en $T(N) = O(N + k) = \Theta(N)$. Este cuenta Con el arreglo A como entrada, B como salida y C es el lugar donde se almacenaran la contabilizacion.



Pasos:

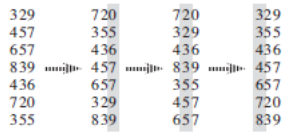
- Primero contabilizamos lo de A, en el arreglo C. (imagen a)
- Segundo Acumulo las contabilizaciones.(imagen b)
- En modo inverso, tomo el ultimo elemento de A (como si fuese una pila), indexo con ese elemento en el arreglo C y obtengo la posición, en el arreglo B. Luego decremento esa posición en uno. (imagen c,d y e).

Retornando el ultimo nodo en aparición podría hacer al algoritmo estable.

```
def CountingSort(A,k)
    c=[0]*k
    #contabilizo lo de A
    for x in A:
        c[x]+=1
    #acumulo las contabilizaciones
    for i in range(1,k):
        c[i]+=c[i-1]
    #pongo el ultimo elemento en su posicion correcta
    for x inreversed(A[:]):
        c[x]-=1
        A[c[x]]=x
```

8.7 RadixSort

NO ES un algoritmo basado en comparaciones, ES estable, pero NO ES in-place. Se usa para extender el rango de la entrada. Donde la entrada tiene d digitos y cada digito toma k posibilidades. RadixSort reordena la entrada de N elementos en orden $T(N) = O(d(N + k)$.




```
radixSort(A,d)
    for i = 1 to d:
       use a stable sort to sort array A on digit i.
```

Part IV
Repazo avanzado (Pila, Cola y Lista)

Las Estructuras de Datos, son tipos de datos abstractos.
Abstract Data Type (ADT): Es el conjunto de:

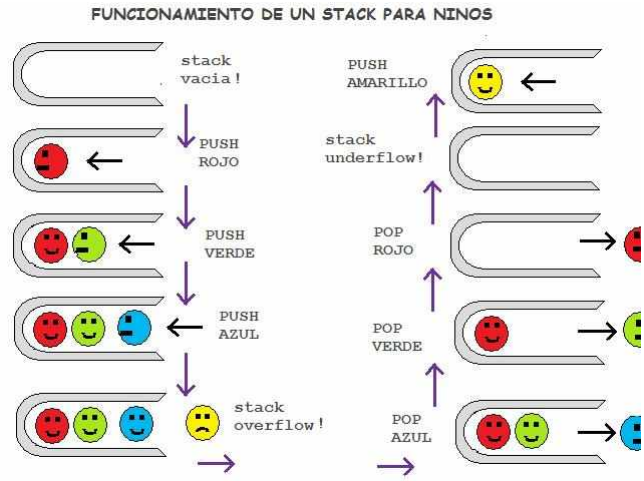
- Representación de los datos.
- Conjunto de las operaciones permitidas.

9 Pilas(Stacks)

Pila:

- Representa los datos como un conjunto de datos que se van apilando uno sobre otro. Como una pila de libros. La pila puede ser implementada con un vector o una lista con la política lifo.
- Las operaciones permitidas restringen el acceso solo a la cima de la pila.
  - push o apilar inserta un elemento en la cima de la pila.
  - pop o desapilar quita un elemento de la cima de la pila.
  - climb o cima lee el elemento más recientemente insertado.

Las operaciones básicas son de orden constante.



10 Colas (Queues)

Cola:

- Representa los datos como un conjunto de datos que van formando cola teniendo prioridad el primero en llegar. Como la cola de un cajero de supermercado. La cola puede ser implementada con un vector o una lista con la política fifo.
- Las operaciones permitidas restringen el acceso solo a la cabeza de la cola.
  - Insertar o insert, inserta un elemento al final de la cola.
  - Quitar o quit, quita el primer elemento de la cola.
  - Primero o first, lee el primer elemento de la cola.

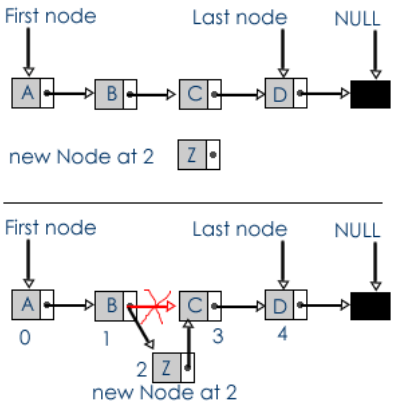
Las operaciones son de orden constante.

11 Lista Enlazada

11.1 Lista Enlazada Simple (Linked list)

Lista Enlazada Simple:

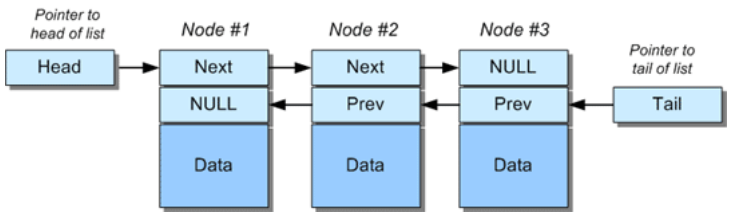
- Los elementos se almacenan en forma no contigua, cada elemento se almacena en un nodo contenedor, con acceso al siguiente nodo de la lista. Los datos se representan como un conjunto de nodos encadenados por el puntero al siguiente elemento.
- El acceso al nodo tiene el costo del recorrido de todos los nodos desde la cabeza (primer nodo) hasta el nodo buscado. Por lo que el acceso es de orden lineal.
  - Inserción, introduce un nodo en la lista, requiere tener la referencia al nodo anterior del que pensamos añadir siguiente a él, para actualizar las referencias.
  - Búsqueda, leer el contenido del nodo buscado.
  - Eliminación, requiere tener la referencia al nodo anterior del buscado que pensamos eliminar, para actualizar las referencias.



11.2 Lista Enlazada Doble(Double Linked List)

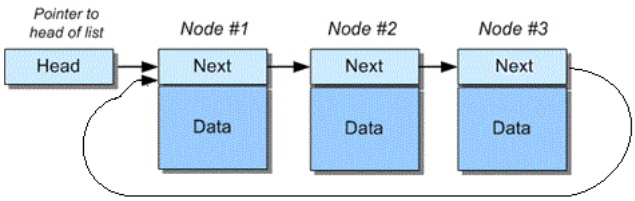
Lista Enlazada Doble:

- Es una lista enlazada simple donde cada nodo contenedor, además de tener acceso al siguiente nodo de la lista, tiene otro acceso al anterior nodo de la lista.
- El acceso al nodo tiene el costo del recorrido de todos los nodos desde la cabeza (primer nodo) hasta el nodo buscado. Por lo que el acceso es de orden lineal.
  - Inserción, introduce un nodo en la lista, requiere tener la referencia al nodo anterior del que pensamos añadir siguiente a él y el nodo posterior, para actualizar las referencias.
  - Búsqueda, leer el contenido del nodo buscado.
  - Eliminación, requiere tener la referencia al nodo anterior y posterior del buscado que pensamos eliminar, para actualizar las referencias.

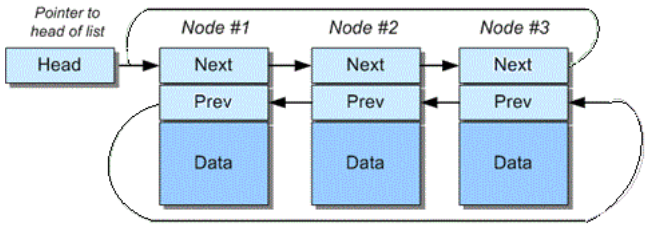


11.3 Lista Enlazada Circular

Lista Enlazada Simple Circular: Es una lista enlazada simple donde el ultimo nodo contenedor, tiene acceso al primer nodo de la lista como nodo siguiente.



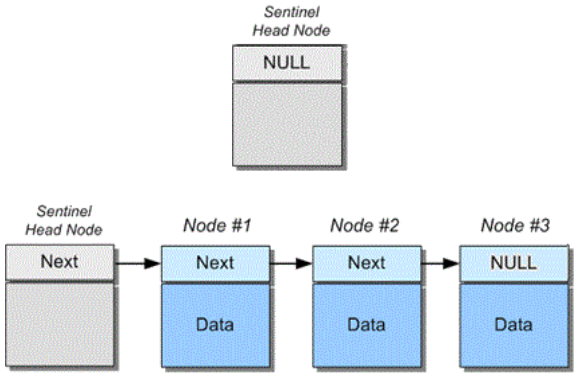
Lista Enlazada Double Circular: Es una lista enlazada doble donde el ultimo nodo contenedor, tiene acceso al primer nodo de la lista como nodo siguiente; y el primer nodo de la lista tiene acceso al ultimo nodo de la lista como nodo anterior.



11.4 Nodos Dummy

Simplificación de la implementación de las listas enlazadas simple mediante la utilización de nodos Dummy (o Sentinel, centinela):

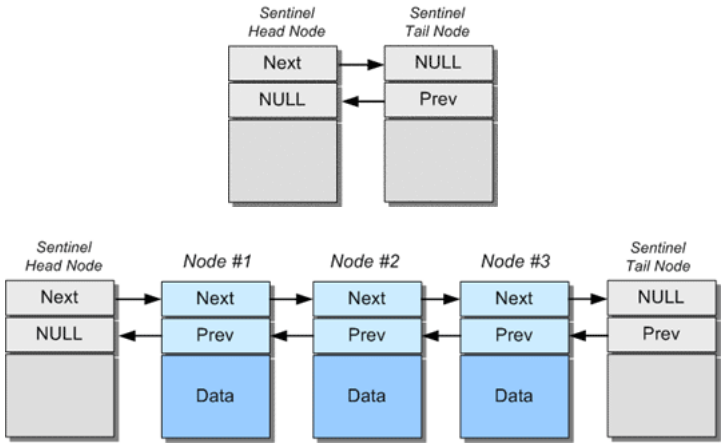
- La implementación de las listas enlazadas, requiere tener la referencia al nodo anterior, por que se piensa que siempre se inserta o elimina un elemento después de un nodo.
- Lo que introduce dos casos especiales:
  - la inserción al principio de la lista
  - la eliminación el principio de la lista.
- Si satisfacemos el requisito de que siempre hay un nodo anterior, simplificamos el código a cambio de una pequeña penalización en memoria, tener siempre al principio un nodo ficticio, al cual las rutinas de exploración ignoraran.



Simplificación de la implementación de las listas enlazadas doble mediante la utilización de nodos Dummy (o Sentinel, centinela):

- La implementación de las listas enlazadas doble, requiere tener la referencia al nodo anterior y siguiente, por que se piensa que siempre se inserta o elimina un elemento después de un nodo y antes de otro.

- Lo que introduce cuatro casos especiales:
  - inserción al principio de la lista
  - inserción al final de la lista
  - eliminación el principio de la lista.
  - eliminación al final de la lista.
- Si satisfacemos el requisito de que siempre hay un nodo anterior y posterior, simplificamos el código a cambio de una pequeña penalización en memoria, tener siempre al principio un nodo ficticio y otro nodo ficticio al final, aquellos nodos las rutinas de exploración ignoraran.



