

Estructura de datos.

Wilson S. Tubín
wilsoneliseogt@gmail.com

Introducción

El documento contiene algo de algoritmia, arreglos, arboles, tablas de dispersión y textos. Todos temas vistos en el curso de estructuras de datos de la carrera de Ingeniería En Ciencias Y Sistemas de la Universidad De San Carlos De Guatemala.

Algoritmia: Estudio de los algoritmos

Arreglos: Estudio de representaciones y algoritmos de manejo de arreglos

Árboles: Conceptos generales y representaciones de árboles

Tablas de dispersión: En esta unidad revisaremos otra estructura para uso de contenedores, que busca obtener el rendimiento de los arreglos lexicográficos y la flexibilidad en el uso de memoria de los árboles.

Textos: Estudio de las diferentes representaciones y aplicaciones del manejo de textos o strings.

Ejercicios de examen: Alguno ejercicios resueltos de exámen. Otros solo se presentan, más carecen de solución.

Capítulo 1

Algoritmia

Estudio de los algoritmos

1.1. Eficiencia de algoritmos

1.1.1. Necesidad de la algoritmia

Dado un problema cualquiera que se desee implementar, habrán casi tantas soluciones efectivas, como programadores trabajen en ella. Sin embargo sólo habrá una solución, que sea la más eficiente. Ésto se debe a que parte de la programación involucra arte y creatividad, y cada persona piensa de manera diferente.

Dado que siempre estamos interesados en reutilizar lo existente, si buscamos una solución para un problema, seguramente encontraremos varias implementaciones, y deberemos elegir una de ellas como la más eficiente para utilizar.

Para ésto, primero tenemos tener claro la diferencia entre efectivo y eficiente:

Efectividad: Capacidad de lograr el efecto que se desea o se espera.

Eficiencia: Capacidad para lograr un fin, empleando los medios más adecuados

Claro, definir «los medios más adecuados» depende de la solución y lo que se declare importante. Por ejemplo: si se desea ir al país vecino ¿cuál es el medio más adecuado para llegar? ¿por bus, por avión, caminando? Tomar la decisión del medio, dependerá de qué lo importante que se desee lograr. Si tenemos tiempo y nos gusta disfrutar el paisaje, el mejor medio es el bus, si tenemos un negocio importante que nos perderíamos si no llegamos a tiempo, el medio adecuado sería el avión.

1.1.2. Eficiencia de algoritmos

Para lograr comparar dos algoritmos tenemos dos opciones:

Método empírico (a posteriori): Consiste en realizar las dos implementaciones y realizar ejecuciones similares, en la misma máquina y con la misma cantidad de datos, y comparar los tiempos de ejecución. Este método es la medida más exacta que se puede lograr, pero tiene la desventaja es que no siempre hay tiempo para hacer varias implementaciones o, si ya están hechas, hacer ambas implementaciones puede ser compleja y no ser factible.

Método teórico (a priori): Consiste en realizar un análisis matemático de los algoritmos y determinar una medida de la eficiencia. Ésto tiene la ventaja que no necesita una implementación real de los algoritmos, ya que basta con un esbozo de la implementación para realizar el análisis.

En el caso de los algoritmos, los medios para lograr la eficiencia se refieren a la utilización de tiempo y memoria, los cuales se tienen que equilibrar, y nos referimos a ellos en el estudio de **complejidad espacial** (eficiencia en el uso de la memoria) y **complejidad temporal** (eficiencia en el tiempo de ejecución).

La **complejidad espacial** se refiere a cuánta memoria necesita un algoritmo para ejecutarse completamente. Aunque actualmente se tiene la impresión que no es tan determinante para la construcción de un algoritmo como solía ser, sigue siendo un factor a tomar en cuenta al momento de implementar una solución, principalmente en ambientes concurrentes, como soluciones web, o en ambientes más restringidos como las soluciones móviles. En el ambiente web, un servicio relativamente simple, si tiene alta complejidad espacial (usa mucha memoria), al atender a miles de clientes, puede colapsar cualquier servidor. En un ambiente móvil, la memoria es más limitada que en una PC o servidor, por lo que se debe tener presente la cantidad de memoria a utilizar.

Anteriormente, era crítico el análisis de complejidad espacial, ya que si un programa no cabía en memoria (programa más datos) no era posible ejecutarlo. Actualmente, esto ya no es restricción debido al uso de la memoria virtual, pero ya que ésta involucra la utilización de disco duro, igualmente se debe vigilar el uso de la memoria para no degradar el rendimiento del sistema. Simplificando a niveles prácticos el análisis de la complejidad espacial, básicamente podemos decir que se trata de determinar el tamaño del ejecutable, adicionado a cuánta memoria necesita cada elemento de una estructura, multiplicado por el máximo número de elementos, y esto lo comparamos con la memoria disponible.

$$\text{memoria libre} \leq \text{tamaño_ejecutable} + \text{tamaño_elemento} * \text{máximo_elementos}$$

En cada estructura que estudiemos, haremos esta comparación, como un hábito que debe tener cada programador.

En el caso de la **complejidad temporal**, sí se requiere un análisis más detenido para llegar a una conclusión ¿cómo podemos analizar teóricamente el tiempo de ejecución sin ejecutar el algoritmo? ¿en qué computadora se realizará? dado que diferentes computadoras utilizan diferentes velocidades de ejecución ¿qué unidad de tiempo utilizar? ¿milisegundo o nanosegundos?

Para realizar el análisis de un algoritmo, independientemente de la computadora a usar, vamos a utilizar el siguiente principio:

Principio de la invarianza: Si dos implementaciones tardan respectivamente $T_1(n)$ y $T_2(n)$, entonces para resolver un problema con datos de tamaño n , existirá siempre una constante positiva c tal que $T_1(n) \leq c * T_2(n)$ para un n suficientemente grande

Esto significa que $T_2(n)$ será c veces más rápido que $T_1(n)$, sin importar la computadora. Si en una máquina $T_1(n) = 5$ segundos y $T_2(n) = 1$ segundo, en

otra máquina más rápida $T_1(n) = 5ms$ y $T_2(n) = 1ms$. En otras palabras $T_2(n)$ siempre más rápido que $T_1(n)$, en cualquier máquina.

Por lo tanto no necesitamos una unidad de tiempo específica para comparación, ya que nuestro análisis es independiente de la máquina. Usaremos una unidad de tiempo adimensional que denominaremos **tiempo mínimo** denotado por t que es el tiempo en el que cualquier procesador ejecuta la instrucción más simple, como una asignación o expresión matemática.

Adicionalmente, la unidad de tiempo es irrelevante ya que la medida de eficiencia no es un número, sino una función:

Definición 1.1.1 Función de orden $O(n)$. Se dice que $f(n)$ es de orden $g(n)$ si y sólo si

$$f(n) \leq c * g(n) \quad \text{para todo } n \geq n_0$$

donde c y n_0 son constantes positivas independientes de n

Ejemplo 1. sea $f(n) = 3n^3 + 2n^2$. ¿Podemos decir que $O(f(n)) = n^3$? si tenemos $c = 5$ y $n_0 = 0$

se cumple que

$$f(n) \leq 5 * g(n) \quad \text{para todo } n \geq 0$$

$$\begin{aligned} 3n^3 + 2n^2 &\leq 5n^3 \\ 3n^3 + 2n^2 &\leq 3n^3 + 2n^3 \\ 2n^2 &\leq 2n^3 \\ n^2 &\leq n^3 \end{aligned}$$

para todo $n \geq 0$

Ejemplo 2. sea $f(n) = 3n + 1$. ¿Podemos decir que $O(f(n)) = g(n) = n$? si tenemos $c = 4$ y $n_0 = 1$

se cumple que

$$f(n) \leq 4 * g(n) \quad \text{para todo } n \geq 1$$

$$\begin{aligned} 3n + 1 &\leq 4n \\ 3n + 1 &\leq 3n + n \\ 1 &\leq n \end{aligned}$$

para todo $n \geq 1$

En nuestros análisis no podremos hacer una hipótesis y comprobarla, como en los ejemplos anteriores, sino que dada la función $f(n)$, deberemos deducir $g(n)$. Ésto podremos hacerlo usando las siguientes propiedades de la función $O(n)$, cuya deducción está fuera del alcance del curso:

Propiedades de $O(n)$

- las constantes no importan: $O[c * f(n)] = c * O[f(n)] = O[f(n)]$
- Regla de la suma:
 - $O[f(n) + t(n)] = \max [O(f(n)), O(t(n))]$
 - $O[f(n)] + O[t(n)] = O [f(n) + t(n)]$
- regla de la multiplicación: $O[f(n)] * O[t(n)] = O[f(n) * t(n)]$
- anidación: $O [O(f(n))] = O [f(n)]$

De los ejemplos anteriores podemos deducir $O(n)$, aplicando estas propiedades así:

$O(3n^3 + 2n^2)$	<i>Regla de la suma :</i> $O(3n^3 + 2n^2) = \max(O(3n^3), O(2n^2)) = O(3n^3)$ <i>Regla de las constantes :</i> $O(3n^3) = O(n^3) = n^3$
$O(3n + 1)$	<i>Regla de la suma :</i> $O(3n + 1) = \max(O(3n), O(1)) = O(3n)$ <i>Regla de las constantes :</i> $O(3n) = O(n) = n$

En general, los resultados de $O(n)$ se pueden clasificar en los siguientes órdenes (del mejor al peor):

- constante: $O(f(n)) = c$, donde c es una constante
- lineal: $O(f(n)) = n$
- logarítmica: $O(f(n)) = \log_c(n)$
- progresiva, geométrica o polinomial: $O(f(n)) = n^c$, $c \geq 2$. Incluye cuadrático (n^2) y cúbico (n^3)
- Exponencial: $O(f(n)) = c^n$, para $c > 1$

Los dos últimos órdenes son especialmente deficientes, y en cualquier lado que los encontremos, debemos esforzarnos en mejorar el rendimiento.

1.2. Eficiencia de algoritmos no recursivos

Para determinar la eficiencia de un algoritmo debemos realizar los siguientes pasos:

1. Determinar una función del tiempo de ejecución del algoritmo para n datos: $T(\text{algoritmo}(n)) = T(n)$

2. Aplicar las propiedades de $O(n)$ para deducir $O(T(n))$

Dado el software es matemática, tal como demostró Allan Turing, entonces todo algoritmo tiene una expresión matemática y viceversa, es posible hacer una expresión matemática, $T(n)$, de cualquier algoritmo .

Para ésto utilizaremos los principios del teorema de la programación estructurada, la cual establece que todo algoritmo puede ser realizado utilizando sólo las siguientes instrucciones básicas:

- Asignaciones y expresiones simples
- Secuencia
- Condición
- Ciclos

Si podemos establecer una función de tiempo para cada una de estas sentencias, entonces podremos deducir dicha función para el algoritmo completo.

Recordar que usamos medidas de tiempo teóricas (adimensionales) = t = tiempo de la instrucción más simple o rápida de ejecutar en el procesador

1.2.1. Asignaciones y expresiones simples

$$\begin{aligned} T(\text{asignacion}) &= T(\text{expresion simple}) = t \\ \mapsto O(\text{asignacion}) &= 1 \text{ (orden constante)} \end{aligned}$$

Ésto incluye sentencias como

$$\begin{aligned} x &= y; \\ y &= 5 + y * z; \end{aligned}$$

1.2.2. Secuencia de instrucciones

$$\begin{aligned} T(\text{secuencia}) &= T(\text{sentencia1}) + T(\text{sentencia2}) + \dots + T(\text{sentencia } n) \\ \mapsto O(\text{secuencia}) &= O[T(\text{sentencia1}) + T(\text{sentencia2}) + \dots + T(\text{sentencia } n)] \\ &= \text{Max} \left[O(T(\text{sentencia1})), O(T(\text{sentencia2})) + \dots + O(T(\text{sentencia } n)) \right] \end{aligned}$$

1.2.3. Instrucciones condicionales

```
if ( condicion ) {
    sentenciaThen
}else{
    sentenciaElse
}
```

$$T(if) = T(condicion) + \max[T(sentenciaThen), T(sentenciaElse)]$$

$$\begin{aligned} \mapsto O(if) &= O[T(condicion) + \max(T(sentenciaThen), T(sentenciaElse))] \\ &= \max[O(T(condicion)), O(T(sentenciaThen)), O(T(sentenciaElse))] \end{aligned}$$