11.5 Implementación de las listas

Implementación de las Listas:

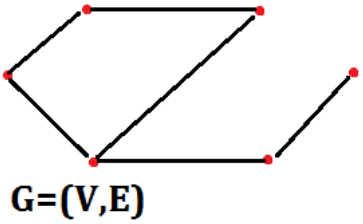
- **Visión clásica:** Como una clase que representa a una lista (puntero al principio del nodo y puntero al nodo actual) y contiene las operaciones principales.
- **Visión recursiva de los nodos:** Como una clase que representa a un nodo como contenedor de una lista (datos del nodo y puntero al siguiente nodo) y contiene las operaciones principales que son recursivas.

Part V
Grafos

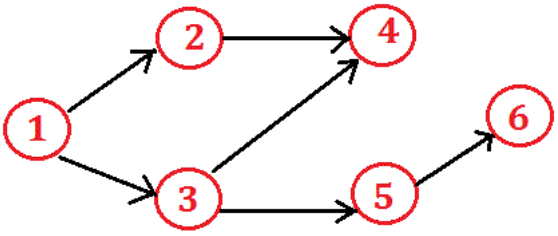
12 Definiciones

Un **grafo es** una estructura de datos que esta formada por:

- El conjunto de los vértices o nodos.
- El conjunto de las aristas o arcos.



G=(V,E)



V={ 1, 2, 3, 4, 5, 6 }
E={ (1,2), (1,3), (2,4), (3,4), (3,5), (5,6) }

Cada arista esta definida por dos vértices que une. Las aristas son pares ordenados. Sí el orden es relevante decimos que el **grafo es dirigido**, pues una permutación diferente es otra arista. Si el orden es relevante la arista (3,4) es diferente de la (4,3). El grafo es dirigido o no dirigido, **no hay híbridos**.

Se dice que el vértice 2 es adyacente al vértice 1 si (1,2) esta en el conjunto de las aristas.

Las aristas pueden contener un único peso.

Los pesos de las aristas suelen representar medidas de distancias o tiempos entre los nodos.

Existe un **camino** de un vértice a otro si hay una sucesión vértices adyacentes en el medio.

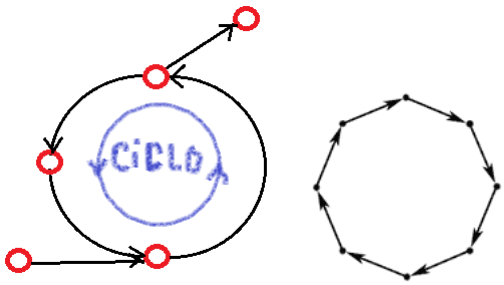
La **longitud del camino** sin pesos, es la cantidad de aristas recorridas en el camino.

La longitud del camino con pesos, es la acumulación de los pesos de las aristas recorridas.

Existe un camino de un vértice a si mismo de longitud cero, sin pasar por ninguna arista.

Por ejemplo el algoritmo RIP (para solucionar el problema de ejercer la función de IGP en los routers): Utiliza la longitud de los caminos sin pesos.

Un **ciclo** en un grafo dirigido, es un camino, que empieza y termina en el mismo vértice que empieza y pasa al menos por una arista. Un grafo acíclico, no posee ciclos.



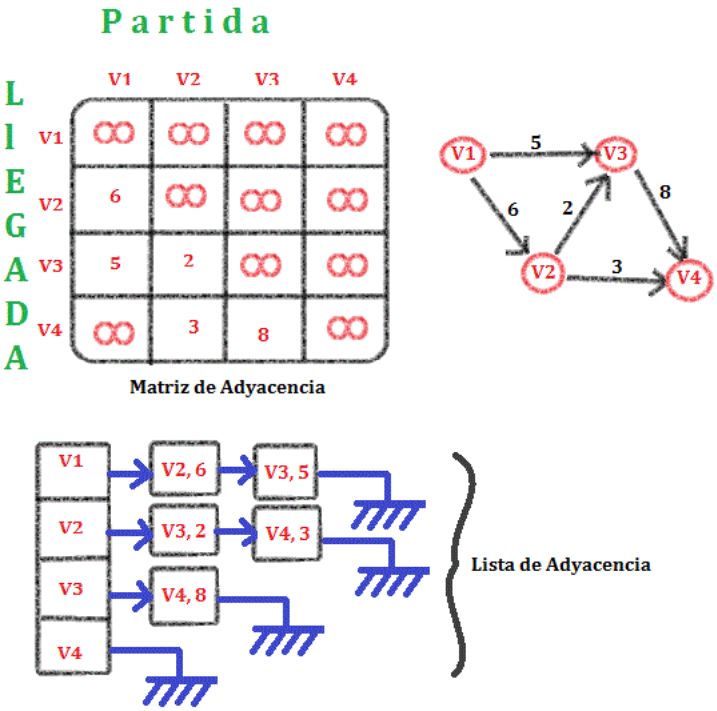
Un **grafo denso** cumple que la cantidad de aristas es proporcional a la cantidad de vértices al cuadrado:$\|E\| = \Theta(\|V\|^2)$. Un grafo disperso cumple que la cantidad de aristas es proporcional a V: $\|E\| = \Theta(\|V\|)$.

La cantidad de aristas entrantes, es el **grado de entrada del nodo**.

La cantidad de aristas salientes, es el **grado de salida del nodo**.

Por ejemplo: Un caso especial, de grafo es el Árbol, que cada nodo tiene grado de entrada a lo sumo uno.

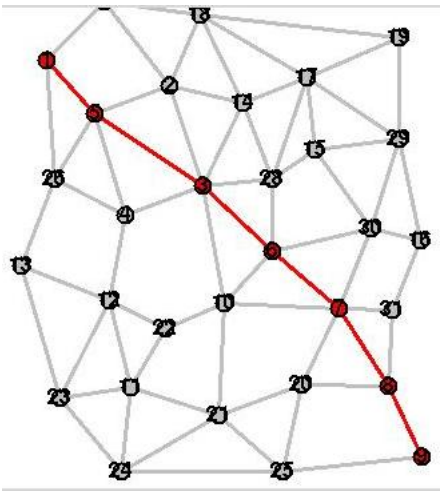
13 Ideas de Implementaciones de grafo



Representación por lista de adyacencia: Que es un conjunto de listas (donde un vector de listas o un diccionario de listas) que a cada vértice, le corresponde una lista de vértices adyacentes a él. El espacio empleado en memoria es de orden de $MEM = \Theta(\|E\|)$, lo cual es justo para grafos dispersos o densos.

Representación por Matriz de adyacencia: Mediante una matriz bidimensional, cada arista (v,w) almacena su peso en el espacio A[v,w], Las aristas inexistentes pesan infinito. Ocupa una memoria un orden de $MEM = \Theta(\|V\|^2)$. La inicialización de la matriz es de orden $O(\|V\|^2)$ .. El espacio almacenado no es justo para matrices dispersas.

14 Calculo del camino mínimo y camino máximo



Ciclo positivo: Cuando al dar un ciclo y acumular el peso de las aristas (o contabilizar las aristas), este tiene un valor positivo.

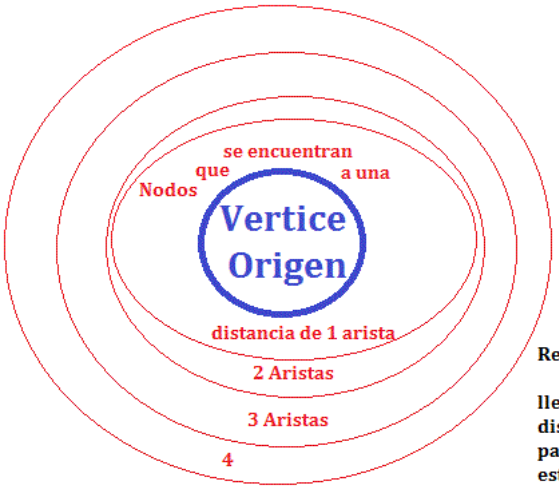
Ciclo negativo: Cuando al dar un ciclo y acumular el peso de las aristas, este tiene un valor negativo.

Marcando a un nodo como nodo Origen y al resto como nodos destino:

- Deseo calcular el camino mínimo que es el camino cuya longitud sea la más pequeña posible.
- Deseo calcular el camino máximo que es el camino cuya longitud es la más grande posible.
- ¿Tiene sentido hablar de un camino máximo entre dos nodos el cual puede pasar por un ciclo positivo? No, pues siempre va ha existir un camino más grande al dar otra vuelta por el ciclo.
- ¿Tiene sentido hablar de un camino mínimo entre dos nodos el cual puede pasar por un ciclo negativo? No, pues siempre va ha existir un camino más chico al dar otra vuelta por el ciclo.
- No existen caminos mínimos en grafos con ciclos negativos y no existen caminos máximos en grafos con ciclos positivos.

14.1 Grafo sin pesos

Problema del camino mínimo sin pesos: Quiero encontrar el camino mínimo desde el vértice origen hacia el resto de los vértices, en un grafo sin pesos (el peso son las aristas recorridas).



Búsqueda en anchura del camino mínimo (BFS):

- Se emplea una cola para almacenar todos los nodos adyacentes al que se analiza.
- Los nodos del grafo poseen color: Negro, Gris o Blanco.
- Los nodos del grafo poseen predecesor: el cual es el nodo anterior que lo puso en la cola.
- Los nodos que están pintados de blanco, representan que nunca fueron visitados (nunca estuvieron en la cola).
- Los nodos grises son nodos visitados (están en la cola), pero nunca se han encontrado el camino minimo.
- Los nodos negros son nodos visitados, pero se ha encontrado el camino mínimo de ellos (salieron de la cola).
- Al inicio todos los nodos están pintados de blanco y tiene longitud del camino mínimo igual a infinito ($+\infty$).
- El nodo origen tiene longitud del camino mínimo igual a 0, esta pintado de gris y es el unico nodo en la cola. El predecesor del nodo origen es nadie.

El algoritmo toma un elemento de la cola:

- El caso especial es que el nodo tomado sea el nodo origen cuyo predecesor es nadie, y no se relaja la longitud del camino mínimo.
- De otra forma se relaja la longitud del camino mínimo desde infinito a la longitud del predecesor + 1.
- En un momento, los nodos negros son los nodos que están a distancia $1, 2, .., n-2, n-1, n$, los nodos grises están a distancia $n, n+1$, el resto son nodos blancos.

Después, tomo los nodos blancos de la lista de adyacencia del nodo, los pinta en color gris y los coloca en la cola (marcando el predecesor que los llamo). Por ultimo pinto al nodo como negro.

Vuelvo empezar hasta que todos los nodos sean negros y termino.

Observar que **cada nodo solo es rebajado una sola vez.** El running time de la búsqueda en anchura es $T(V, E) = O(\|V\| + \|E\|)$. Podemos estar seguros de que obtenemos el camino mínimo, actualizando (relajando) una sola vez a la longitud, porque los nodos se van procesando en orden de niveles, por lo que, otro coste, seria superior o igual al camino mínimo.

14.2 Grafo sin aristas de peso negativo

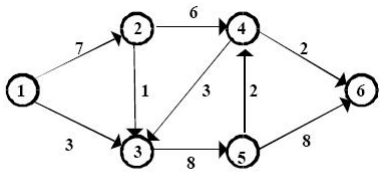


Figura 1.

Problema del camino mínimo con pesos positivos: Quiero encontrar el camino mínimo desde el vértice origen hacia el resto de los vértices, en un grafo con pesos no negativos.

Proponemos que al visitar nodo a nodo y relajando al nodo w que es adyacente a v como $D_w = min(D_w, D_v + c_{vw})$. Esto hace que no se garantice que actualizaremos una sola vez.

Teorema: Si vamos posando la atención de vértice en vértice, a un vértice no visitado, entre los que minimicen el valor de D_i, el algoritmo producirá correctamente el camino mínimo, siempre que no halla aristas negativas.

Algoritmo de Dijkstra:

- Emplea una cola de prioridad para almacenar los nodos adyacentes al visitado.
- El nodo origen tiene un camino mínimo igual a 0, el predecesor es nadie, esta pintado de gris y esta en la cola de prioridad.
- Al principio todos los nodos tienen un camino mínimo igual a infinito y están pintados de blanco, porque nunca estuvieron en la cola de prioridad.
- Todos los nodos grises están en la cola de prioridad para ser visitados (relajados de ser necesario).
- Todos los nodos negros ya tienen la longitud del camino mínimo.
- El algoritmo termina cuando todos los nodos son negros.

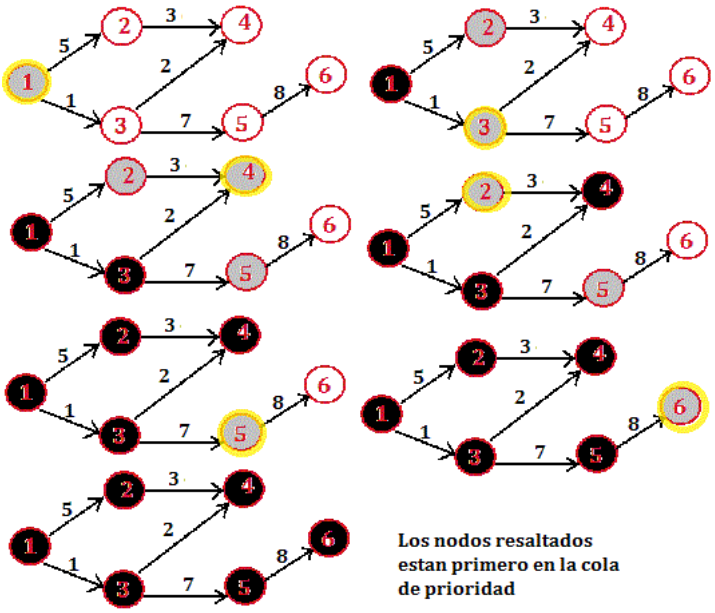
El algoritmo toma un elemento , elemento v, de la cola de prioridad (que es el elemento con menor longitud del camino de los nodos en la cola), Relajo los nodos blancos y grises de la lista de adyacencia de v con $D_w = min(D_w, D_v + c_{vw})$.

Pinto los nodos blancos en gris y los coloco en la cola de prioridad.

Pinto el nodo v en color negro.

Y vuelvo a comenzar hasta que todos los nodos sean negros.

El running time del algoritmo de dijkstra es de $T(E, V) = O(\|E\|log\|V\|)$, puesto que insertar un elemento en la cola de prioridad es de orden constante y extraer un nodo es de orden logarítmico con respecto a la cantidad de nodos, cuando la cola de prioridad se ha implementado como un montículo binario.

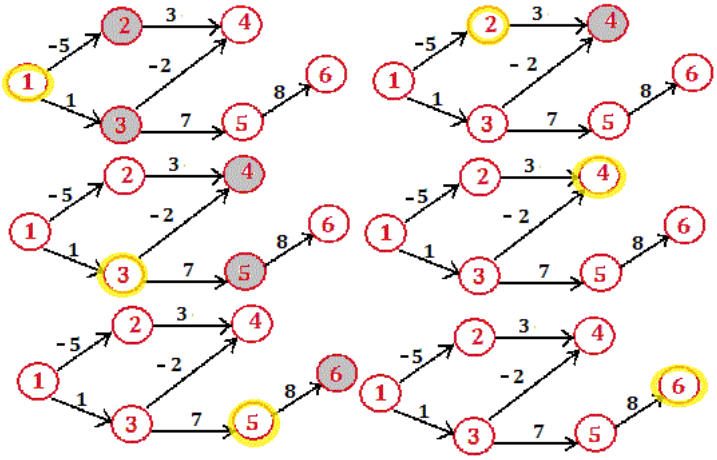


14.3 Grafo sin ciclos negativos

Problema del camino mínimo sin ciclos negativos: Quiero encontrar el camino mínimo desde el vértice origen hacia el resto de los vértices, en un grafo con sin ciclos negativos.

Algoritmo Bellman-Ford: Recorre todos los vértices y para cada vértice recorre su lista de adyacencia y rebaja cada vez que puede. Cada vértice es analizado una vez por cada arista de entrada que tenga.

El running time del algoritmo es $T(E,V) = O(\|E\|\|V\|)$



```
Bellman-ford()
    para cada v en V (desde el origen)
       para cada w en LISTA-DE-ADYACENCIA(w)
          Relax(w): Dw=min(Dw, Dv+Cvw)
```

14.4 Grafo con orden topológico

Son grafos que no contienen ciclos, el cual se puede ordenar los nodos en una secuencia de vértices, los cuales se verifica que hay un solo camino de un vértice a otro.

Si hay un camino de v a w, entonces w solo puede aparecer después de v en una de las posibles ordenaciones topológicas.

Un ejemplo de grafo con orden topológico son los PERT.

Propiedades del grafo:

- El recorrido del grafico en orden topológico, garantiza que el nodo tendrá el camino mínimo, al pasar por él e intentar relajar los nodos adyacentes a él.
- Algoritmo que arma el orden topológico, lo hace con ayuda de una cola: Poner los nodos de grado de entrada 0 en la cola, luego elimina los enlaces logicamente de la lista de adyacencia. Buscar nuevo nodos con grado de entrada 0, y repite hasta que esten todos los nodos en la cola o que solo halla nodos con grado superior o igual a 1.
  - Si hay nodos con grado de entrada mayor o igual a uno, es que hay presencia de ciclos, por lo que se rompe el orden topologico.
- Dado que no hay ciclos positivos podemos buscar el camino más largo, llamado camino critico.
- Por ejemplo usamos esto en un grafo de eventos PERT.

Problema del camino critico: Quiero encontrar el camino máximo desde los vértice de origen (con longitud cero) hacia el resto de los vértices, en un grafo con sin ciclos.

Algoritmo: Tomo un nodo de la cola hasta que este vacía, y modifico el tiempo del camino máximo como $D_w = max(D_w, D_v + c_{vw})$.

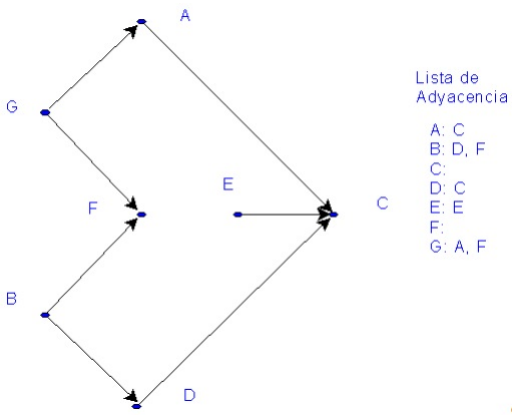
Suponiendo que son actividades que corren a contratiempo y deseo minimizar el tiempo del proyecto. El nodo con mayor longitud, es el tiempo del proyecto.

Problema del retraso máximo de la actividad: Si quiero calcular cuanto puedo demorar una actividad sin retrasar al proyecto. A los últimos nodos los igualo al camino máximo y de ahí empiezo.

Lleno la cola en orden topológico inverso, con los nodos con su correspondiente camino critico.

Algoritmo: Tomo un nodo de la cola hasta que este vacía, y modifico el tiempo de demora como $H_v = min(H_v, H_w - c_{vw})$.

Las aristas que cumplen $H_v - D_w - c_{vw} == 0$ son parte del camino critico.



Part VI
Árboles con raíz

15 Definiciones

Un **árbol** es un grafo acíclico, en el cual el conjunto de nodos solo tiene a lo sumo una arista entrante y existe un nodo distinguido llamado **raíz**.

A cada nodo c, excepto a la raíz, le llega una arista desde el nodo p. Decimos que c es hijo del nodo p y que p es padre del nodo c. Un nodo puede tener uno o más hijos, un nodo sin hijos, es llamado **nodo hoja**.

Hay un único camino desde la raíz hasta cada nodo. Si un árbol tiene N nodos, entonces tiene N-1 aristas.

Longitud del camino, es el número de aristas recorridas.

Profundidad(para arriba): es el número de aristas desde el nodo a la raíz.

Con visión recursiva: la raíz tiene profundidad 0, y un nodo tiene la profundidad del padre +1.

Altura(para abajo): es el número de aristas desde el nodo al nodo más profundo. Toda hoja tiene altura cero.

Altura del árbol, es la altura de la raíz.

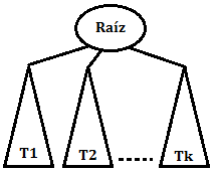
Los nodos que tienen el mismo padre se denominan **hermanos** o **siblings**.

Si hay un camino de u a v, entonces decimos que v es **descendiente** de u y u es **ascendente** de v.

El tamaño de un nodo es el número de descendientes que tiene + 1 (más el mismo).

El **tamaño del árbol** es el tamaño de la raíz.

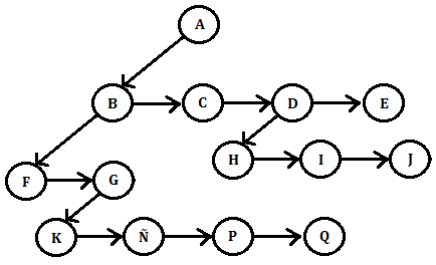
Visión Recursiva de árbol: Cuando hay 0 nodos se dice que el árbol esta vacío, en caso contrario el árbol consiste en un nodo denominado raíz, el cual tiene 0 o más referencias a otros árboles, conocidos como subárboles. Un árbol es vacío o una raíz y cero o más subárboles no vacíos T1, T2, ,...,Tk. . Las raíces de los subárboles se denominan hijos de la raíz, y consecuentemente la raíz se denomina padre de las raíces de sus subárboles. Cada uno de los subárboles estan conectados a la raíz mediante una arista. Cada nodo define un subárbol del cual el es raíz. Una visión gráfica de esta definición recursiva se muestra en la siguiente figura:



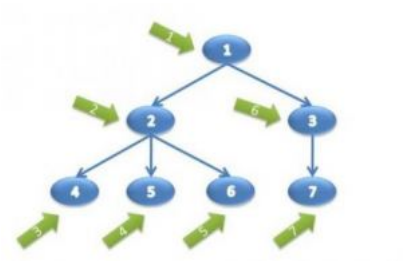

```
Clase NodoArbol:

    Clave
    Clase NodoArbol hijo
    Clase NodoArbol siguienteHermano
```

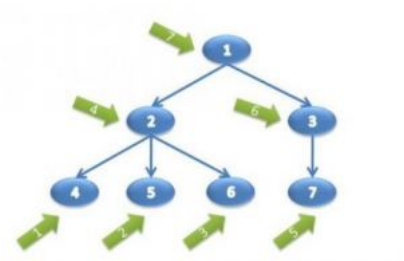
Cada nodo es una lista enlazada de hermanos.