1.2.4. Instrucciones de iteración (for - while)

for

```
for (asignacionInicial ; condicion ; asignacionFinal)
    sentenciaCiclo
}
```

$$T(for) = T(asignacionInicial) + [T(condicion) + T(sentenciaCiclo) + T(asignacionFinal)] * v$$

donde **v** es el número máximo de veces que se repite el ciclo

while

```
i = 0 ; // asignacion inicial
while ( condicion ) {
    sentenciaCiclo
}
```

$$T(while) = T(asignacionInicial) + [T(condicion) + T(sentenciaCiclo)] * v$$

donde **v** es el número máximo de veces que se repite el ciclo en el peor de los casos

$$\begin{aligned} \mapsto O(ciclo) &= O[T(asignacionInicial) + [T(condicion) + T(sentenciaCiclo)] * v] \\ &= \max[O(T(asignacionInicial)), O((T(condicion) + T(sentenciaCiclo)) * v)] \end{aligned}$$

En estos casos, lo complicado suele ser determinar **v** en términos de **n** (**v** = **f(n)**)

1.2.5. Llamadas a procedimientos

La llamada en sí misma es de tiempo constante, pero toma como tiempo de la llamada, el tiempo de la ejecución completa

Está determinado por el cuerpo del procedimiento, dado que el paso de parámetros es constante (igual que una asignación)

1.2.6. Ejemplos

1. `x = x + 1 ;`

$$\begin{aligned} T(n) &= t \\ \mapsto O(\text{asignacion}) &= O[T(n)] = O(1) = 1 \text{ (constante)} \end{aligned}$$

2. `for (i = 1; i <= n; i++)
 x = x + 1 ;`

$$T(\text{for}) = T(\text{asignacionInicial}) + v * [T(\text{condicion}) + T(\text{cuerpoDelFor}) + T(\text{asignacionFinal})]$$

En este caso

$$T(\text{for}) = t + n(t + t + t) = t + 3nt = t(3n + 1)$$

$$\begin{aligned} \mapsto O[T(\text{for})] &= O[t(3n + 1)] \\ &= O(t) * O(3n + 1) \\ &= O(1) * \max[O(3n), O(1)] \\ &= O(3n) \\ &= O(n) = n \text{ (lineal)} \end{aligned}$$

3. `int A (int n) {
 x=1 ;
 while (x<n)
 x= 2 * x ;
 return x ;
}`

$$T(A) = T(\text{asignacion}) + T(\text{while}) + T(\text{asignacion})$$

$$T(A) = t + [T(\text{condicion}) + T(\text{cuerpoDelWhile})] * v + t$$

v =número de veces que se ejecuta el cuerpo y condición del while.

¿cuánto es v en términos de n ¹?

Podemos hacer un algoritmo equivalente

¹ n representa el numero de datos que recibe el argoritmo

```

int A1(int n) {
    x=1 ;
    v=0 ;
    while (x < n) {
        x = 2 * x ;
        v = v + 1 ;
    }
    cout << "v=" << v ;
    return x;
}

```

Valores de v y x respecto a n

n	v	x
1	0	$x=1/v=0$
2	1	$x=2/v=1$
3	2	$x=4/v=2$
4	2	
5	3	$x=8/v=3$
6	3	
7	3	
8	3	
9	4	$x=16/v=4$
10	4	
.	.	
16	4	
17	5	$x=32/v=5$
..	..	
32	5	
33	6	$x=64/v=6$
..	..	
64	6	

Podemos deducir que existe una relación entre v y n así:

$$2^v \geq n$$

por tanto

$$\begin{aligned}
 \log_2(2^v) &\geq \log_2(n) \\
 v &\geq \log_2(n)
 \end{aligned}$$

la función del tiempo sería

$$\begin{aligned}
 T[A(n)] &= T(\text{asignacion}) + T(\text{while}) \\
 &= t + [T(\text{condicion}) + T(\text{cuerpoDelWhile})] * \log_2(n) \\
 &= t + \log_2(n) * (t + t) \\
 &= t + 2t * \log_2(n)
 \end{aligned}$$

la función $O(n)$ sería

$$\begin{aligned}
 \mapsto O[A(n)] &= O[1 + 2 * \log_2(n)] \\
 &= \log_2(n) \quad (\text{logaritmico})
 \end{aligned}$$

1.3. Eficiencia de algoritmos recursivos

Para calcular el tiempo de un algoritmo recursivo, básicamente es el mismo procedimiento, pero tenemos que tener en cuenta los siguientes cambios:

- Dado que es un algoritmo recursivo, la función de tiempo también será recursiva
- Identificar la condición de salida del algoritmo, para que la función recursiva de tiempo, sea dual, con una parte no recursiva, definida por la condición de salida
- Una vez definida la parte recursiva, se deberá expandir a modo de resolver sin recursión, por medio de expansiones sucesivas.

1.3.1. Ejemplo 1

```

int factorial (int n) {
    if ( n <= 0 )
        return 1 ;
    else
        return n * factorial (n-1) ;
}
    
```

La condición de salida es $n \leq 0$, por lo tanto nuestra función de tiempo será dual y recursiva así:

$$T[\text{factorial}(n)] = T(n)$$

$$T(n) = \begin{cases} \text{si } n \leq 0 : & T(\text{condicion}) + T(\text{asignacion}) = 2t \quad (\text{condición de salida}) \\ \text{si } n > 0 : & T(\text{return}) + T(\text{llamadaRecursiva}) = t + T(n - 1) \quad (\text{parte recursiva}) \end{cases}$$

Resolviendo la parte recursiva:

$$\begin{aligned}
 T(n) &= t + T(n-1) \\
 &= t + [t + T(n-2)] = 2t + T(n-2) \\
 &= 2t + [t + T(n-3)] = 3t + T(n-3) \\
 &= \dots
 \end{aligned}$$

hasta la k -ésima expansión tenemos que

$$T(n) = kt + T(n-k)$$

entonces, usando la condición de salida, llegamos hasta que $n-k=0$

$$\begin{aligned}
 k &= n \\
 \mapsto T(n) &= nt + T(0) \\
 &= nt + 2t \\
 &= (n+2)t
 \end{aligned}$$

$$\mapsto O[\text{factorial}(n)] = O[T(N)] = O(n+2) = n \rightarrow \text{lineal}$$

Otra forma de lograr el mismo resultado

```
int factorial (int n) {
    int fact = 1 ;
    for (int i=1; i<=n; i++)
        fact=fact*i ;
    return fact ;
}
```

También es de orden lineal $O(n) = n$

1.3.2. Ejemplo 2

```
int recursiva2 (int n) {
    if ( n <= 1)
        return 5 ;
    else
        return recursiva2 (n-1) + recursiva2 (n-1) ;
}
```

$$T[\text{recursiva2}(n)] = T(n)$$

$$T(n) = \begin{cases} \text{si } n \leq 1 : & T(\text{condicion}) + T(\text{asignacion}) = 2t \\ \text{si } n > 1 : & T(\text{return}) + T(\text{llamadaRecursiva1}) + T(\text{llamadaRecursiva2}) \\ & t + T(n-1) + T(n-1) \\ & t + 2T(n-1) \end{cases}$$

Expandiendo la parte recursiva

$$\begin{aligned} T(n) &= t + 2T(n-1) \\ &= t + 2[t + 2T(n-2)] = 3t + 4T(n-2) \\ &= 3t + 4[t + 2T(n-3)] = 7t + 8T(n-3) \\ &= \dots \end{aligned}$$

hasta la k-ésima expansión

$$T(n) = (2^k - 1)t + (2^k)T(n-k)$$

según la condición de salida, llegamos hasta que $n - k = 1 \rightarrow k = n - 1$

$$\begin{aligned} \mapsto T(n) &= (2^{n-1} - 1)t + 2^{n-1}T(1) \\ &= (2^{n-1} - 1)t + 2^{n-1}t \\ &= (2^{n-1})t - t + (2^{n-1})t \\ &= 2(2^{n-1})t - t \\ &= 2^n t - t \\ &= (2^n - 1)t \end{aligned}$$

Encontrando la funcion $O(n)$

$$\begin{aligned} \mapsto O[\text{recursiva2}(n)] &= O[(2^n - 1)t] \\ &= O(2^n - 1) * O(t) \\ &= \max[O(2^n), O(-1)] * 1 \\ &= 2^n * 1 \\ &= 2^n \rightarrow \text{exponencial} \end{aligned}$$

1.3.3. Ejemplo 3

Un algoritmo que multiplica el $O(\text{for})$ por el $O(f(n))$

```
for (i=1; i <= f(n) ; i ++ )
```

...

equivalente más eficiente

```
x = f(n) ;  
for (i=1; i <= x ; i ++ )..
```

En general

```
if (x < f(n) ) {  
    ...  
    y = f(n) * z ;  
    z = f(n) ;  
    ...  
}
```

Cambiar a:

```
f1 = f(n) ;  
  
if (x < f1 ) {  
    ...  
    y = f1 * z ;  
    z = f1 ;  
    ...  
}
```


Capítulo 2

Arreglos

Estudio de representaciones y algoritmos de manejo de arreglos

2.1. Conceptos generales de arreglos

2.1.1. Objetivo

En esta unidad, se busca que el estudiante pueda entender y comparar las distintas representaciones internas de arreglos, determinando para cada una:

- facilidad de manejo
- rapidez de acceso

2.1.2. Definición

Un arreglo es un sistema de elementos del mismo tipo, indizado por un sistema de coordenadas enteras, que permite acceder y alterar elementos individuales .

En la programación, los arreglos son de uso extendido, tal que en la mayoría de lenguajes de programación, los arreglos están incluidos entre los tipos primitivos, como los enteros y caracteres.

Ejemplos del uso de arreglos:

```
int arreglo [5] ;    // indizado 0..4
int matriz [5][10] ; // indizado 0..4 * 0..9
arregloPascal: array [11..20, 1..5] of integer ; // de tamaño 10 * 5
```

Acceso de los elementos:

```
arreglo[3] = 4 ,
matriz [4,9] = 5 ;
arregloPascal [12, 3] = 100 ;
```

2.1.3. Problema

En los ejemplos anteriores, vemos que la notación para acceder los elementos es relativamente simple. Sin embargo, para el compilador no es tan simple, ya que implica una serie de cálculos relativamente complejos. Aunque no tengamos problemas para conceptualizar un arreglo bidimensional (una matriz) o tridimensional (un cubo), estas estructuras deben ser puestos en memoria, la cual es unidimensional.

La razón principal, se debe a que cuando se declara un arreglo, la variable que representa dicho arreglo es en realidad una apuntador al **primer** elemento del arreglo, como es el caso de las variables *arreglo*, *matriz* y *arregloPascal* en los ejemplos anteriores, y cada una de las sentencias de acceso anteriores implica un cálculo basado sólo en la dirección del primer elemento y las dimensiones declaradas de cada arreglo. Notación

Para las secciones siguientes, utilizaremos la siguiente notación

parejas límite	$n_1..m_1 * n_2..m_2 * \dots * n_k..m_k$ donde $n_x \leq m_x$	Conjunto de límites, para cada dimensión del arreglo
elemento del arreglo	$A[i,j,k,..]$	Notación para acceder un elemento del arreglo, dada por las coordenadas
dirección de un elemento	$\&A[i,j,k,..]$	Dirección de un elemento individual
función de localización	$\text{Loc}(A[i,j,k,..])$	Fórmula que nos permite calcular la posición, dentro del arreglo, de un elemento indizado
dirección inicial	$\alpha = \&A[n_1, n_2, \dots, n_k]$	Dirección del primer elemento.

2.2. Arreglos lexicográficos

2.2.1. Almacenamiento lexicográfico

La primera representación que estudiaremos, que es la usual en los lenguajes de programación, es la que cada arreglo se es puesto en una sola región de memoria, de forma contigua, y se tiene que reservar la memoria para todos los posibles elementos del arreglo, aunque no se estén utilizando. De ahí el nombre de arreglos estáticos.

Como programadores, es fácil para nosotros abstraer estructuras como matrices y cubos de forma simple, con las siguientes instrucciones:

```
int matriz[4,5] ;
```

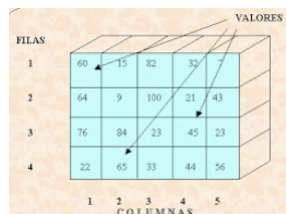
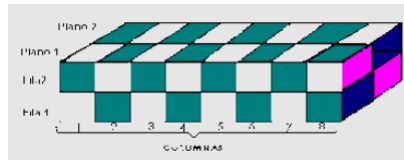


Diagrama de una matriz 4x5. Las filas están numeradas del 1 al 4 y las columnas del 1 al 5. Los valores en las celdas son:

FILAS \ COLUMNAS	1	2	3	4	5
1	60	15	82	39	7
2	64	9	100	21	43
3	76	84	23	45	23
4	22	65	33	44	56

```
int cubo[2,2,8]
```



Estas estructuras las podemos trabajar con instrucciones como:

```
matriz[3,2] = 0 ;
o
int x = cubo [1,2,4] * 3 ;
```

Sin embargo, debemos tomar en cuenta que, aunque el lenguaje nos permita realizar este acceso fácilmente, la memoria de la computadora no es multidimensional, y las matrices, cubos y arreglos de más de 3 dimensiones, deben poder mapear cada uno de los elementos, con una coordenada única (conjunto de índices).