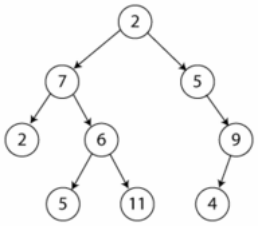
Recorrido en Preorden: Se lleva a cabo el procesado sobre el nodo, antes de hacerlo sobre sus hijos.



Recorrido en Postorden: Se lleva a cabo el procesado sobre sus hijos, antes de hacerlo sobre el nodo.



16 Árbol binario



Un Árbol binario, es un árbol el cual cada nodo puede tener a lo sumo 2 hijos. Distinguidos ahora como **nodo izquierdo** y **nodo derecho**.

El **tamaño** de un nodo es: el tamaño del hijo izquierdo+ el tamaño del hijo derecho + 1 (más el mismo).

El tamaño del árbol es el tamaño de la raíz.

Altura del nodo es igual a la del hijo con mayor altura más 1.

16.1 Recorrido e iteraciones sobre el Árbol binario

La **iteración** de los nodos en un determinado orden de recorrido de los arboles: consiste en remplazar las rutinas de recorridos con corutinas y remplazar donde diga "procesar(nodo)" por "yield(nodo)".

Los **recorridos de forma no recursiva**, se implementan con una pila, donde se apila una tupla que es el nodo con el estado. La cima de la pila representa el nodo que estamos visitando en cierto instante.

16.1.1 Preorden

Recorrido en Preorden: Se lleva a cabo el procesado sobre el nodo, antes de hacerlo sobre sus hijos.

```
Preorden(nodo):
    procesar(nodo)
    if nodo.izquierdo:
       Preorden(nodo.izquierdo)
    if nodo.derecho:
       Preorden(nodo.derecho)
```

Algoritmo no recursivo: Cada nodo es visitado 1 sola vez, es decir es apilado y cuando es desapilado se procesa el nodo y luego se añaden sus hijos, primero el derecho y luego el izquierdo (acordarse que la pila invierte el orden), para que sea procesado primero el izquierdo y luego el derecho.

La tupla no requiere estado.

```
Preorden(nodoRaíz):
    Stack.push(nodoRaíz)
    do:
       nodo = Stack.pop()
       procesar(nodo)
       if nodo.derecho not null:
          Stack.push(nodo.derecho)
       if nodo.izquierdo not null:
          Stack.push(nodo.izquierdo)
    while( not Stack.isEmpty() )
return
```

16.1.2 Postorden

Recorrido en Postorden: Se lleva a cabo el procesado sobre sus hijos, antes de hacerlo sobre el nodo.

```
Postorden(nodo):
    if nodo.izquierdo:
       Postorden(nodo.izquierdo)
    if nodo.derecho:
       Postorden(nodo.derecho)
    procesar(nodo)
```

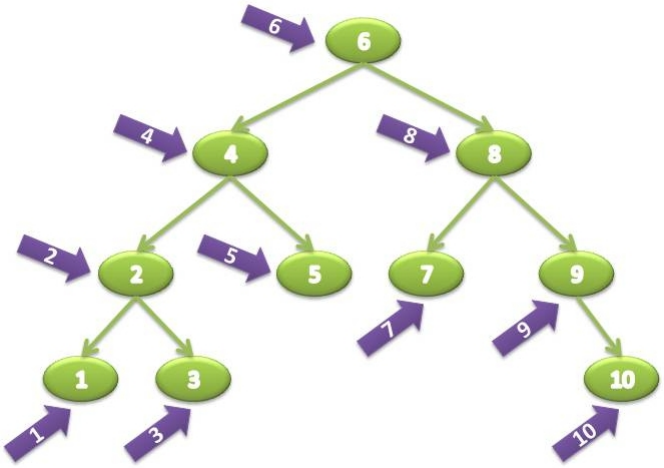
Algoritmo no recursivo: Cada nodo es visitado 3 veces, es decir apilado y desapilado 3 veces.

Si el nodo es desapilado con el estado:

- 'primera': A punto de visitar/apilar el hijo izquierdo del nodo en estado 'primera'. El nodo se aplila antes en estado 'segunda'.
- 'segunda': A punto de visitar/apilar el hijo derecho del nodo en estado 'primera'. El nodo se aplila antes en estado 'tercera'.
- 'tercera': A punto de procesar nodo. El nodo ya no va ha ser más necesitado.

```
Postorden(nodoRaíz):
    Stack.push(nodoRaíz, 'primera')
    do:
       nodo, estado = Stack.pop()
       switch(estado):
          case 'primera':
            if nodo.izquierdo not null:
              Stack.push(nodo, 'segunda')
              Stack.push(nodo.izquierdo, 'primera')
              break
          case 'segunda':
            if nodo.derecho not null:
              Stack.push(nodo, 'segunda')
              Stack.push(nodo.derecho, 'primera')
              break
          case 'tercera':
            procesar(nodo)
            break
       fin-switch
    while( not Stack.isEmpty() )
return
```

16.1.3 Orden simétrico



Recorrido en orden simétrico: Se lleva a cabo el procesado sobre su hijo Izquierdo, luego se procesa sobre el nodo, antes de hacerlo sobre su hijo derecho.

```
Enorden(nodo):
    if nodo.izquierdo:
       Enorden(nodo.izquierdo)
    procesar(nodo)
    if nodo.derecho:
       Enorden(nodo.derecho)
```

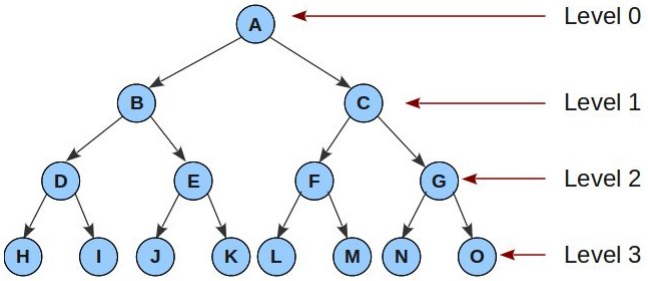
Algoritmo no recursivo: Cada nodo es visitado 2 veces, es decir apilado y desapilado 2 veces.

Si el nodo es desapilado con el estado:

- 'primera': A punto de visitar/apilar el hijo izquierdo del nodo en estado 'primera'. El nodo se aplila antes en estado 'segunda'.
- 'segunda': A punto de procesar el nodo, luego el nodo ya no va ha ser más necesitado y se visita/apila el hijo derecho del nodo en estado 'primera'.

```
Postorden(nodoRaíz):
    Stack.push(nodoRaíz, 'primera')
    do:
       nodo, estado = Stack.pop()
       switch(estado):
          case 'primera':
            if nodo.izquierdo not null:
              Stack.push(nodo, 'segunda')
              Stack.push(nodo.izquierdo, 'primera')
              break
          case 'segunda':
            procesar(nodo)
            if nodo.derecho not null:
              Stack.push(nodo.derecho, 'primera')
              break
       fin-switch
    while( not Stack.isEmpty() )
return
```

16.1.4 Por niveles



Depth fo the above tree (d) = 3

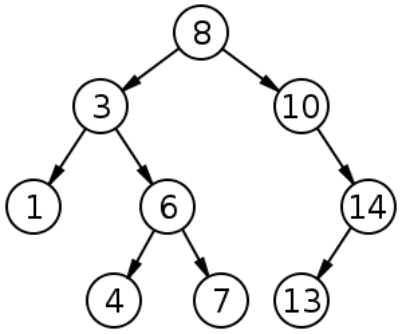
Recorrido del Árbol por niveles en forma no recursiva: El procesado comienza en la raíz, luego sus hijos y luego los nietos, va por niveles de profundidad. Para ello se emplea una cola, porque hay que que ir en orden.

```
PorNivel(nodoRaíz):
    Queue.insert(nodoRaíz)
    do:
       nodo = Queue.quit()
       procesar(nodo)
       if nodo.izquierdo not null:
          Queue.insert(nodo.izquierdo)
       if nodo.derecho not null:
          Queue.insert(nodo.derecho)
    while( not Queue.isEmpty() )
return
```

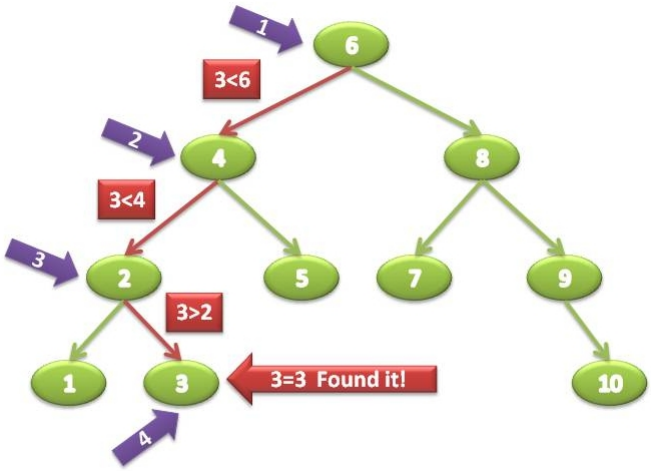
17 Árbol Binario de búsqueda

El Árbol Binario de búsqueda, es un árbol binario que satisface la propiedad estructural de búsqueda ordenada, la cual reza: Para cada nodo X del árbol, los valores de todas las claves del subárbol izquierdo, son menores a la clave del nodo X y los valores de todas las claves del subárbol derecho, son mayores a la clave del nodo X.

Por lo tanto, todas las claves del árbol están ordenadas en forma consistente al recorrido en orden simétrico. Dicha propiedad impide la existencia de elementos duplicados.



17.1 Búsqueda



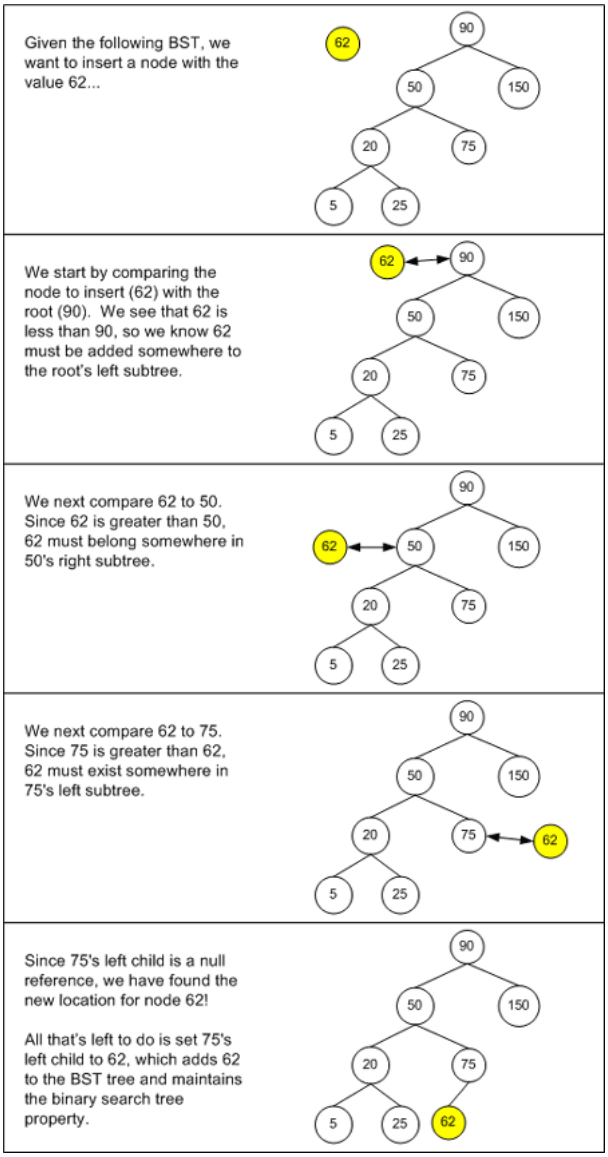
Operación: **Búsqueda de un elemento**: Se basa en la comparación de la clave buscada contra la clave de turno comenzando desde la raíz:

- Si la clave buscada es idéntica a la clave de turno se retorna el puntero al nodo.
- Si la clave buscada es menor a la clave de turno se sigue buscando en el subárbol izquierdo.
- Si la clave buscada es mayor a la clave de turno se sigue buscando en el subárbol derecho.
- Si el subárbol es vacío, retornar que no se ha encontrado el nodo.

```
Buscar(nodoDeTurno, claveB):
    if(nodoDeTurno not null):
       if(claveB == nodoDeTurno.clave):
          return nodoDeTurno
       if(claveB < nodoDeTurno.clave):

          return Buscar(nodoDeTurno.izquierdo, claveB)
       if(claveB > nodoDeTurno.clave):
          return Buscar(nodoDeTurno.derecho, claveB)
    return null
```

17.2 Inserción



Operación: **Inserción de elemento:**

- Si el árbol esta vacío, creamos un nuevo árbol con un único nodo.
- Si el árbol no esta vacío, Se basa en la comparación de la nueva clave contra la clave de turno comenzando desde la raíz:
- Si la nueva clave es idéntica a la clave de turno, no se hace nada.

- Si la nueva clave es menor a la clave de turno se sigue la inserción en el subárbol izquierdo.
- Si la nueva clave es mayor a la clave de turno se sigue la inserción en el subárbol derecho.
- Si el subárbol es vacío, crear el nodo y actualizar la referencia del padre.

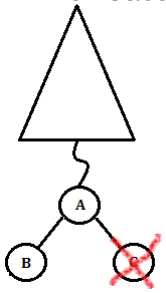
```
Insertar(nodoDeTurno, claveN):
    if(nodoDeTurno == null):
       nodoDeTurno = new NodoÁrbol(claveN)
       return
    if(claveN == nodoDeTurno.clave):
       return
    if(claveN < nodoDeTurno.clave):
       Insertar(nodoDeTurno.izquierdo, claveN)
       return
    if(claveN > nodoDeTurno.clave):
       Insertar(nodoDeTurno.derecho, claveN)
       return
```

17.3 Eliminación

Operación: **Eliminación de un nodo en el árbol:** En los árboles binarios son los nodos quienes mantienen conectado al árbol.

Una vez encontrado el nodo a eliminar puede pasar tres cosas:

- **Primer Caso:**



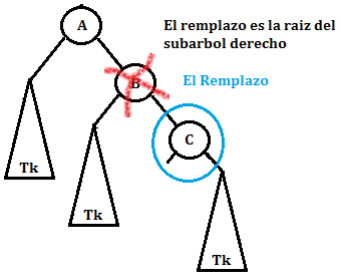
El nodo ha eliminar es una hoja, como no mantiene al árbol unido, simplemente averiguamos si es hijo hijo izquierdo o derecho , actualizamos la referencia en el padre y lo eliminamos.

- **Segundo Caso:**

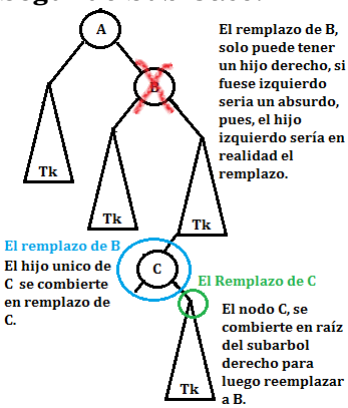
El nodo ha eliminar tiene un solo hijo, simplemente averiguamos si es hijo hijo izquierdo o derecho , actualizamos la referencia en el padre para que apunte al único hijo (esto restablecería la propiedad estructural del árbol binario) y lo eliminamos.

1. **Tercer Caso:**El nodo ha eliminar tiene dos hijos, para restablecer la propiedad estructural del árbol binario, solo hay dos posibles reemplazantes el menor elemento del subárbol derecho o el mayor elemento del subárbol izquierdo.
   Para simplificar: escogemos aquí siempre, el menor elemento del subárbol derecho.
   Esto genera 3 subcasos, que en la practica se transforman en un solo subcaso:

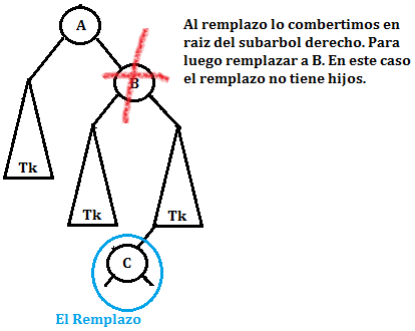
   - **Tercer Sub-Caso:**

     El nodo reemplazante es la raíz del subárbol derecho (es hijo del nodo que va ser eliminado).
     - Actualizamos la referencia izquierda del nodo que va a ser eliminado en la referencia izquierda nodo remplazante.
     - Actualizamos la referencia en el padre para que apunte al hijo derecho del reemplazante.
     - Eliminamos el nodo.
   - **Segundo Sub-Caso:**

     El nodo reemplazante no es la raíz del subárbol derecho (no es hijo del nodo ha ser eliminado) pero tiene hijo derecho:
     - Actualizamos la referencia en el padre del nodo reemplazante para que apunte al hijo derecho del remplazante.
     - Actualizamos la referencia derecha del nodo reemplazante para que apunte a la raiz del subarbol derecho. Por lo que ahora el nodo remplazante es la nueva raíz. (Convirtiéndose en el tercer subcaso).
   - **Primer Sub-Caso:**

     El nodo reemplazante no tiene hijos.
     - Actualizamos la referencia en el padre del nodo reemplazante para que apunte a null.

- Actualizamos la referencia derecha del nodo reemplazante para que apunte a la raíz del subárbol derecho. Por lo que ahora el nodo reemplazante es la nueva raíz. (Convirtiendose en el tercer subcaso).

El pseudo codigo sería:

```
#Retorna la referencia del minimo
getMinimo(nodo):
     if nodo.izquierdo not null:
         return getMinimo(nodo.izquierdo)
     return nodo

#Retorna la referencia del padre del nodo
getPadre(nodoDeTurno, nodoX):


     if nodoDeTurno.izquierdo.clave == nodoX.clave OR nodoDeTurno.derecho.clave == nodoX.clave:
         return nodoDeTurno
     if nodoDeTurno.clave > nodoX.clave:
         return getPadre(nodoDeTurno.izquierdo, nodoX)
     return getPadre(nodoDeTurno.izquierdo, nodoX)

#Trasplantar actualiza la referencia del padre del nodoA
#para que apunten al nodoB en vez de el nodoA
trasplantar(nodoRaíz, nodoA,nodoB):
     if nodoRaíz == nodoA:
         nodoRaíz = nodoB
         return
     padreDeA=getPadre(nodoRaíz, nodoA)
     if padreDeA.izquierdo == nodoA
         padreDeA.izquierdo = nodoB
         return
     padreDeA.derecho = nodoB
     return

#Elimina un nodo de un arbol
Eliminar(nodoRaíz, nodoPorEliminar):
     if nodoPorEliminar.izquierdo == null:

         #Supongo primer y segundo caso como uno (nodoPorEliminar.derecho podria ser null)
         trasplantar(nodoRaíz, nodoPorEliminar , nodoPorEliminar.derecho)
         return
     if nodoPorEliminar.derecho == null:
         #Supongo segundo caso: tiene hijo izquierdo
         trasplantar(nodoRaíz, nodoPorEliminar , nodoPorEliminar.izquierdo)
         return
     # Tercer Caso nodoPorEliminar tiene los dos hijos
     nodoMinimo = getMinimo(nodoPorEliminar.derecho)
     if nodoPorEliminar.derecho != nodoMinimo:

         #Supongo primer y segundo sub-caso como uno (nodoMinimo podria tener subárbol vacío)
         trasplantar(nodoRaíz, nodoMinimo , nodoMinimo.derecho)
         #El padre de nodo minimo apuntara al nodo minimo derecho.
         nodoMinimo.derecho = nodoPorEliminar.derecho
         #transformamos primer y segundo sub-caso en tercer subcaso
     # tercer subcaso-Caso nodoMinimo es raíz del sub arbol
     nodoMinimo.izquierdo = nodoPorEliminar.izquierdo
     trasplantar(nodoRaíz, nodoPorEliminar , nodonodoMinimo)
     delete-memoria(nodoPorEliminar)

return
```

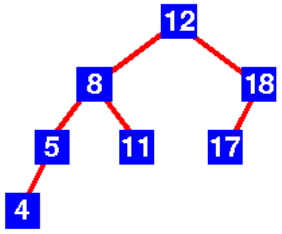
17.4 Ordenes

Los **Ordenes de las operaciones**: Los ordenes de las operaciones son proporcionales a la profundidad de los nodos con que se trabaja.

La propiedad estructural del orden no garantiza equilibrio.

Los running time de las operaciones son $\Theta(Profundidad)$. Para un árbol equilibrado las operaciones tienen un costo logaritmico, para un arbol totalmente degenerado las operaciones tienen un costo lineal , porque el árbol tiene forma de lista en lazada. El **costo medio** para casos aleatoreo es $O(N \log N)$.

18 Árboles AVL



Un árbol AVL, es un árbol binario de búsqueda con una propiedad estructural más:

La propiedad de equilibrio: La altura de los hijos izquierdo y derecho, puede diferir a lo sumo en una unidad.

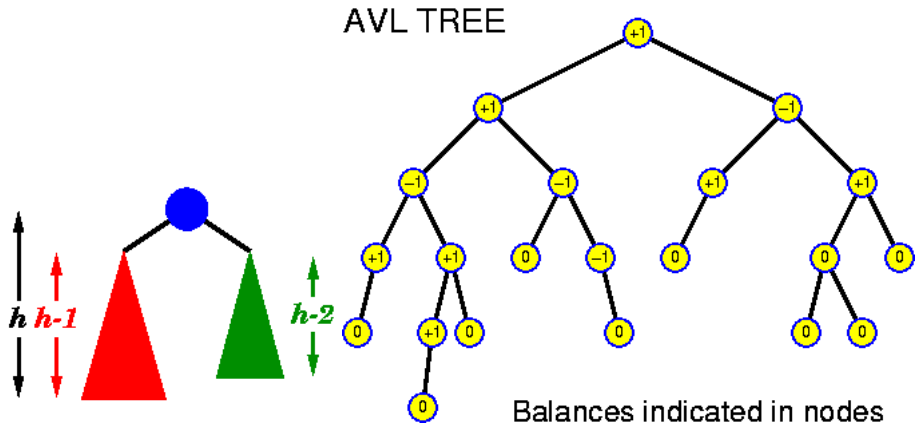
Tomamos la altura del árbol vacío como -1.