Esto implica que cada elemento debe tener un orden definido, de forma que sea posible accederlo de forma inequívoca. Dicho orden es el orden lexicográfico:

Definición 2.2.1 Orden lexicográfico por fila: la posición (a_1, a_2, \dots, a_k) es menor que (b_1, b_2, \dots, b_k) , si y sólo si para algún $1 < j \leq k$, existe un i tal que $a_i = b_i$, para $1 \leq i < j$, y $a_j < b_j$.

Por ejemplo: si tenemos las coordenadas $A=(1, 5, 1, 9)$ y $B = (1, 5, 2, 8)$, el orden lexicográfico nos dice que $A < B$. En este caso $k = 4$, entonces necesitamos un número entre 1 y 4 (la variable j), que sea el índice de la coordenada tal que las coordenadas anteriores sean iguales en A y B , y a_j sea menor a b_j . En este caso, ese número j corresponde a 3. Nótese que en la primera y segunda coordenada tienen el mismo valor, y en la tercera se cumple que $1 < 2$.

Dado que tenemos las dimensiones de un arreglo, y su dirección inicial, es factible calcular la posición de cualquier elemento, dada sus coordenadas. Lo cual veremos del caso más simple al caso general.

2.2.2. Caso unidimensional

Supongamos el siguiente vector V de 4 elementos

$V : 1..4$

1	2	3	4
---	---	---	---

¿cuál sería la fórmula que nos permita mapear cualquier elemento de este vector? $Loc(V[i]) = ?$

Por ejemplo: encontrar el elemento $V[3]$. Aquí, la dirección del primero elemento es

$$\alpha = \&V[1]$$

y

$$Loc(V[3]) = \alpha + 2$$

Para encontrar el elemento $V[i]$

$$Loc(V[i]) = \alpha + i - 1$$

Si cambiamos la definición de V a

$V : 10..13$

10	11	12	13
----	----	----	----

aquí, la dirección del primero elemento es

$$\alpha = \&V[10]$$

Entonces, para encontrar el elemento $V[12]$, ya no nos funcionaría la fórmula anterior, dado que el índice inicial cambia de 1 a 10, por lo tanto tenemos que cambiar a:

$$Loc(V[12]) = \alpha + 12 - 10$$

En general si tenemos un vector V con pares límites arbitrarios así:

$V : n_1..m_1$

aquí, la dirección del primero elemento es

$$\alpha = \&V[n_1]$$

Entonces, la fórmula cambia así:

$$Loc(V[i]) = \alpha + i - n_1$$

2.2.3. Caso bidimensional

Para el caso de una matriz (arreglo bidimensional), de 3×4 definida como:

$M : 1..3 * 1..4$

aquí, la dirección del primero elemento es

$$\alpha = \&M[1, 1]$$

La conceptualizamos así:

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4

Pero en memoria, realmente queda así:

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

¿cuál sería la fórmula que nos permita mapear cualquier elemento de esta matriz? Si queremos encontrar el elemento $M[2,3]$.

$$Loc(M[2,3]) = \alpha + ?$$

Dado que el elemento está en la fila 2, sabemos que se debe avanzar totalmente la primera fila (avanzarAFila), y luego aplicamos al fórmula del vector, ya que cada fila es un vector. Así:

$$Loc(M[2,3]) = \alpha + avanzarAFila(2) + 3 - 1$$

De esta forma tenemos que avanzar 4 elementos, ya que cada fila es de tamaño 4.

$$Loc(M[2,3]) = \alpha + 1 * 4 + 3 - 1$$

Generalizando un poco más, si en esta matriz queremos encontrar un elemento arbitrario $A[i, j]$, aplicamos la fórmula así:

$$Loc(M[2,3]) = \alpha + avanzarAFila(i) + (j - 1)$$

Es decir debemos avanzar las filas necesarias para llegar a la fila i y luego dentro de esa fila aplicamos la fórmula para un vector.

En general $avanzarAFila(i)$, se compone de dos elementos: $FilasAntesDe(i) * TamanoDeCadaFila$, que en este caso son

$$\begin{aligned} Loc(M[2,3]) &= \alpha + FilasAntesDe(i) * TamanoDeCadaFila + (j - 1) \\ &= \alpha + (i - 1) * (4) + (j - 1) \end{aligned}$$

En general, si M tiene pares límites arbitrarios, definida así:

$$M : n_1..m_1 * n_2..m_2$$

aquí, la dirección del primero elemento es

$$\alpha = \&M[n_1, n_2]$$

y la fórmula cambia a:

$$\begin{aligned} Loc(A[i_1, i_2]) &= \alpha + avanzarAFila(i_1) + (i_2 - n_2) \\ &= \alpha + FilasAntesDe(i_1) * TamanoDeCadaFila + (i_2 - n_2) \end{aligned}$$

Dado que el límite inferior de la primera dimensión es n_1 entonces

$$FilasAntesDe(i_1) = i_1 - n_1$$

y, el tamaño de cada fila está dado por los límites de las columnas, entonces

$$TamanoDeCadaFila = m_2 - n_2 + 1$$

Por lo tanto, la fórmula queda

$$Loc(A[i_1, i_2]) = \alpha + (i_1 - n_1) * (m_2 - n_2 + 1) + (i_2 - n_2)$$

2.2.4. Caso tridimensional

Sea el cubo

$$C : 1..3 * 1..4 * 1..2$$

aquí, la dirección del primer elemento es

$$\alpha = \&C[1, 1, 1]$$

Tener en cuenta que en este caso, tenemos que cada índice en la primera dimensión representa una matriz y cada índice en la segunda dimensión, representa una fila de la matriz correspondiente, y la última dimensión representa un elemento individual.

Así, se se desea acceder el elemento $A[2,3,2]$, aplicamos la siguiente deducción

$$\begin{aligned} Loc(C[2, 3, 2]) &= \alpha + avanzarAMatriz(2) + AvanzarAFila(3) + (2 - 1) \\ &= \alpha + MatricesAntesDe(2) * TamanoDeCadaMatriz \\ &\quad + FilasAntesDe(3) * TamanoDeCadaFila + (1) \end{aligned}$$

Sabemos que

$$TamanoDeCadaMatriz = Filas * Columnas = 4 * 2 = 8$$

y

$$MatricesAntesDe(2) = 1$$

y

$$TamanoDeCadaFila = 2$$

y

$$FilasAntesDe(3) = 2$$

Por lo tanto

$$Loc(C[2, 3, 2]) = \alpha + (1) * 8 + (2) * 2 + (1) = \alpha + 13$$

En general, si C tiene pares límites arbitrarios, así:

$$C : n_1..m_1 * n_2..m_2 * n_3..m_3$$

aquí, la dirección del primero elemento es

$$\alpha = \&C[n_1, n_2, n_3]$$

y la fórmula cambia a:

$$\begin{aligned} Loc(C[i_1, i_2, i_3]) &= \alpha + avanzarAMatriz(i_1) + AvanzarAFila(i_2) \\ &\quad + AvanzarAlElemento(i_3) \\ &= \alpha + \textcolor{red}{MatricesAntesDe}(i_1) * \textcolor{blue}{TamanoDeCadaMatriz} \\ &\quad + \textcolor{blue}{FilasAntesDe}(i_2) * \textcolor{green}{TamanoDeCadaFila} + (i_3 - n_3) \\ &= \alpha + (\textcolor{red}{i_1} - n_1) * (\textcolor{blue}{m_2} - n_2 + 1) * (\textcolor{blue}{m_3} - n_3 + 1) \\ &\quad + (\textcolor{blue}{i_2} - n_2) * (\textcolor{green}{m_3} - n_3 + 1) + (i_3 - n_3) \end{aligned}$$

2.2.5. Caso k-dimensional

Sea el siguiente arreglo k-dimensional

$$A : n_1..m_1 * n_2..m_2 * \dots * n_k..m_k$$

aquí, la dirección del primero elemento es

$$\alpha = \&A[n_1, n_2, \dots, n_k]$$

Entonces deducimos la fórmula así:

$$\begin{aligned} Loc(A[i_1, i_2, \dots, i_k]) &= \alpha + ElementosDimension_1AntesDe(i_1) \\ &\quad + ElementosDimension_2AntesDe(i_2) + \dots \\ &\quad + ElementosDimension_{k-1}AntesDe(i_{k-1}) \\ &\quad + AvanzarAlElemento(i_k) \end{aligned}$$

Sabemos que

$$\text{ElementosDimension}_x \text{AntesDe}(i_x) = (i_x - n_x) * \text{TamanoDeCadaElementoEnDimension}_x$$

y para calcular el $\text{TamanoDeCadaElementoEnDimension}_x$, revisemos los casos anteriores:

Tamaño de cada elemento en la última dimensión:

$$1$$

Tamaño de la ultima fila (dimensión k-1):

$$m_k - n_k + 1$$

Tamaño de la última matriz (dimension k -2):

$$(m_{k-1} - n_{k-1} + 1) * (m_k - n_k + 1)$$

Tamaño del último cubo (dimensión k-3):

$$(m_{k-2} - n_{k-2} + 1) * (m_{k-1} - n_{k-1} + 1) * (m_k - n_k + 1)$$

...

Tamaño de la dimensión k-j:

$$(m_{k-j+1} - n_{k-j+1} + 1) * ... * (m_{k-1} - n_{k-1} + 1) * (m_k - n_k + 1)$$

...

Tamaño de la primera dimensión (cuando k-j=1 y k-j+1=2):

$$(m_2 - n_2 + 1) * ... * (m_{k-1} - n_{k-1} + 1) * (m_k - n_k + 1)$$

Por lo tanto:

$$\begin{aligned} \text{Loc}(A[i_1, i_2, \dots, i_k]) &= \alpha + (i_1 - n_1) * (m_2 - n_2 - 1) * ... * (m_{k-1} - n_{k-1} + 1) * (m_k - n_k + 1) \\ &\quad + (i_2 - n_2) * (m_3 - n_3 - 1) * ... * (m_{k-1} - n_{k-1} + 1) * (m_k - n_k + 1) \\ &\quad + (i_3 - n_3) * (m_4 - n_4 - 1) * ... * (m_{k-1} - n_{k-1} + 1) * (m_k - n_k + 1) \\ &\quad + + \\ &\quad + (i_{k-1} - n_{k-1}) * (m_k - n_k + 1) \\ &\quad + (i_k - n_k) \end{aligned}$$

$$\text{Loc}(A[i_1, i_2, \dots, i_k]) = \alpha + \sum_{x=1}^k \left[(i_x - n_x) \prod_{y=x+1}^k (m_y - n_y + 1) \right]$$

Ejemplo

Definición 2.2.2 Orden lexicográfico por columna. La posición (a_1, a_2, \dots, a_k) es menor que (b_1, b_2, \dots, b_k) , si y sólo si para algún $1 < j \leq k$, existe un i tal que $a_i = b_i$, para todo $j < i \leq k$, y $a_j < b_j$

Ej. Matriz : $1..3 * 1..4$

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4

Se almacena así:

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

sea A un arreglo de $n_1..m_1 * n_2..m_2$ ordenado lexicográficamente por columnas. Deducir $LOC(A[i, j]) = \alpha + (j - n_2) * (m_1 - n_1 + 1) + (i - n_1)$

En general, para el arreglo $A : n_1..m_1 * n_2..m_2 * \dots * n_k..m_k$

$$\begin{aligned}
 LOC(A[i_1, i_2, \dots, i_k]) &= \alpha + (i_k - n_k) * (m_1 - n_1 + 1) * (m_2 - n_2 + 1) * \dots * (m_{k-1} - n_{k-1} + 1) \\
 &\quad + (i_{k-1} - n_{k-1}) * (m_1 - n_1 + 1) * (m_2 - n_2 + 1) * \dots * (m_{k-2} - n_{k-2} + 1) + \\
 &\quad \dots \\
 &\quad (i_2 - n_2) * (m_1 - n_1 + 1) + \\
 &\quad (i_1 - n_1)
 \end{aligned}$$

2.3. Arreglos esparcidos

Vimos que el uso de arreglos lexicográficos tiene el mejor rendimiento posible (constante) para el acceso de elementos por índice. Sin embargo, el hecho de que necesiten reservar memoria contigua para todos sus posibles elementos, implica que existen situaciones en las que no son recomendables, debido principalmente a dos factores:

1. Desperdicio de memoria: consideremos una matriz de 1,000 por 1,000 elementos, de los cuales sólo se están utilizando 10. Sería un gran desperdicio de memoria, alojar 1,000,000 de elementos para sólo utilizar 10.
2. Incapacidad de alojar espacio para todos los posibles elementos: las hojas electrónicas tienen una gran capacidad de celdas, pero normalmente sólo se utiliza una pequeña porción de ellas: Excel tiene su máxima celda en IV65536, lo que significa una capacidad para $(26*8+22)*65536 = 15,073,280$ celdas. Si cada una de estas ocupara 1 byte, no tendríamos memoria suficiente para una hoja electrónica..

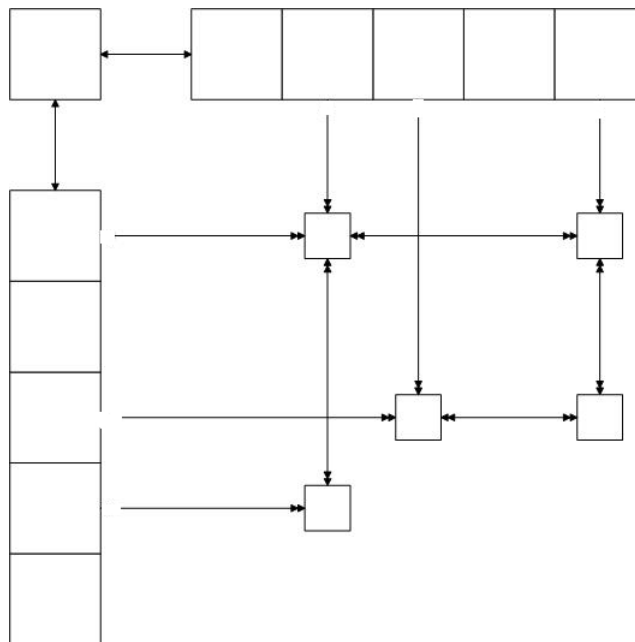
En cualquiera de estos casos resultan útiles los arreglos esparcidos, los cuales tienen como característica fundamental que **sólo utilizan la memoria realmente necesaria conforme se agregan elementos al arreglo.**

2.3.1. Técnicas de representación

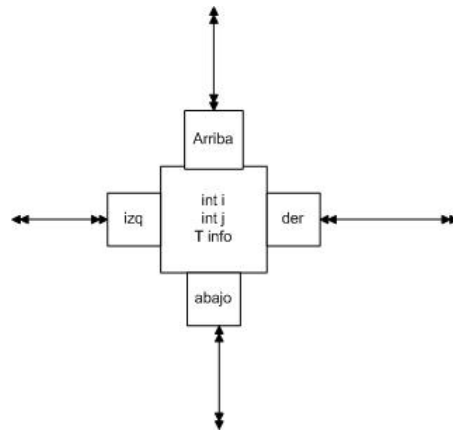
Entre muchas técnicas para lograr los arreglos esparcidos, distinguiremos dos principales:

- Arreglos lexicográficos de encabezados, con celdas de referencias cruzadas.
- Listas de encabezados

El uso de arreglos lexicográficos para los encabezados se ilustran en la siguiente figura:



Se tiene un control de encabezados, el cual contiene dos apuntadores: uno para el arreglo de las filas y el otro para las columnas. Los encabezados no necesitan más información que el apuntador al primer nodo de la fila o columna. Cada uno de los nodos tiene la siguiente estructura:



Los cuatro apuntadores de navegación nos sirven para facilitar los recorridos y búsqueda de cada nodo. Notemos que, además de la info, se debe tener en cada nodo la información de los índices que representan, ya que la característica de esparcido no permite calcular el índice de un nodo basado en sus posición de forma fácil. Se podría omitir esta información, para para buscar cada nodo implicaría una doble búsqueda, por fila y columna, para cada nodo, lo cual lo haría muy ineficiente.

La estrategia básica del algoritmo de localización de un elemento $A[i,j]$

Seleccionar el encabezado de las filas, aunque también puede ser por columna
*Para cada nodo en la lista de la fila **i***

```

Si nodo.j < j
    nodo = nodo.der ; //seguimos buscando
sino
    si nodo.j=j
        return nodo ; // encontramos el nodo
    sino
        return null ; // no se encuentra el nodo con los índices indicados
    fin
fin

```

Una implementación más formal, podría ser la siguiente:

```

public class ArregloEsparcido<E> extends ArregloGeneral<E> {
    .
    .
    .

    public E get (int i[]) {
        /* resumiendo a búsqueda por filas */
    }
}

```

```
        Nodo n = fila [i] ;

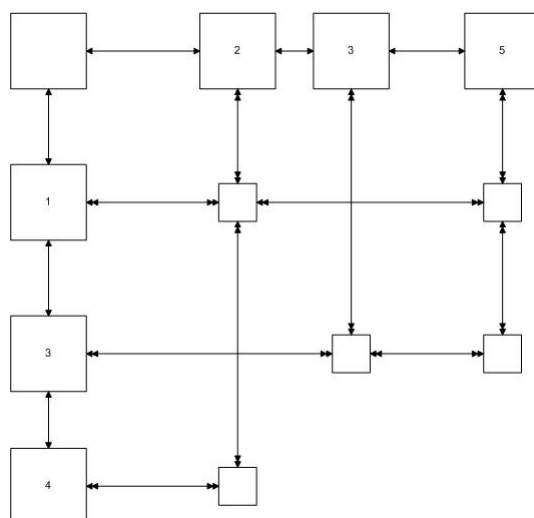
        while (n != null &&  n.col < j) {
            n = n.derecha ;
        }
        if ( n == null )
            return null ;
        else
            return n.obj ;
    }
}
.
```

Analizando el orden de este algoritmo:

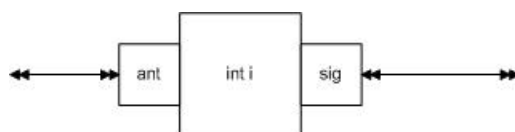
- El peor de los casos es una matriz de una solo fila, la cual está llena y estamos buscando el último elemento de dicha fila. En este caso el $O(\text{get}(i)) = n$, es decir lineal, ya que visitaremos todos los elementos.
- Si fuera una matriz cuadrada totalmente llena, entonces $n = \text{filas} * \text{columnas}$ y $\text{filas} = \text{columnas} = x$, por lo tanto $n = x^2$. Al hacer la búsqueda del último elemento de cada fila, recorreremos x elementos, es decir $O(\text{get}(i)) = \sqrt{n}$.
- En el caso general, en que las filas \neq columnas, $O(\text{get}(i)) = n/\text{columnas}$, es decir se convierte en sublineal.

Por lo tanto el orden de este algoritmo varia de lineal a sublineal, dependiendo de la estructura de la matriz.

Las otra técnica de representación está basada en que también los encabezados son listas, en vez de arreglos lexicográficos. Así, otra representación de la matriz anterior es así:



Aquí los nodos de los encabezados tendrían la siguiente estructura:



Esta representación al seguir manteniendo los nodos con doble enlace en ambas dimensiones, permite tener la misma flexibilidad en el recorrido de elementos que la representación con encabezados de arreglos. La diferencia básica es que el recorrido en los encabezados. Sin embargo, cuando se tienen múltiples dimensiones, mantener en los nodos el doble enlace hacia cualquier dirección, complica de gran manera las operaciones de inserción y eliminación, además de representar un consumo alto de memoria por uso de dos apuntadores por nodo, por dimensión. Para estos casos se podría utilizar una representación como la siguiente:

arreglos-ortogonales-listas2

En este caso, los nodos de información ya no necesitan almacenar ni apuntadores, ni índices de la posición que representa, ya que dicha posición, es determinada en el recorrido de los encabezados.

El almacenamiento encadenado puede ser útil para acomodar el crecimiento de los arreglos en direcciones arbitrarias, a costa de una utilización de espacio reducido

2.3.2. Comparación de técnicas

El rendimiento del acceso a los arreglos, está dado por la representación interna

Criterios de comparación

- simplicidad de acceso a los elementos
- facilidad de recorridos en diferentes rutas
- eficiencia de uso del almacenamiento
- facilidad de crecimiento

Almacenamiento lexicográfico vs esparcido

Ventajas

- cálculo fácil ($O(n) = c$)
- facilidad de recorrido
- crecimiento fácil en la primera dimensión

Desventajas

- crecimiento muy costos en las otras dimensiones
- Uso de la memoria sólo es óptimo si está lleno (o casi lleno). $Radio = (\text{tam(objeto)} / \text{tam(objeto} + \text{apuntadores)})$ indica el máximo porcentaje de llenado a partir del cual es más eficiente un arreglo lexicográfico.

Capítulo 3

Árboles

Conceptos generales y representaciones de árboles

3.1. Conceptos generales de árboles

3.1.1. La teoría de grafos y los árboles

El concepto de **árbol** como estructura de datos tiene su origen en el concepto de grafo. Repasando, a continuación algunos conceptos de grafos:

- **Grafo:** es una colección de de vértices V , y una colección de lados L , donde por cada lado en L , hay una línea que une un par de vértices en V
- **Vértices Adyacentes:** son los que están unidos por un lado
- **Ruta:** de longitud n , es una secuencia de lados $l_1 l_2 \dots l_n$, donde lado l_k comparte un vértice en común con el lado precedente l_{k-1} , y un vértice en común con el lado subsiguiente l_{k+1} , para $1 < k < n$. Otra definición es: Ruta de longitud n es un conjunto de vértices $v_0 v_1 \dots v_n$, tal que v_{k-1} es adyacente a v_k , para $1 < k < n$.
- **Ruta Simple:** cuando todos los vértices en él son distintos, excepto posiblemente por el primero y último
- **Ciclo:** es una ruta de longitud tres o más, que conecta un vértice v con él mismo.
- **Ciclo Simple:** es un ciclo, cuya ruta es simple.
- **Grafo Conectado:** es aquel donde existe, al menos una ruta de un vértice a otro.
- **Árbol libre:** es un grafo finito y conectado, sin ciclos simples.
- **Árbol orientado:** es un grafo, donde cada lado de L , es un par de vértices de la forma (v, v') donde se dice que v es el origen y v' es el destino y la dirección de l va de v a v' , y se toma un vértice r como la raíz donde inicia el grafo.
- **Árbol ordenado:** es un conjunto finito de uno o más vértices tales que hay un vértice designado r , llamado la raíz, y tal que los vértices restantes, son divididos en $n > 0$ subconjuntos mutuamente exclusivos, cada uno de los cuales son a la vez árboles ordenados.
- **Árbol binario:** Es un conjunto finito de vértices que están vacíos o consiste de un vértice llamado raíz y dos subárboles binarios, los cuales son disjuntos uno del otro, y son llamados subárboles izquierdo y derecho
- **Árbol binario lleno:** Es un árbol binario en el cual cada nodo o es hoja o tiene exactamente dos descendientes no vacíos.

- **Árbol binario completo:** es un árbol binario con hojas en los dos últimos niveles y en las hojas del último nivel están a la izquierda

Los árboles como estructuras de datos son sólo una de las aplicaciones que existen para este concepto. Entre ellos están los **árboles de costo mínimo**, los **árboles de expresiones**, **mapas**, etc.

3.1.2. Representaciones de árboles en memoria

Los árboles binarios son construidos típicamente con una estructura de nodos y apuntadores en la cual se almacenan datos, cada uno de estos nodos tienen una referencia o apuntador a un nodo izquierdo y a un nodo derecho denominados hijos. En ocasiones, también contiene un apuntador a un único nodo. Si un nodo tiene menos de dos hijos, algunos de los punteros de los hijos pueden ser definidos como nulos para indicar que no dispone de dicho nodo. En la figura adjunta se puede observar la estructura de dicha implementación.

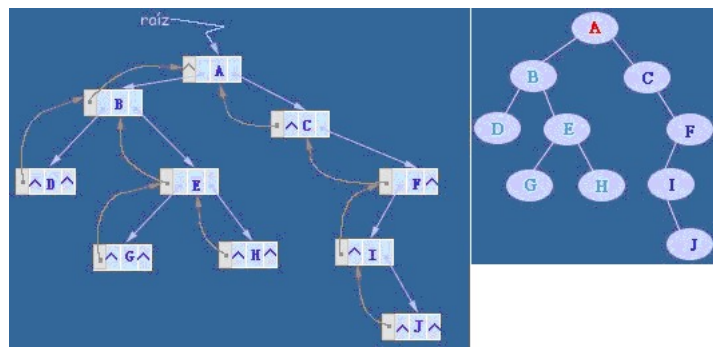


Figura 3.1: Árbol en memoria

Los árboles binarios también pueden ser almacenados como una estructura de datos implícita en arrays, y si el árbol es un árbol binario completo, este método no desaprovecha el espacio en memoria. Tomaremos como notación la siguiente: si un nodo tiene un índice i , sus hijos se encuentran en índices $2i + 1$ y $2i + 2$, mientras que sus padres (si los tiene) se encuentra en el índice $(i - 1)/2$ (partiendo de que la raíz tenga índice cero). Este método tiene como ventajas el tener almacenados los datos de forma más compacta y por tener una forma mas rápida y eficiente de localizar los datos en particular durante un preorden transversal. Sin embargo, *si el árbol no está completo, desperdicia mucho espacio en memoria.*

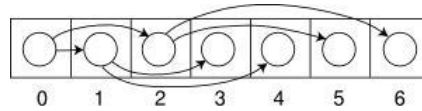


Figura 3.2: Árbol binario en array.