Esto implica tener siempre profundidad logaritmica; La forma del árbol (al intentar degenerar, crece en forma de fractal).

El número de nodos crece como mínimo exponencialmente con la altura.

Teorema: Un árbol AVL, de altura H tiene por lo menos $F(H+3)-1$ nodos, donde $F(i)$ es el i-esimo número de fibonacci.

La inserción y eliminación en el árbol AVL, puede destruir su propiedad estructural de equilibrio.



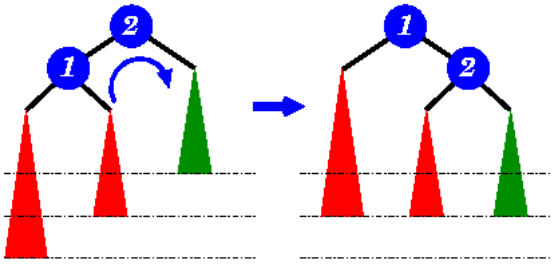
Tras realizar la inserción o eliminación común, puede violarse de 4 formas la propiedad estructural de equilibrio:

1. El subárbol izquierdo del hijo izquierdo del nodo X, es más profundo que el hijo derecho del nodo X
2. El subárbol derecho del hijo izquierdo del nodo X, es más profundo que el hijo derecho del nodo X
3. El subárbol izquierdo del hijo derecho del nodo X, es más profundo que el hijo izquierdo del nodo X
4. El subárbol derecho del hijo derecho del nodo X, es más profundo que el hijo izquierdo del nodo X

Rotación simple: Solo para caso 1 y 4: Intercambio los papeles del padre de los hijos del extremo que producen conflicto (o sea el nodo X) contra su hijo del extremo que produce conflicto. O sea que voy a:

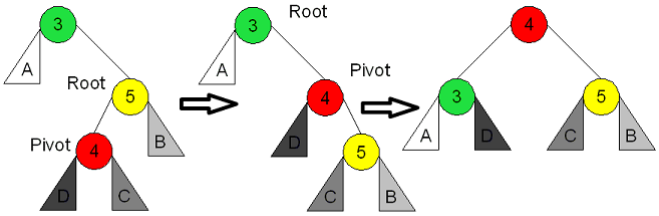
- Trasplantar el nodo X con el hijo de X extremo que produjo el conflicto.
- Remplazar el lazo que une con el hijo con el subárbol que no produce problemas
- Remplazar el lazo del hijo extremo que iba el subárbol que no producía problemas con el nodo X.

¿Resolviendo el problema de equilibrio para la posición donde estaba X, recuperando su vieja altura, resuelvo el problema hasta la raíz? Como queda el lazo con exactamente la misma altura que antes, no es necesario realizar otras actualizaciones hasta la raíz.



Rotación doble: Solo para caso 2 y 3: Es el equivalente a dos rotaciones simple: Intercambio los papeles de la raíz del subárbol que causa problemas con el nodo x: O sea que voy a:

- Trasplantar el nodo X con la raíz del subárbol interno que produjo el conflicto.
- Remplazar en X, el lazo que une con el hijo con el subárbol que produce problemas, con un subárbol de la raíz del subárbol problematico.
- Remplazar el lazo del hijo extremo del nodo X que va al subárbol problemático, con el otro subárbol de la raíz del subárbol problemático.
- Los hijos de la raíz del subárbol problematico, pasan a ser el nodo X y el hijo del nodo x.



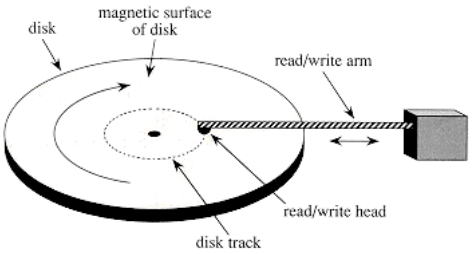
¿Resolviendo el problema de equilibrio para la posición donde estaba X, recuperando su vieja altura, resuelvo el problema hasta la raíz? Como queda el lazo con exactamente la misma altura que antes, no es necesario realizar otras actualizaciones hasta la raíz.

19 Árboles Red-Black

Para más información leer el Articulo: Árboles Rojo-Negros de Guido Urdaneta (14 pags).

20 Árboles B

El árbol B, surge ante la necesidad de resolver una serie de problemas ocasionados cuando los árboles ocupan mucho espacio. Cuando los árboles ocupan mucho espacio, el árbol no quepa en memoria principal, la mayor parte esta en la memoria secundaria.



El acceso a memoria secundaria tiene un costo de acceso es más lento en comparación a millones de instrucciones de procesador, por eso debemos ahorrar accesos a disco tanto como podamos. Si tenemos mayor grado de ramificación (un nodo tiene más hijos), tenemos menor altura, y si tenemos menor altura, recorremos menos nodos y por lo tanto menos accesos.

$$K_1 < K_2 < \cdots < K_{m-1}$$

$$N+1 \geq 2E^{n-1}\left(\frac{m+1}{2}\right)$$

$$n \leq 1+\log_{\frac{m+1}{2}}\left(\frac{N+1}{2}\right)$$

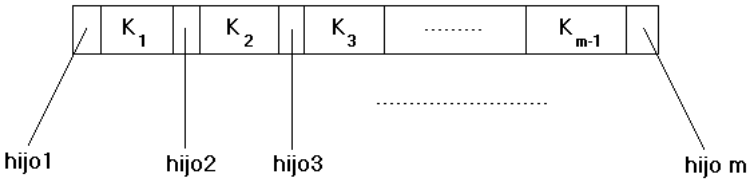
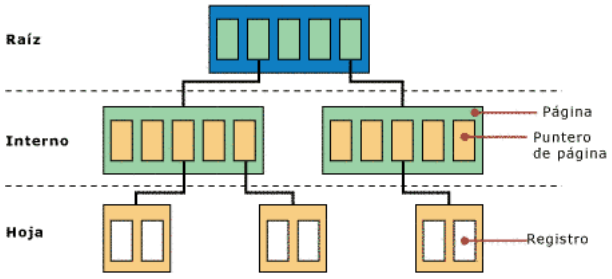
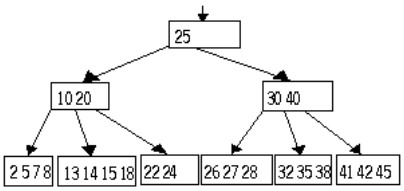


Figura 1:Esquema de un nodo de un arbol B de orden m.

Se llama **árbol m-ario**, aquel árbol que el nodo tiene de 1 a m hijos. Para poder efectuar una búsqueda ordenada se le añade la propiedad de orden. Cada nodo del Árbol m-ario tiene m-1 claves para decidir cual es el próximo nodo a visitar. Efectuar la comparación contra las m-1 claves es algo estúpido frente a la espera de un acceso a disco. Cómo el árbol m-ario se puede degenerar, aparece el árbol B.





20.1 Reglas

El árbol B, es árbol m-ario con una propiedad estructural, **las reglas a cumplir son:**

1. Los datos se almacenan solo en las hojas.
2. Los nodos internos tienen m-1 claves para guiar la búsqueda. La clave i-esima es mayor que las claves de sus descendientes de la rama i-1. La clave i-esima es menor que las claves de sus descendientes de la rama i+1.
3. La raíz es una hoja o tiene entre 2 y m hijos. Es un nodo especial a comparación de los nodos internos.
4. Los nodos internos tienen entre $\lceil \frac{M}{2} \rceil$ y m hijos.
5. Todas las hojas se encuentran en la misma profundidad. y tienen entre$\lceil \frac{L}{2} \rceil$ y L datos.

L y M no poseen un valor arbitrario, el valor es el que minimiza los accesos.
Datos a conocer:

- Tamaño de bloque de disco.
- Tamaño de una rama del árbol, es decir el tamaño de la referencia o puntero al proximo nodo.
- Tamaño de la clave, es decir el tamaño de la variable que contiene la clave.
- Tamaño de cada dato o registro en un nodo hoja.

Como van a quepar L datos en una hoja:
$L(Tamañoregistro) \leq Tamañodebloquededisco$
Como un nodo interno tiene M ramas y m-1 claves:
$(M-1)(Tamañodelaclave) + M(Tamañorama) \leq Tamañodebloquededisco$

20.2 Ejemplo:

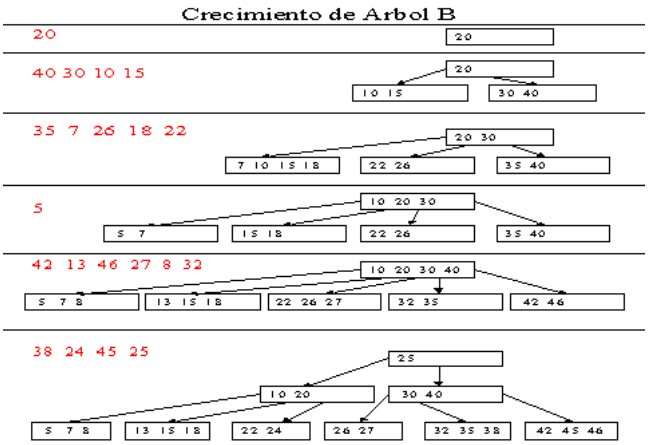
- Tamaño de bloque de disco = 8192 bytes.
- Tamaño de una rama = 4 bytes (un puntero).
- Tamaño de la clave = 32 bytes (un string).
- Tamaño de cada registro = 256 bytes.
- M = 228; L=32.

20.3 Algoritmo de inserción:

Algoritmo de inserción:

- Efectuamos la búsqueda hasta dar con la hoja donde correspondería la inserción.
- Si la hoja cuenta con un espacio, el algoritmo finaliza.
- Si la hoja esta llena, ahora posee L+1 elementos. En ese caso dividimos la hoja en 2 hojas de $\lceil \frac{L}{2} \rceil$elementos cada una.

- Esto generará 2 accesos al disco para escritura y uno para actualizar al padre (2 claves y 2 ramas).
- Si el padre posee un lugar libre el algoritmo finaliza.
- Si el padre esta completo ahora posee M claves (m+1 hijos). En ese caso dividimos el nodo en 2 nodos de $\lceil \frac{M-1}{2} \rceil$ elementos cada uno. Si era la raíz, creamos una nueva raíz con 2 hijos. En caso contrario actualizamos al padre del padre, así hasta finalizar.



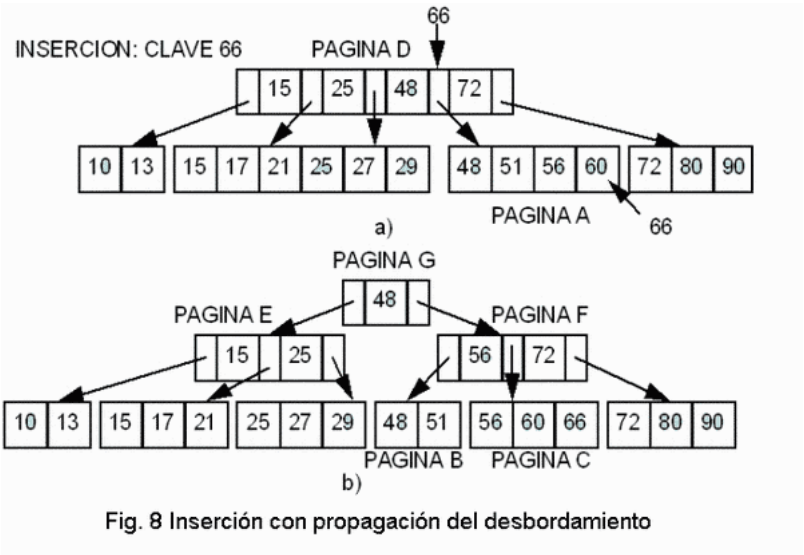
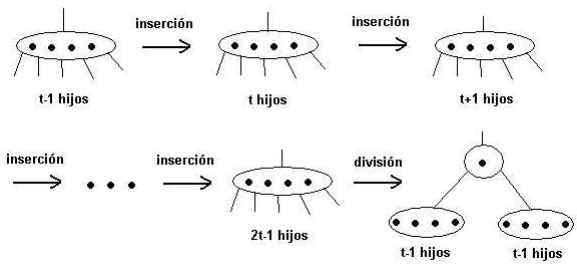


Fig. 8 Inserción con propagación del desbordamiento



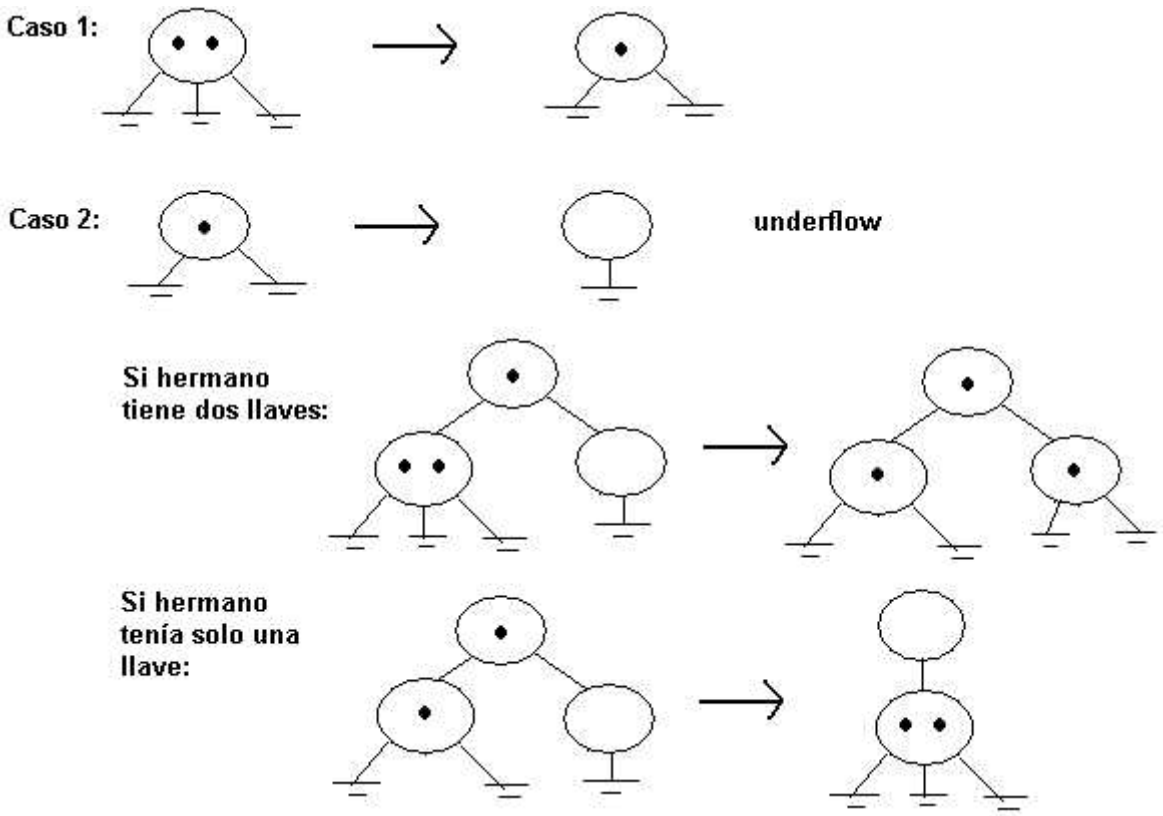
20.4 Algoritmo de eliminación:

Algoritmo de eliminación:

- Si la hoja tiene más de $\frac{L}{2}$ registros, el algoritmo finaliza. En caso contrario, ahora la hoja tiene menos de $\frac{L}{2}$ hijos, debe pedirle 1 registro a un hermano que tenga más de $\frac{L}{2}$ hijos. Si no pueden sus hermanos hacer eso, entonces puedo fusionar las dos hojas en

una sola. Después se le informa al padre que tiene un hijo menos y se le actualiza la clave.

- Ahora se repite la misma historia con los padres: Si el padre tiene más de $\frac{M-1}{2}$ claves, finaliza caso contrario, se le piden claves a un hermano. Si no las pueden ceder, se fusionan. Si la raíz queda con un solo hijo, se elimina y su hijo pasa a ser la nueva raíz.



Part VII
Tablas Hash y Diccionarios

21 Diccionario

Es una estructura de datos, en donde almaceno objetos y accedo a objetos , conociendo su nombre, preferentemente en forma de string. El diccionario, es implementado mediante una tabla hash.

22 Tabla Hash

La tabla hash, es un vector que es indexado mediante una función de localización, llamada función hash.

La **función hash**, es una regla que convierte un elemento (un objeto) a un entero adecuado para indexar al vector, donde allí finalmente se almacenará el elemento.

La eficiencia depende del grado de ocupación del vector.

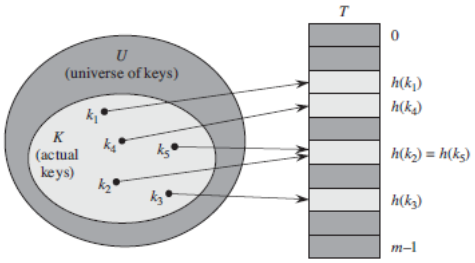


Figure 11.2 Using a hash function h to map keys to hash-table slots. Because keys k_2 and k_5 map to the same slot, they collide.

22.1 Sobre el vector y la función

Las entradas del vector no son infinitas, por lo tanto no podemos tener todas las combinaciones de entrada posibles.

Por ejemplo: si mi objeto tiene 4 bytes, tenemos 4 billones de combinaciones, y 4 billones de entradas es imposible. La función hash asocia un conjunto de partida más grande que el de llegada. Por el principio del palomar, (porque existen más elementos potenciales que posiciones,) podemos decir que es posible que dos elementos se correspondan en la misma posición, eso se denomina **colisión**. Debido a las colisiones, la función hash es inyectiva. La función hash debe ser sencilla de computar y distribuir las claves en forma uniforme, esto es para minimizar la cantidad de colisiones. Porque si existen colisiones, se debe efectuar un mecanismo de resolución de colisiones y la eficiencia baja.

El factor de carga de una tabla hash, es la fracción ocupada de la tabla. Denotamos al factor de carga con λ. Un $\lambda = 0$ para la tabla hash vacía y$\lambda = 1$ para la tabla hash llena.

$\lambda = \frac{CantidadDeEntradasTotal}{CantidadDeEntradasUsados}$.

Si la tabla es grande y los intentos de inserción son independientes, la fracción ocupada esλ. Al examinar una celda la probabilidad de estar ocupada es λ. porlo que el numero medio de celdas a examinar es $\lambda = \frac{1}{1-\lambda}$.

22.2 Mecanismo de resolución de colisiones por direccionamiento abierto

22.2.1 Exploración Lineal

En la rutina insertar, las colisiones se resuelven examinando secuencialmente el vector, con circularidad, en caso de ser necesario, hasta encontrar una posición vacía.

La rutina buscar, en caso de no conseguir el elemento buscado en la correspondiente celda, sigue exactamente el mismo camino que la rutina insertar, pues compara contra el elemento hallado y sigue en caso de no coincidir , si llega a una celda vacía se finaliza la búsqueda.

La eliminación estándar no puede usarse, cada elemento esta activo o borrado, esto se le denomina eliminación perezosa.

El numero medio de accesos:

- Para inserción y búsqueda sin éxito es igual a $\frac{1+\frac{1}{(1-\lambda)^2}}{2}$
- Para la búsqueda con éxito es igual a $\frac{1+\frac{1}{1-\lambda}}{2}$

Problema de la Exploración lineal: Agrupación primaria: Esto ocurre porque la independencia entre los elementos en la inserción no se cumple. Tenemos formaciones de grupos de celdas ocupadas. Haciendo que las inserciones dentro del grupo de celdas sea más costoso. y crezca el grupo, con cada elemento que se agrega.

22.2.2 Exploración Cuadrática

Elimina el problema de la agrupación primaria. Las colisiones se resuelven examinando con circularidad, la posiciones

$H, H+1, H+4, ..., H+i^2$

Garantía: Si el tamaño de la tabla es primo y el factor de carga no excede nunca el valor de 0,5. Entonces todos los intentos se realizan sobre celdas distintas. - Si el factor de carga es mayor a 0,5 entonces expandimos el tamaño de la tabla al próximo número primo. El proceso de expansión, se conoce como **re-hashing**. Incrementando la tabla y reinsertando los elementos mediante una nueva función hash. Es fácil encontrar el próximo número primo solo hay que examinar un orden de $O(\log N)$ números, con un test de primalidad de un orden de $O(N^{\frac{1}{2}})$ por lo que cuesta encontrarlo un orden de $O(N^{\frac{1}{2}} \log N)$.

Problema de la Exploración Cuadrática: Agrupación secundaria.

Esto ocurre porque los elementos indexados a la misma celda prueban las mismas celdas alternativas. Esto hace que se incremente el numero de celdas exploradas, en factores de carga grandes.

Para resolver este conflicto (Agrupación secundaria) se usa **El doble Hashing**, que es usar una segunda función de hashing, en la cual podemos examinar todas las celdas. $hash_1(), hash_2(), 2hash_2()$.

22.3 Mecanismo de resolución de colisiones por encadenamiento separado

El vector de la tabla hash es un vector de listas enlazadas $\{L_0, L_1, ..., L_{m-1}\}$. La función de hashing indica en que lista debemos insertar ese elemento. En este caso el factor de carga es igual a $\lambda = \frac{N}{M}$, la longitud media es λ, el numero medio de entradas usadas es λ. La búsqueda exitosa recorre $1 + \frac{\lambda}{2}$. La eficiencia no se ve afectada por el incremento del factor de carga, pudiendo así evitar el re-hashing.