3.1.3. Codificación de árboles n-arios como árboles binarios

Hay un mapeo uno a uno entre los árboles generales y árboles binarios, el cual en particular es usado en Lisp para representar árboles generales como árboles binarios. Cada nodo N ordenado en el árbol corresponde a un nodo N' en el árbol binario; el hijo de la izquierda de N' es el nodo correspondiente al primer hijo de N , y el hijo derecho de N' es el nodo correspondiente al siguiente hermano de N , es decir, el próximo nodo en orden entre los hijos de los padres de N .

Esta representación de árbol binario de un árbol general, a veces se hace referencia como un árbol binario **primer hijo/siguiente hermano**, o un árbol doblemente encadenado.

Una manera de pensar acerca de esto es que los hijos de cada nodo estén en una lista enlazada, encadenados junto con campo derecho, y el nodo sólo tiene un **apuntador** al comienzo o la cabeza de esta lista, a través de su campo izquierdo.

Por ejemplo, en el árbol de la izquierda, la A tiene 6 hijos (B, C, D, E, F, G). Puede ser convertido en el árbol binario de la derecha.

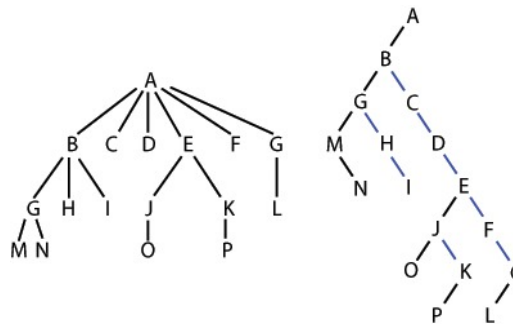


Figura 3.3: Árbol general a binario

Esta representación tiene la ventaja que puede representar cualquier árbol multihijos, utilizando sólo dos apuntadores por nodo. Es útil cuando el árbol a representar tiene una gran cantidad de hijos por nodo, o la cantidad de hijos es variable y/o indeterminada.

La desventaja es que los recorridos se complica.

3.2. Operaciones sobre árboles

Como estructura de datos, sobre los árboles se pueden realizar diferentes operaciones como agregar, buscar y eliminar elementos del árbol. Sin embargo cualquier problema que se quiera resolver con el uso de árboles, en general, se convertirá en realizar un **recorrido** sobre el árbol.

3.2.1. Recorridos

Definición 3.2.1 Un **recorrido** es una operación sobre un árbol en el que se recorren los elementos en algún orden definido, para realizar sobre los elementos algún proceso, llamado VISITA.

Según esta definición, hay $n!$ formas de recorrer un árbol con n elementos, lo cual nos deja un número igual de recorridos posibles. Sin embargo, las operaciones de recorrido definidas se refieren normalmente a **recorridos completos**.

Definición 3.2.2 Un **recorrido completo** es un tipo de recorrido en el cual se VISITAN todos elementos del árbol, en algún orden definido.

Existen varios recorridos definidos, pero veremos cuatro principales. Así, sea un árbol T con subárbol izquierdo T_i y subárbol derecho T_d , entonces se tienen los siguientes recorridos

Preorden: es el recorrido donde se VISITA T y luego se realiza el recorrido preorden de T_i y luego el recorrido preorden de T_d

En orden: es el recorrido donde se realiza el recorrido en orden de T_i , y luego se VISITA T y luego el recorrido en orden de T_d

Postorden: es el recorrido se realiza el recorrido post orden de T_i , luego el recorrido post orden de T_d y luego se VISITA T

Por niveles: es el recorrido donde se VISITAN los nodos en el orden del nivel del árbol, así se visita T , luego se realiza el recorrido de las raíces de T_i y T_d , luego el recorrido por nivel de los hijos inmediatos de T_i y T_d , y así sucesivamente hasta recorrer todos los niveles.

Cada uno de estos recorridos produce una secuencia de elementos que toma el nombre del recorrido correspondientes. Así hablamos de la *secuencia preorden*, *secuencia en orden*, etc. Por ejemplo: en el siguiente árbol

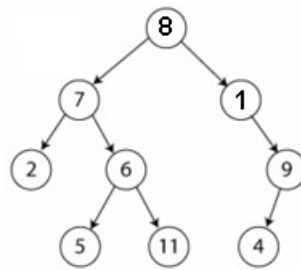


Figura 3.4: Secuencias de un árbol.

Tiene las siguientes secuencias:

- Secuencia pre orden: 8, 7, 2, 6, 5, 11, 1, 9, 4
- Secuencia en orden: 2, 7, 5, 6, 11, 8, 1, 4, 9
- Secuencia post orden: 2, 5, 11, 6, 7, 4, 9, 1, 8
- Secuencia por niveles: 8, 7, 1, 2, 6, 9, 5, 11, 4

En cada secuencia de llaves, que determina una relación de sucesor y predecesor en cada recorrido, así hablamos de relaciones sucesor-predecesor, según las secuencias. Por ejemplo: 5 es predecesor en preorden del 11, predecesor en orden del 6, sucesor en postorden del 2

Una posible implementación es:

```

class NodoBin {
    NodoBin *izq ;
    NodoBin *der ;
    TInfo info ;
    .....
}

class ArbolBinario {
public:
    void preOrden () {
        preOrden(raiz) ;
    }

    void enOrden () {
        enOrden(raiz) ;
    }
}
  
```

```
void postOrden () {
    postOrden(raiz) ;
}
private:
    NodoBin *raiz ;
    ....
void visita (NodoBin *n) {
    // cualquier operación que se desee hacer sobre cada nodo
}

void preOrden (NodoBin *n) {
    if ( n != null) {
        visita (n) ;
        preOrden(n->izq) ;
        preOrden(n->der) ;
    }
}

void enOrden (NodoBin *n) {
    if ( n != null) {
        enOrden(n->izq) ;
        visita (n) ;
        enOrden(n->der) ;
    }
}

void postOrden (NodoBin n) {
    if ( n != null) {
        postOrden(n->izq) ;
        postOrden(n->der) ;
        visita (n) ;
    }
}
}
```

3.3. Árboles binarios de búsqueda

Un árbol de búsqueda, también de comparación o decisión, es un árbol en el que cada llave L tiene subárbol izquierdo y derecho, tal que las llaves del subárbol izquierdo de L son menores a L , y las del subárbol derecho son mayores a L , y los subárboles derecho e izquierdo de L son también árboles de búsqueda.

Nótese que hablamos de llaves en vez de nodos

3.3.1. Árbol binario de búsqueda

Es un árbol de búsqueda en el que cada nodo tiene si mucho 2 hijos.

Por ejemplo, en la siguiente gráfica tenemos el árbol binario anterior, convertido en árbol binario de búsqueda:

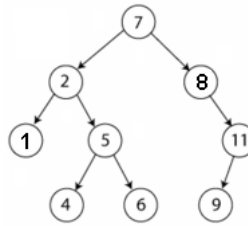


Figura 3.5: Árbol binario de búsqueda de Figura 3.4

Otro ejemplo:

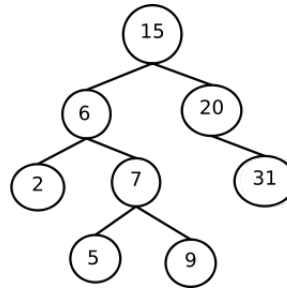


Figura 3.6: Otro árbol binario de búsqueda.

3.3.2. Búsqueda

La búsqueda consiste acceder a la raíz del árbol, si el elemento a localizar coincide con éste la búsqueda ha concluido con éxito, si el elemento es menor se busca en el subárbol izquierdo y si es mayor en el derecho. Si se alcanza un nodo hoja y el elemento no ha sido encontrado se supone que no existe en el árbol. Cabe destacar que la búsqueda en este tipo de árboles es muy eficiente, representa una función logarítmica de base 2.

```

class NodoBin {
    TLLave llv ;
    TValor value
    NodoBin *izq, *der ;

    NodoBin (TLLave llv, TValor value) {

```

```

        this->llv = llv ;
        this->value = value ;
        izq = null ;
        der = null ;
    }
}

class ArbolBinBusqueda :: ArbolBinario {
    NodoBin *raiz ;
    .....
    TValor get(TLlave llv) {
        NodoBin * temp = raiz ;

        while ( temp != nullptr && llv != temp->llv) {
            if (llv < temp->llv) {
                temp = temp->izq ;
            }else{
                temp = temp->der ;
            }
        }

        if ( temp == null)
            return null ; // no está
        else
            return temp->valor ;
    }
}

```

Uso correcto de clases: En general, cuando trabajamos programación orientada a objetos, se comete el error común de saltarnos un método de una clase. Usualmente tenemos un método como el siguiente:

```

class x {
    ....
    int x1(T obj) {
        if ( obj.a() )
            x = obj.b() ;
        ....
        obj.c() ;
        ..
        obj.d() ;
        ..
    }
    ...
    int y() {

```



```

        ...
        x1(o) ;
        ..
    }
}

```

Nos damos cuenta que en el método `x1`, se se recibe un parámetro del clase `T`, y toda la lógica del método es alrededor de dicho parámetro (`obj`), entonces ésto debería convertirse en:

```

class T {
    int x1() {
        if (a() ) ..
            x = b() ;
        c() ;
        ..
        d() ;
        ..
    }
    ....
}

class x {
    ...
    int y() {
        ...
        o.x1() ;
        ..
    }
}

```

3.3.3. Orden de la búsqueda

$$\begin{aligned}
 T(\text{get}(n)) &= T(\text{asignacion}) + T(\text{while}) + T(\text{if}); \\
 &= t + (T(\text{condicion}) + K * T(\text{cuerpo})) + (T(\text{condicion}) + \max(T(\text{then}), T(\text{else}))) \\
 &= t + (t + K(T(\text{if}))) + (t + \max(t, t)) \\
 &= t + (t + K(T(\text{condicion}) + \max(T(\text{then}), T(\text{else})))) + (t + t) \\
 &= t + (t + K(t + \max(t, t))) + (t + t) \\
 &= t + (t + K(t + t)) + (t + t)
 \end{aligned}$$

K es el número de veces que se ejecuta el cuerpo del while, en términos de n , donde n es el número de llaves.

K = altura del árbol y la relación entre la altura y N está dada por

$$2^k - 1 = n \longrightarrow k = \lg(n + 1)$$

por lo tanto

$$\begin{aligned} T(\text{get}(n)) &= t + (t + \lg(n + 1))(t + t) + (t + t) \\ &= 4t + \lg(n + 1)(2t + t) \longrightarrow O(\text{get}(n)) = \lg(n) \end{aligned}$$

3.3.4. Inserción

La inserción es similar a la búsqueda. Se procede de la siguiente forma, si el nodo pasado como parámetro está vacío se crea un nuevo nodo para él cuyo contenido correspondiente sería el elemento a insertar. Si no lo está, se comprueba si el elemento dado es menor que el de la raíz del árbol con lo que se inserta en el subárbol izquierdo, o mayor, insertándose en el subárbol derecho. Se observa que de este modo las inserciones se realizan en las hojas, es la forma más simple de llevar a cabo esta tarea, aunque no la única.