Part VIII
Montículos Binarios, Colas de Prioridad y Ordenación Interna

23 Cola de Prioridad

Cola de Prioridad, es una estructura que permite la inserción de elementos, pero el acceso solo se otorga al elemento cuya prioridad es máxima. Es implementado a través de un montículo binario. El elemento con mayor prioridad es raíz del montículo.

24 Montículos binarios

Un montículo binario, es un árbol binario con una propiedad de orden y estructural diferente a los árboles vistos anteriormente.

24.1 Propiedad estructural:

El montículo binario es un **Árbol binario completo:**

- Es un árbol completamente lleno, con excepción del nivel inferior que debe llenarse de izquierda a derecha.
- Su altura es a lo sumo $\lfloor \log N \rfloor$
- Con altura H tiene 2^H o $2^{H+1} - 1$ nodos.
- La implementación estándar es la **representación implícita**, es decir, mediante un vector.
  - Cada nodo no tiene referencia a su hijo izquierdo ni al derecho.
  - De esta forma almacenamos los elementos por niveles en el vector, pero requerimos de un entero que nos indique cuantos nodos contiene el vector. Contando desde 1.
  - El hijo izquierdo del i esimo elemento, esta en la posición 2i.
  - El hijo derecho del i esimo elemento, esta en la posición 2i +1 .
  - El padre del i esimo elemento esta en $\lfloor \frac{i}{2} \rfloor$.
  - Todos los nodos salvo la raíz tienen padre.
  - La posición i=0, se considera dummy.
  - En caso obligado de empezar en 0
    - El hijo izquierdo del i esimo elemento, esta en la posición 2i+1.
    - El hijo derecho del i esimo elemento, esta en la posición 2i +2 .
    - El padre del i esimo elemento esta en $\lfloor \frac{i-1}{2} \rfloor$.

24.2 Propiedad de ordenación:

El montículo puede ser solo uno de los dos casos:

Montículo minimal La mínima clave la posee solo la raiz. Solo se da acceso al mínimo elemento, o sea solo a la raíz. Para cada nodo x con padre p, la calve p, es menor a x.

Montículo maximal La máxima clave la posee solo la raiz. Solo se da acceso al máximo elemento, o sea solo a la raíz. Para cada nodo x con padre p, la calve p, es mayor a x.

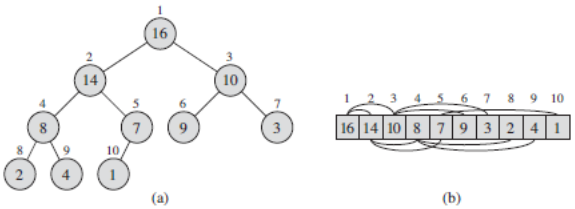


Figure 6.1 A max-heap viewed as **(a)** a binary tree and **(b)** an array. The number within the circle at each node in the tree is the value stored at that node. The number above a node is the corresponding index in the array. Above and below the array are lines showing parent-child relationships; parents are always to the left of their children. The tree has height three; the node at index 4 (with value 8) has height one.

La introducción de elementos no garantiza que se mantenga la propiedad de ordenación.

24.3 Inserción:

Se implementa creando un hueco en el siguiente lugar disponible. Donde colocamos allí el elemento nuevo, si violamos la propiedad de ordenación, intercambiamos el elemento con el padre, así hasta cumplir con la propiedad. Esa operación se denomina reflotar (shifup).

24.4 Eliminación:

Se implementa creando un hueco en la raíz, para cumplir con la propiedad de ordenación, el hueco por el mayor si es maximal, o el menor si es minimal. Los candidatos a llenar el hueco son sus dos hijos y el ultimo elemento del vector. De esta forma no pierde la propiedad estructural ni la de ordenación. El intercambio de la posición vacía con un elemento es llamada hundir (o shifdown).

24.5 Construcción del montículo con a partir de un vector con información ya puesta:

Para n elementos desde el elemento n/2 al primer elemento i=1, verificar la propiedad de orden con sus hijos.

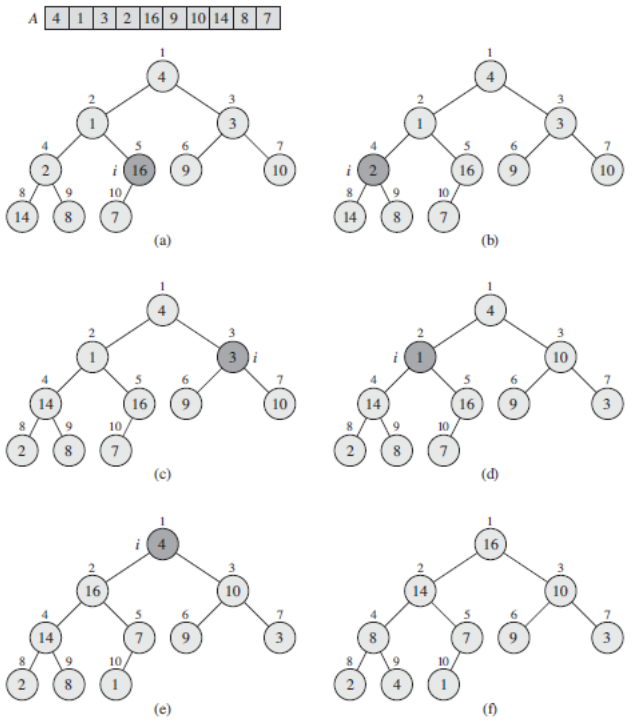


Figure 6.3 The operation of BUILD-MAX-HEAP, showing the data structure before the call to MAX-HEAPIFY in line 3 of BUILD-MAX-HEAP. **(a)** A 10-element input array A and the binary tree it represents. The figure shows that the loop index i refers to node 5 before the call MAX-HEAPIFY(A, i). **(b)** The data structure that results. The loop index i for the next iteration refers to node 4. **(c)–(e)** Subsequent iterations of the **for** loop in BUILD-MAX-HEAP. Observe that whenever MAX-HEAPIFY is called on a node, the two subtrees of that node are both max-heaps. **(f)** The max-heap after BUILD-MAX-HEAP finishes.

24.6 Búsqueda de un elemento y cambio de clave (conocido como reducción de clave, envejecimiento):

Aveces algunas aplicaciones deben establecer las prioridades en forma dinamica, por lo que deben buscar el elemento y hacerlo reflotar de ser necesario.

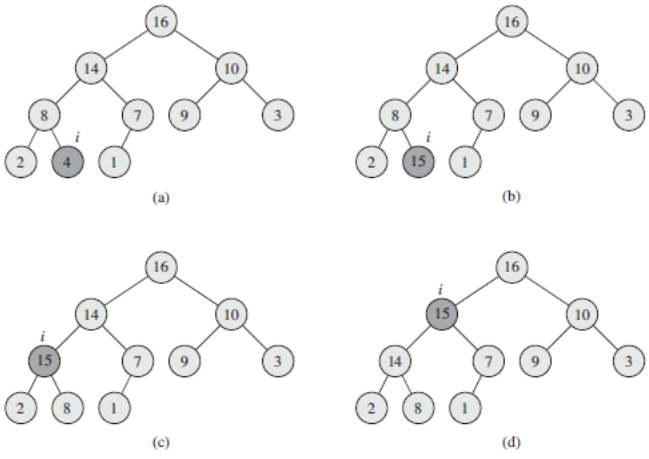


Figure 6.5 The operation of HEAP-INCREASE-KEY. **(a)** The max-heap of Figure 6.4(a) with a node whose index is i heavily shaded. **(b)** This node has its key increased to 15. **(c)** After one iteration of the **while** loop of lines 4–6, the node and its parent have exchanged keys, and the index i moves up to the parent. **(d)** The max-heap after one more iteration of the **while** loop. At this point, $A[\text{PARENT}(i)] \geq A[i]$. The max-heap property now holds and the procedure terminates.

25 Ordenación Interna

Consiste en tomar un vector de n elementos desordenado y ordenarlo mediante la utilización de un montículo:

- Haciendo del vector un montículo
- y eliminando al montículo n-veces el elemento de máxima prioridad.
- Cargando en un vector auxiliar los elementos que voy retirando.

26 Ordenación Heapsort

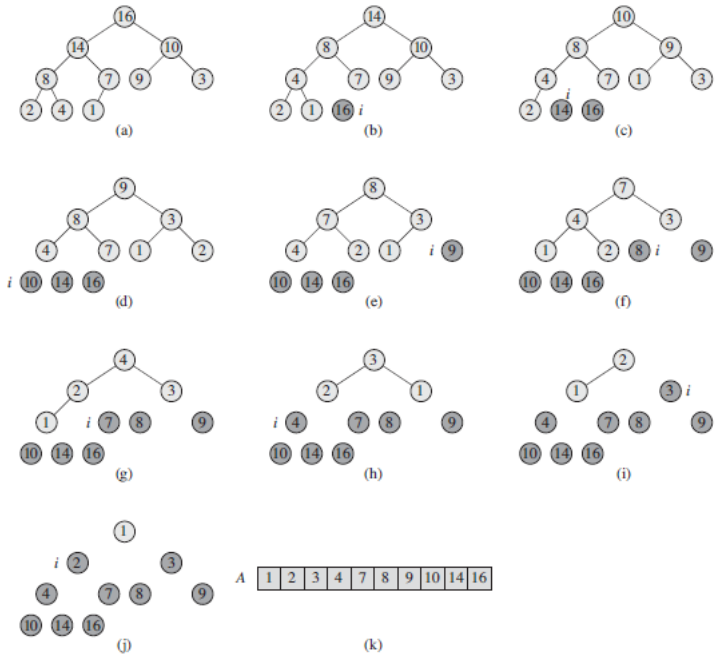


Figure 6.4 The operation of HEAPSORT. **(a)** The max-heap data structure just after BUILD-MAX-HEAP has built it in line 1. **(b)–(j)** The max-heap just after each call of MAX-HEAPIFY in line 5, showing the value of i at that time. Only lightly shaded nodes remain in the heap. **(k)** The resulting sorted array A.

Consiste en tomar un vector de n elementos desordenado y ordenarlo mediante la utilización de un montículo:

- Haciendo del vector un montículo maximal
- y eliminando al montículo n-veces el elemento máximo.
- No se requiere de un vector auxiliar, puesto que los elementos que voy retirando crean un espacio muerto al final del vector, que puedo utilizar colocando allí los elementos que voy retirando.

26.1 Análisis de Heapsort:

- Es in-place.
- Es basado en comparaciones
- y no es estable.

26.2 Análisis de los ordenes de Heapsort:

Las operaciones que se realizan son:

- 1 heapify (una Heapificación es la construcción de un montículo) que es de orden $T(N) = O(N)$
- N eliminaciones, que es de orden $T(N) = O(\log N)$.
- Por lo tanto Heapsort es de orden $T(N) = O(N \log N)$.

References

[1] Apuntes tomados en clases del Ing. Miguel Montes.

[2] Apuntes tomados de una copia de “Introduction to Algorithms” 3°ED de Cormen, Rivest, Stein y Leiserson.

[3] Apuntes tomados de una copia de “Estructuras de Datos en Java y Algoritmos” de Alan Waiss.

[4] Reconocimiento a foros: rohitab, code-makers.net y stackoverflow.

[5] Muchas paginas de Internet.