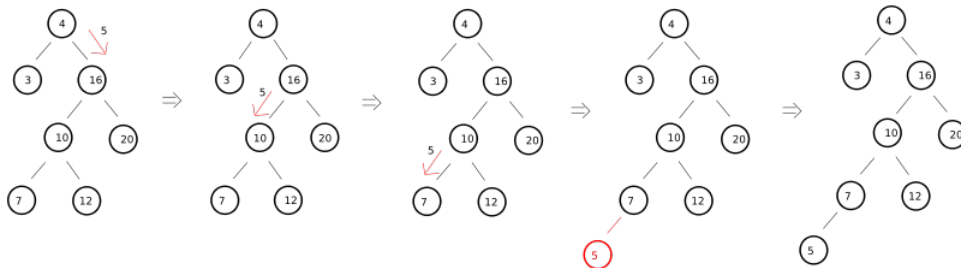


Figura 3.7: Insercion en árbol

3.3.5. Eliminación

La operación de borrado no es tan sencilla como las de búsqueda e inserción. Existen varios casos a tener en consideración:

- Borrar un nodo sin hijos ó nodo hoja: simplemente se borra y se establece a nulo el apuntador de su padre.
- Borrar un nodo con un subárbol hijo: se borra el nodo y se asigna su subárbol hijo como subárbol de su padre.
- Borrar un nodo con dos subárboles hijo: la solución está en reemplazar el valor del nodo por el de su predecesor o por el de su sucesor en orden y posteriormente borrar este nodo. Su predecesor en orden será el nodo

más a la derecha de su subárbol izquierdo (mayor nodo del subarbol izquierdo), y su sucesor el nodo más a la izquierda de su subárbol derecho (menor nodo del subarbol derecho). En la siguiente figura se muestra cómo existe la posibilidad de realizar cualquiera de ambos reemplazos:

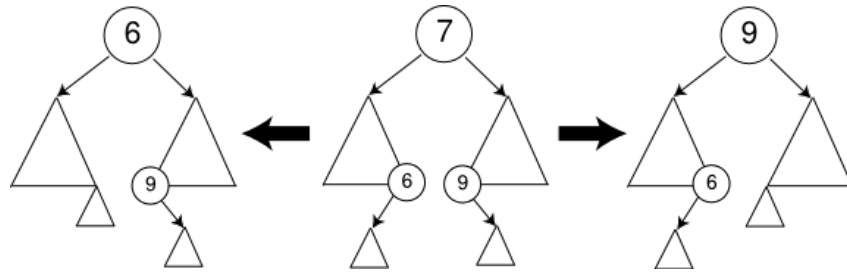


Figura 3.8: Eliminación en árbol

```
class ArbolBinBusqueda:: ArbolBinario {
public:
    ....
    TValor remove(TLlave key) {
        return remove (raiz, key) ;
    }

    TValor remove (NodoBin *n, TLlave key) {
        TValor value = null ;
        NodoBin *padre = null ;

        while (n != nullptr && n->llave != key) {
            padre = n ;
            if (key < n->llave)
                n=n->izq ;
            else
                n = n->der ;
        }

        if (n!=nullptr) {
            value = n->value ;
            if (n->izq == null && n->der == nullptr) { //es nodo hoja
                if (padre->izq == n)
                    padre->izq = null ;
                else
                    padre->der = null ;
            }else{

```

```
        if (n->izq == null) { // sólo tiene hijo derecho
            if (padre->izq == n)
                padre->izq = n->der ;
            else
                padre->der = n->der ;
        }else{
            if (n->der == nullptr) { // sólo tiene hijo izquierdo
                if (padre->izq == n)
                    padre->izq = n->izq ;
                else
                    padre->der = n->izq ;
            }else{ // tiene dos subárboles
                // predecesor en orden
                NodoBin *temp = n->izq ;
                while (temp->der != nullptr)
                    temp = temp->der ;
                n->llave = temp->llave ;
                n->valor = temp->valor ;
                remove (n->izq, n->llave) ;
            }//if interno
        }//if
        return value ;
    }//funcion remove
}
```

En el caso ideal una búsqueda de una llave tiene un orden $O(n) = \lg(n)$, sin embargo puede suceder el fenómeno del desbalance

3.3.6. Árbol AVL

El árbol AVL toma su nombre de las iniciales de los apellidos de sus inventores, Adelson-Velskii y Landis. Lo dieron a conocer en la publicación de un artículo en 1962: «An algorithm for the organization of information» («Un algoritmo para la organización de la información»).

Los árboles AVL están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha. Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad $O(\lg n)$. El factor de equilibrio puede ser almacenado directamente en cada nodo o ser computado a partir de las alturas de los subárboles.

Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se ha de realizar de una forma especial. Si al realizar una operación

de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de rotaciones de los nodos.

Los árboles AVL más profundos son los árboles de Fibonacci.

Definición.

Definición 3.3.1 *Altura de un árbol.* Sea T un árbol binario de búsqueda y sean T_i y T_d sus subárboles, su altura $H(T)$, es:

- 0 si el árbol T está vacío
- $1 + \max(H(T_i), H(T_d))$ si no está vacío

Definición 3.3.2 *Árbol AVL.* Sea T un árbol binario de búsqueda (ABB) con T_i y T_d siendo sus subárboles izquierdo y derecho respectivamente, tenemos que:

- Si T es vacío, es un árbol AVL
- Si T es un ABB no vacío, es AVL si (si y sólo si):
 - * T_i y T_d son AVL y
 - * $-1 \leq H(T_i) - H(T_d) \leq 1$

Ej: árbol binario de búsqueda desbalanceado

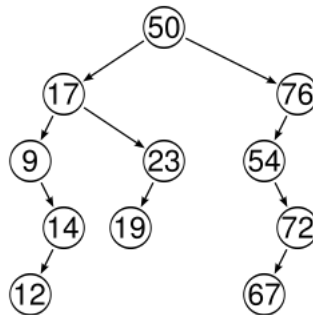


Figura 3.9: Árbol binario de búsqueda desbalanceado

Mismo árbol balanceado

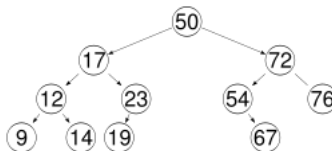


Figura 3.10: Árbol binario de búsqueda balanceado

Por esta definición tenemos que el árbol de la figura de arriba no es AVL, mientras que el de abajo sí lo es. Véase también que se trata de un árbol ordenado, en el cual para cada nodo todos los nodos de su subárbol izquierdo tienen un valor de clave menor y todos los nodos de su subárbol derecho tienen un valor de clave mayor que el suyo, cumpliendo así la propiedad de los ABB.

Factor de equilibrio

Cada nodo, además de la información que se pretende almacenar, debe tener los dos apuntadores a los árboles derecho e izquierdo, igual que los árboles binarios de búsqueda (ABB), y además el dato que controla el factor de equilibrio.

El factor de equilibrio es la diferencia entre las alturas del árbol derecho y el izquierdo:

$FE = \text{altura subárbol derecho} - \text{altura subárbol izquierdo};$

Por definición, para un árbol AVL, este valor debe ser -1, 0 ó 1.

Operaciones

Las operaciones básicas de un árbol AVL implican generalmente el realizar los mismos algoritmos que serían realizados en un árbol binario de búsqueda desequilibrado, pero precedido o seguido por una o más de las llamadas «rotaciones AVL».

Búsqueda Las búsquedas se realizan de la misma manera que en los ABB, pero al estar el árbol equilibrado la complejidad de la búsqueda nunca excederá de $O(\log n)$.

```
class NodoAVL::NodoBin {
    int balance ;
    NodoAVL(TLlave llv, TValor value):NodoBin (llv,value) {
        balance = 0 ;
    }

    NodoAVL *rotacionDer() {
        NodoAVL *temp = izq ;
        izq = izq.der ;
        temp.der = this ;
        return temp ;
    }
}

class ArbolAVL::ArbolBinarioBusqueda {
private:
```

```
NodoAVL<K,V> raiz ;
public:
.....
TValor get (TLlave llv) {
    NodoBin *temp = raiz ;
    while (temp != nullptr && llv != temp->llv) {
        if (llv < temp->llv) {
            temp = temp.izq ;
        }else{
            temp = temp.der ;
        }
    }
    if (temp == nullptr)
        return nullptr ; // no está
    else
        return temp.valor ;
}
.....
}
```

Inserción La inserción en un árbol de AVL puede ser realizada insertando el valor dado en el árbol como si fuera un árbol de búsqueda binario desequilibrado y después retrocediendo hacia la raíz, rotando sobre cualquier nodo que pueda haberse desequilibrado durante la inserción.

Dado que como mucho un nodo es rotado 1.5 veces $\log n$ en la vuelta hacia la raíz, y cada rotación AVL tarda el mismo tiempo, el proceso de inserción tarda un tiempo total de $O(\log n)$.

```
class ArbolAVL::ArbolBinarioBusqueda {
private:
    NodoAVL *raiz ;
public:
.....
TValor put (TLlave llv, TValor value) {
    bool crecio= false ;
    if (raiz == nullptr) // árbol vacío
        raiz = new NodoAVL(llv, value) ;
    else
        raiz = put(llv, value, raiz, crecio) ;
    return value ;
}
NodoAVL *put (TLlave llv, TValor value, NodoAVL *n, bool &crecio) {
    if (llv == n->llv) // ya está ==> error
        throw "Duplicado" ;
```

```
else if (llv < n->llv) { // debe insertarse a la izquierda
    if (n->izq != null) { // existe hijo izquierdo
        put (llv, value, n->izq, crecio) ;
        if ( crecio ) {
            n->balance -- ;
            if (n->balance < -1) { //necesario balancear
                if ( n->izq->balance > 0 ) //rotación doble
                    n->izq = n->izq->rotacionIzq() ;
                crecio = false ;
                return n->rotacionIzq() ;
            }
        }
    }else{ // no existe hijo izquierdo
        n->izq = new NodoAVL(llv, value) ;
        n.balance -- ;
        crecio = (n.balance!=0) ;
    }
}
else{ // debe insertarse en la derecha
    if (n->der != nullptr) { // existe hijo derecho
        put (llv, value, n->der, crecio) ;
        if ( crecio ) {
            n->balance ++ ;
            if (n->balance > 1) { //necesario balancear
                if ( n->der->balance < 0 ) //rotación doble
                    n->der = n->der->rotacionDer() ;
                crecio = false ;
                return n->rotacionIzq() ;
            }
        }
    }
}
else { //no existe hijo derecho
    n->der = new NodoAVL(llv, value) ;
    n.balance ++ ;
    crecio = (n->balance!=0) ;
}
}
```

Extracción El problema de la extracción puede resolverse en $O(\log n)$ pasos. Una extracción trae consigo una disminución de la altura de la rama donde se extrajo y tendrá como efecto un cambio en el factor de equilibrio del nodo padre de la rama en cuestión, pudiendo necesitarse una rotación.

Esta disminución de la altura y la corrección de los factores de equilibrio con sus posibles rotaciones asociadas pueden propagarse hasta la raíz.

3.3.7. Árbol HB[K]

Sea T un árbol binario de búsqueda (ABB) con T_i y T_d siendo sus subárboles izquierdo y derecho respectivamente, tenemos que:

Si T es vacío, es un árbol HB[k]

Si T es un ABB no vacío, es HB[k] si y sólo si:

- T_i y T_d son HB[k] y
- $-k \leq H(T_i) - H(T_d) \leq k$

HB[1] = AVL

3.4. Árboles B

Hasta ahora se ha descrito sólo árboles binarios. Sin embargo, en las aplicaciones donde se necesitan árboles de búsqueda a gran escala son necesarios los árboles multicaminos, o multihijos. Éstos tienen la propiedad que pueden tener más de un hijo por nodo.

Un árbol B de orden k ($k \geq 2$), tiene las siguientes propiedades:

1. Es un árbol de búsqueda
2. Todos los nodos tienen como máximo de k llaves
3. Todos los nodos tienen como mínimo el piso $(k/2)$ llaves, excepto la raíz que puede tener 1
4. Un nodo con x llaves tiene $x + 1$ hijos o es hoja
5. Todas las hojas aparecen en el mismo nivel

3.5. Ejercicios

3.5.1. Árboles HB[K]

Dadas las siguientes clases:

```
template <class T>
class NodoBin {
public:
    T valor ;
    NodoBin *izq ;
    NodoBin *der ;
```

```

    NodoBin();
    virtual ~NodoBin();
};

template <class T, class K>
class NodoBinBusq: NodoBin<T> {
public:
    K llave ;

    NodoBinBusq();
    virtual ~NodoBinBusq();
};

template <class T, class K>
class ArbolBinBusq {
public:
    ArbolBinBusq();
    virtual ~ArbolBinBusq();

    /* retorna true si el árbol es un HB[k], caso contrario retorna false

    bool esHB(int k) ;

    /* retorna la altura del árbol

    int altura() ;

private:

    NodoBinBusq *raiz;

    .....

};

```

Implementar los métodos `ArbolBinBusq::altura()` y `ArbolBinBusq::esHB(int k)`.
Este último debe ser $O(n)=n$.

No utilizar ninguna estructura de datos adicional.

Entregar compilables: .h y .cpp

3.5.2. Árbol en arreglo

Dado un árbol binario de búsqueda, representado en un arreglo unidimensional, haga un algoritmo que realice un recorrido tal que imprima las llaves en orden descendente. No utilizar ninguna estructura de datos adicional. El orden debe ser $O(n)=n$

3.5.3. Árbol de expresiones

Dada una clase `ArbolExp`, que representa un árbol de expresiones, escriba un método `valor()` que dé como resultado el valor de la expresión representada. Puede suponer que sólo existen los operadores de suma, resta, multiplicación y división. También se supone que todos los operandos son constantes, es decir no hay variables. No debe utilizar ninguna estructura de datos adicional. Suponga que sólo existen las siguientes variables de cada clase:

```
class ArbolExp {  
  
public:  
  
.....  
  
int valor ( ) ;  
  
....  
  
private:  
  
NodoExp *raíz ;  
  
....  
  
}  
  
class NodoExp {  
  
.....  
  
void *valor ; //constante u operador  
  
NodoExp *izq ;  
  
NodoExp *der ;  

```

```
.....
}
```

3.5.4. Ordenamiento de árbol B[k]

Escriba un método de una clase ArbolB, que representa un árbol B[k], que imprima las llaves en orden de mayor a menor. Dicho algoritmo debe ser de orden $O(n)=n$ y no debe utilizar ninguna estructura de datos adicional. Suponga que sólo existen las siguientes variables de cada clase:

```
class ArbolB {

public:

.....

void recorridoInverso() ;

....

private:

NodoB *raíz ;

int k ;

....

}

class NodoB {

.....

int llaves[] ;

int info[];

NodoB* hijo[]

int k ;
```

```
int numElementos ;  
  
.....  
}
```

Capítulo 4

Tablas de dispersión

En esta unidad revisaremos otra estructura para uso de contenedores, que busca obtener el rendimiento de los arreglos lexicográficos y la flexibilidad en el uso de memoria de los árboles. El contenido básico es:

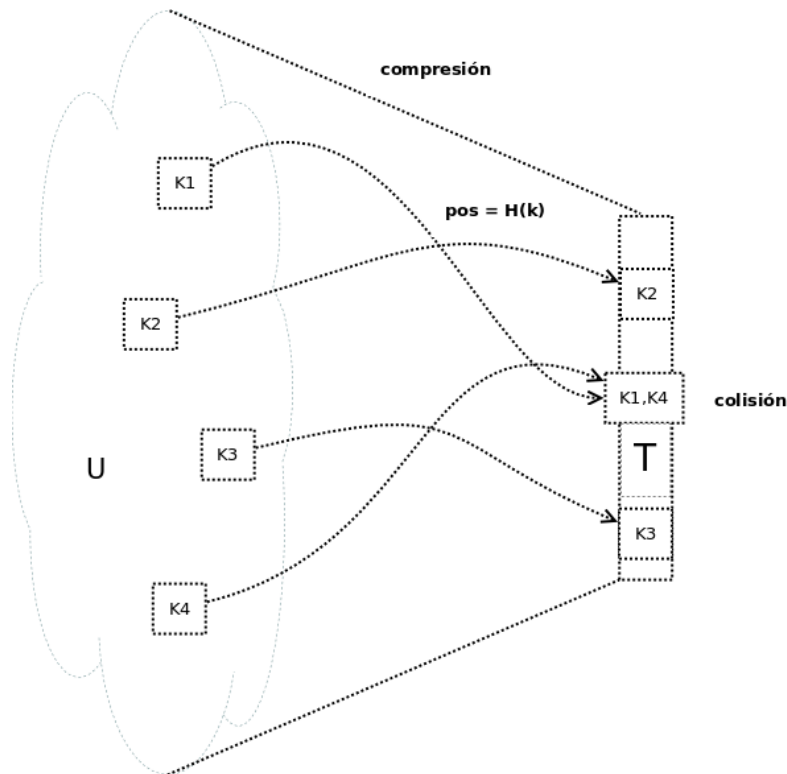
- Componentes generales de una tabla de dispersión
- Compresión
- Funciones de dispersión
- Resolución de colisiones

4.1. Componentes generales de las tablas de dispersión

Hemos visto, como estructura de contenedor, los arreglos y los árboles, cada uno con sus propias ventajas y desventajas. Por un lado los arreglos lexicográficos con una gran eficiencia de acceso, y por otro, los árboles con el dinamismo y optimización en el uso de la memoria. ¿Y si pudiéramos tener una estructura que combine las ventajas de los arreglos lexicográficos y árboles?

Esa es la idea principal de los proponentes de las tablas de dispersión, por un lado se tiene una tabla representada en una arreglo lexicográfico, que puede tener elementos dinámicos para crecimiento a futuro.

Los componentes general de una tabla de dispersión se resumen en la siguiente gráfica.



Podemos distinguir los siguientes elementos:

- Hay un universo U de elementos posibles que pueden llegar a ser utilizados en la memoria
- Tenemos la tabla T, que es un arreglo lexicográfico en la memoria de la computadora, con los elementos de U, realmente utilizados

- Los elementos de U están identificados por llaves k_i
- Las llaves de cada elemento son mapeadas a posiciones individuales por medio de una función de dispersión ($\text{pos} = h(k)$)
- Dado que las llaves k_i pueden ser alfanuméricas, puede existir primero una conversión de alfanumérico a numérico. Este proceso de conversión y mapeo puede ser considerado dentro de la misma función de dispersión $h(k)$
- Dado que $U \gg T$ (hay muchos más elementos en U que en T) invariablemente se dará el caso en que dos llaves mapearán a la misma posición. Ésto es una **colisión**.

4.2. Políticas de resolución de colisiones

Además de la función de dispersión $H(K)$, el otro componente de las tablas de dispersión es la política de resolución de colisiones.

Esta política se refiere a la estrategia que se usará para resolver el problema de las colisiones, que inevitablemente se dará, dado el hecho que el número de elementos posible en el universo, es mucho mayor que el número de elementos que caben en la tabla en memoria.

La elección de la política de resolución es tan importante que normalmente esta decisión da nombre a la estructura, como veremos en las siguientes secciones.

4.3. Dispersión por encadenamiento

La forma más simple, es que cada posición de la tabla tiene un apuntador a una lista que contiene todos los elementos que colisionaron en esa posición, así:

Esta estrategia tiene la ventaja que es relativamente simple de implementar, sin embargo su rendimiento, $O(n)$, varía entre n/m y n . Donde n es el número de elementos en la tabla y m .

El rendimiento está dado así porque, en el mejor de los casos, en que la función de dispersión cumple con repartir uniformemente los elementos en la tabla, las listas en cada posición, en el largo plazo, tendrán la misma cantidad de elementos (n/m). El acceso a una posición de la tabla está dado por la ejecución de $H(k)$, lo que se toma como $O(n)=1$, y la búsqueda en una lista está dado por el tamaño de la lista, por lo tanto $O(\text{Tabla.get}(k)) = n/m$.

Sólo en el caso extremo, en que todos los elementos colisionen en la misma posición, el rendimiento se degenerará a $O(n)=n$, dado que todos los elementos de la tabla estarán en la misma lista.

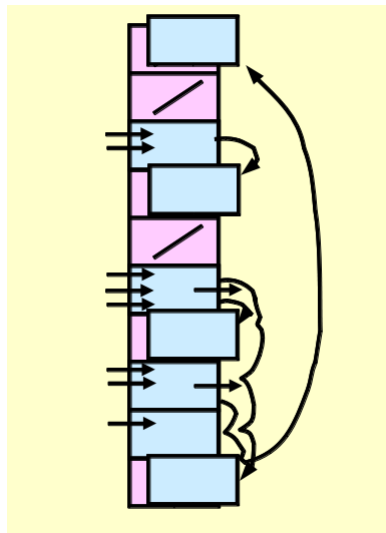
En busca de optimizar dicho rendimiento, se ha pensado en utilizar árboles AVL en vez de listas. Esto implica que $O(n) = \lg(n/m)$. Otra vez, debido a que los árboles tenderán a tener la misma cantidad de elementos: n/m .

De forma similar, si usáramos árboles B[k], tendríamos un rendimiento $O(n) = \log_k(n/m)$.

En general, cualquier estructura dinámica que se use para resolver la colisión, el efecto es que el rendimiento se multiplicará, dado que es más eficiente buscar en una lista o árbol de tamaño n/m que en una lista o árbol de n , si no usáramos una tabla de dispersión.

4.4. Direcccionamiento abierto

Al contrario del encadenamiento, no utiliza memoria adicional a la tabla en sí. Las colisiones se resuelven dentro de la misma tabla, así:



Básicamente, cuando una llave colisiona en una posición, se sigue la siguiente estrategia:

- Se realiza un ciclo de intentos de buscar una posición al elemento
 - En cada intento se examina la posición dada por $\text{posición actual} + S(k,i) \% m$, donde k es la llave a insertar, i es el número de intento y $S(k,i)$ es la función que determina la **secuencia de prueba**.
 - Si la posición examinada está libre, entonces se asigna dicha posición y se finaliza

Según la naturaleza de $S(k,i)$, puede ser que los intentos generen un ciclo infinito de intentos, por lo que debe establecer un límite al número de intentos.

Dicho límite establecerá el rendimiento máximo de la estructura. Diferentes autores han propuesto diferentes límites, que van desde m/e hasta $2m$, donde e es la constante de Euler. Esto dependerá del diseñador de la estructura.

$S(k,i)$ determinará el comportamiento de la política, que puede ser:

Lineal: $S(k,i)=1$, lo que significa que se examinarán posiciones continuas a la posición de colisión. Ésto provocará que se formen grupos de llaves continuas que irán creciendo conforme se inserten más llaves. Ésto se denomina agrupamiento primario.

Cuadrática: $S(k,i) = i^2$ creará secuencias de pruebas con saltos de la forma 1, 4, 9, 16,... a partir de la posición de colisión. Ésto elimina el agrupamiento primario, pero provoca que todas las llaves que colisionan en una misma posición, sigan la misma secuencia de prueba, lo que se denomina agrupamiento secundario.

Doble dispersión: $S(k,i) = (k \% c + 1) * i$. Se utiliza una segunda función de dispersión (por ejemplo $k \% c + 1$) lo cual agrega una variación de una llave a otra, por lo que, aunque dos llaves colisionen en la misma posición, seguirán secuencias de prueba distintas.

$M = 11$
 $H(k) = k \% M$
 $S(k,i) = 1$

12 25 37 50
 23 45 78

0	
1	12
2	
3	25
4	37
5	
6	50
7	
8	
9	
10	

$h(12) = 12 \% 11 = 1$
 $h(25) = 25 \% 11 = 3$
 $h(37) = 37 \% 11 = 4$
 $h(50) = 50 \% 11 = 6$
 $h(23) = 23 \% 11$
 $h(45) = 45 \% 11 = 1$

23 45
 78

$M = 11$
 $H(k) = k \% M$
 $S(k,i) = 1$

12, 25, 37, 50
23, 45, 78

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

Capítulo 5

Textos

Estudio de las diferentes representaciones y aplicaciones del manejo de textos o strings.

5.1. Generalidades

Los textos son, por mucho, las estructuras de datos más utilizadas en la computación. Toda operación en una computadora implica algún procesamiento de texto. Por ejemplo:

- **Compilar un programa:** significa procesar un texto de entrada, que es el programa fuente, hecho en algún lenguaje de programación legible para los humanos, para producir un texto de salida: un programa binario, que legible para la computadora, que aunque el humano no lo puede leer directamente, sigue siendo un texto
- **Procesador de palabras:** lee del teclado, letras y comandos que generarán un texto. En estos programas usualmente hay muchas operaciones adicionales sobre los textos como la verificación de ortografía, formateo de la impresión, etc.

En general, los textos se utilizan en todas las aplicaciones que exista comunicación entre dispositivos, entre computadoras, entre computadoras y personas.

Claro que cuando hablamos de textos como estructura de datos, ya no estamos hablando de contenedores mapas, como árboles y tablas, ya que no hay elementos de un mismo tipo que se puedan acceder por una llave.

Los textos son estructuras de datos, ya que se trata de conjuntos de datos semánticamente relacionados, con la cual se pueden realizar distintas operaciones y transformaciones. También existen diferentes formas de representación que influirán en el desempeño de una aplicación.

Texto

Secuencia de caracteres de longitud finita, que se usa para representar información semánticamente relacionada.

Nótese que esta definición acepta desde los textos legibles por humanos, programas ejecutables y paquetes de transmisión en la red.

5.1.1. Representaciones

Normalmente estamos acostumbrados a una sólo representación de textos, que es la representación contigua con **codificación de longitud fija**, como la codificación ASCII o la UNICODE. Sin embargo veremos que existen otras codificaciones, no necesariamente contiguas y donde el número de bits utilizado para representar cada carácter puede variar de carácter a carácter.

5.1.2. Operaciones

Cuando se habla de operaciones sobre textos o strings, inmediatamente pensamos en catenación, eliminación de caracteres o inserción de caracteres. Sin embargo éstas no son las únicas operaciones que podemos hacer en un texto, y, sin darnos cuenta, las realizamos en muchas aplicaciones de la vida diaria.

Las operaciones que vamos a estudiar son:

Encriptación: Transformación de un texto legible en uno ilegible (criptograma) que sólo puede ser interpretado por quien tenga una palabra oculta (contraseña)

Búsqueda de patrones: Consiste en buscar dentro de texto si existe o no una palabra determinada (patrón) y, si existe, determinar la posición o posiciones donde se encuentra.

Macroexpansión: Consiste en expandir un abreviatura de un texto, a otro texto basado en parámetros. Por ejemplo: lo que realiza el precompilador de C/C++ con las directivas `#define`.

Compresión: Es la transformación que consiste en representar un texto en menos espacio que en su representación normal.

5.2. Búsqueda de patrones

La búsqueda de patrones es una de las operaciones más comunes a realizar en los textos en cualquier aplicación.

La operación básica es que se tiene un texto de n caracteres $S[n]$ y un texto de m caracteres $p[m]$, donde $m \leq n$, y se desea determinar si $p[m]$ forma parte de $s[n]$, y si es así determinar la posición dentro de $s[n]$ donde se encuentra.

Existen diferentes métodos para realizar esta operación:

- Búsqueda simple o de fuerza bruta
- Booyer Moore
- Knutt Morris Pratt

Las cuales detallaremos a continuación.

5.2.1. Búsqueda simple o de fuerza bruta

Es el método en el cual se compara directamente caracter a caracter cada una de las posiciones de $p[m]$ en $s[n]$. Se puede realizar con el siguiente algoritmo:

```
int buscar (char *p, char *s, int m, int n) {
    while (i <= n-m && !encontrado)
    {
        j = 1 ;
        while (j <= m && i+j-1<=n && s[i+j-1] == p[j])
            j ++ ;
        i ++ ;
        encontrado = (j > m) ;
    }

    return encontrado?i:-1 ;
}
```

Si vemos el proceso como una sucesión de superposiciones de $s[n]$ y $p[m]$, la ejecución del algoritmo, sobre $s[10]$ ="HOLALALASA" y $p[5]$ ="LALAS" puede verse así:

```
H O L A L A L A S A
L A L A S
L A L A S
L A L A S
L A L A S
L A L A S <---- encontrado
```

La letra negrilla indica el caracter de $p[m]$ que se compara con la posición superpuesta de $s[n]$ y el subrayado indica la comparación que falla.

El rendimiento de éste algoritmo es de $n*m$ ya que en el peor de los casos se compararán todas las posiciones de p contra s , por ejemplo: si $s[9]$ ="LALALALAS" y $p[3]$ ="LAS"

Ejercicio: demostrar que el $O(\text{buscar}(n,m)) = n*m$

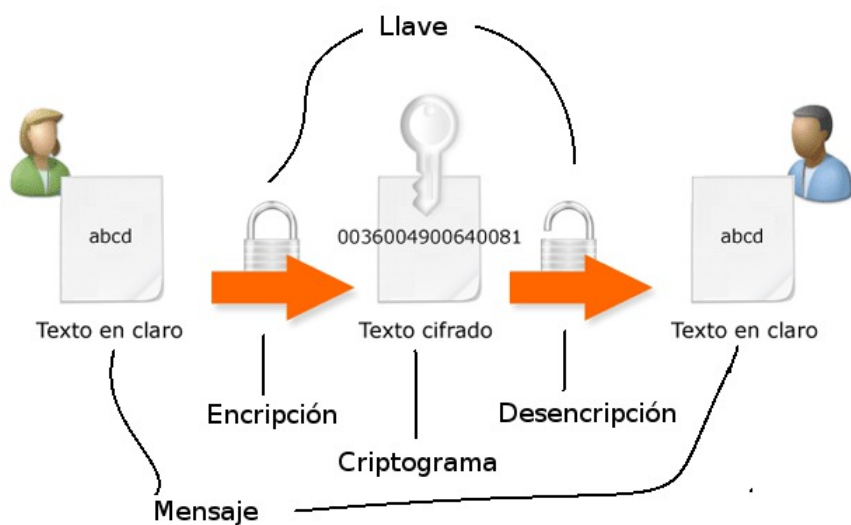
5.2.2. Boyer Moore y Knutt Morris Pratt

[Búsqueda en texto.](#)

5.3. Encriptación

La encriptación es una operación sobre textos que consiste en una transformación de un texto legible, para transformarlo en un texto ilegible, que pueda ser transmitido por un medio público sin que pueda ser descifrado, sino hasta que llegue al destinatario, quien podrá aplicar la transformación inversa y obtener el texto original.

Este proceso se ilustra en la siguiente gráfica:



Como podemos apreciar, hay varios elementos involucrados:

Mensaje: Es el texto legible para cualquier persona

Llave: Es una palabra o contraseña secreta, que sólo sabe el remitente y el destinatario del mensaje

Encriptación: Es la operación de transformar el texto en claro, utilizando la llave, en un texto ilegible

Criptograma: es el texto ilegible, resultado de la operación de encriptación

Desencriptación: Es la operación de transformar el criptograma en el texto original, utilizando la llave predeterminada.

Mas formalmente podemos decir que la encriptación es:

Si se tiene un texto T y una clave K , se puede aplicar una función de encriptación $E(T,K) = C$, donde C se llama criptograma, al cual se le puede aplicar una función de desencriptación $D(C,K) = T$ para obtener el texto original

El uso de la encriptación viene desde los inicios de la historia, desde que las personas han deseado la privacidad, pero la aplicación principal ha sido la guerra.

Los espartanos, son de los primeros que se sabe hacían uso de la encriptación en el Siglo V AC, empleando cintas de lino y un bastón llamado escitalo. La cinta se enrollaba en el bastón y se escribía el texto transversalmente en el bastón, tal como se muestra en la siguiente gráfica:



El mensajero se le entregaba la cinta, la cual al extenderla era ilegible. Para descifrarla, el destinatario debería tener un bastón del mismo diámetro, enrollar la cinta, y leer la sección transversal que tuviera sentido. Este tipo de encriptación es lo que actualmente se llama **transposición**, porque se cambia de sitio los símbolos del mensaje original

En Roman, Julio César desarrolló un mecanismo de codificación que lleva su nombre. Consiste en la **sustitución** de cada letra por la que aparece tres posiciones a su derecha en el alfabeto, así la A se convierte en D, la B en E y así sucesivamente. Ejemplo:

JUAN ->KVBO (corrimiento de 1 caracter)

Otro método más moderno es la **confusión** u **ofuscación**, que consiste en ocultar mensajes dentro de otro mensaje claro. Ejemplo de esto es lo que comunmente se llama "leer entre líneas". Por ejemplo:

Éste es un mensaje

en el cual no hay nada

oculto entre

este mensaje y

otro mensaje que

aunque es peligroso
parece inofensivo
y puede representar
ante los que no sepan leer
la sabiduría que está
entre líneas.

Leer este mensaje de forma normal no releva nada claro, pero si leemos las líneas impares podremos leer el mensaje *"Éste es un mensaje oculto entre otro mensaje que parece inofensivo ante los que no sepan leer entre líneas"*.

La **estenografía**, aunque algunos autores no la consideran encriptación en sí, es un método que aprovecha los vacíos que se dan entre los diferentes formatos de gráficos y multimedios para ocultar información. Por ejemplo en el formato jpg hay espacios que no se utilizan como parte de la gráfica que representa, por lo tanto, dichos espacios pueden ser utilizado para poner información. Lo mismo sucede con el formato **mp3**, y otros formatos de música y gráficos.

Claro que en la computación moderna, éstos métodos distan mucho de ser efectivos, ya que serían fácilmente descifrados. Actualmente se han desarrollado una gran cantidad de algoritmos y métodos de encriptación, los cuales clasificaremos en dos grupos:

1. encriptación simétrica o privada
2. Encriptación asimétrica o pública

5.3.1. Encriptación simétrica o privada

Es el primer tipo de encriptación moderno trabajado a nivel computacional, incluyen máquinas como la máquina codificadora **Enigma**, durante la Segunda Guerra Mundial, en el cual la misma llave utilizada para encriptar, es la que se necesita para realizar la descricpción. Un ejemplo simple es la mezcla con secuencias fijas aleatorias, en la cual se aprovecha la siguiente propiedad de la operación xor

sea x & y dos enteros independientes, entonces se cumple que

- $(x \text{ xor } y) \text{ xor } x = y$
- $(x \text{ xor } y) \text{ xor } y = x$

Si T es el texto a encriptar, T se ve como una secuencia de enteros $T_1 T_2 T_3 \dots T_n$. La llave secreta K , también se ve como una secuencia de enteros $K_1 K_2 K_3 \dots K_m$, donde $m \leq n$. El criptograma C , visto como $C_1 C_2 C_3 \dots C_n$ se forma por sucesivas operaciones xor, de superposiciones de la llave K sobre el texto T , así:

$$\begin{array}{cccccccccc}
 T & T_1 & T_2 & \dots & T_m & T_{m+1} & T_{m+2} & \dots & T_{n-1} & T_n \\
 & & & & \text{xor} & & & & & \\
 K & K_1 & K_2 & \dots & K_m & K_1 & K_2 & \dots & K_i & K_{i+1} \\
 & & & & = & & & & & \\
 C & C_1 & C_2 & \dots & C_m & C_{m+1} & C_{m+2} & \dots & C_{n-1} & C_n
 \end{array}$$

y la operación de descricción sería así:

$$\begin{array}{cccccccccc}
 C & C_1 & C_2 & \dots & C_m & C_{m+1} & C_{m+2} & \dots & C_{n-1} & C_n \\
 & & & & \text{xor} & & & & & \\
 K & K_1 & K_2 & \dots & K_m & K_1 & K_2 & \dots & K_i & K_{i+1} \\
 & & & & = & & & & & \\
 T & T_1 & T_2 & \dots & T_m & T_{m+1} & T_{m+2} & \dots & T_{n-1} & T_n
 \end{array}$$

Una encripción simple y eficiente, pero en una combinatoria de 16^m puede encontrarse la llave K (suponiendo que 16 bits es el tamaño de un entero).

Entre los algoritmos de encripción simétrica actuales se encuentran:

- DES (Data Encryption Standard): llave de 56 bits de longitud, tiempo menor a 3 meses para descryptar el mensaje
- Triple – DES
- Advanced Encryption Standard (AES)
- Blowfish

A pesar de la importancia que han tenido (y seguirán teniendo) la encripción simétrica, en nuestro mundo actual de comunicación masiva y globalizada, tener comunicación privada, a través de estos métodos, presentan dos problemas importantes:

La paradoja del canal seguro: si deseamos encriptar un mensaje, es porque seguramente debemos enviarlo a través de un medio o canal que se considera inseguro, como el correo electrónico. El problema se deriva que el destinatario está geográficamente distante y no es posible enviarla la contraseña por un medio seguro, y si hubiera un medio seguro, no sería necesario encriptar el mensaje

La multiplicidad de intercambios: Si 2 personas necesitan comunicación privada, deben realizar un intercambio de llaves. Si son 3 personas, deben realizar un total de 6 intercambios (2 por cada participante), sin son 4,

12 intercambios y así sucesivamente, que resulta en que si n personas desean comunicarse privadamente, deben haber $n*(n-1)$ intercambios, lo que representa un orden cuadrático.

Estos problemas hicieron que la encriptación simétrica fuera insuficiente para lograr una comunicación privada eficiente en el Internet, y es allí donde cobra importancia la encriptación asimétrica.

5.3.2. Encriptación asimétrica o pública

El RSA es el criptosistema de llave pública más popular basado en el modelo de Diffie-Hellman, el cual ofrece encriptación y firmas digitales (autenticación). Ron Rivest, Adi Shamir y Leonard Adleman desarrollaron el RSA en 1977, de ahí su nombre formado por la primera letra del apellido de sus inventores.

La base de este sistema es la compleja matemática detrás de la factorización de números primos grandes (ver links publicados), que van desde 128 bits hasta 1024, lo cual está fuera del alcance del curso. Sin embargo, lo importante de este método es que tiene las siguientes propiedades:

1. Cada usuario tiene dos llaves, una privada y otra pública. La llave privada es conocida sólo por el propietario y normalmente es almacenada en un archivo protegido por una llave simétrica, conocida sólo por el dueño. La llave pública, por el contrario debe ser conocida por todos los posibles destinatarios, y generalmente se publica por cualquier canal inseguro (correo electrónico, redes sociales, directorio públicos, etc)
2. Sea
 - a) T el texto o mensaje en claro
 - b) X_{prv} la llave privada de la persona X
 - c) X_{pbl} la llave pública de X
 - d) $C=E(T,K)$ la función de encriptación del texto T con la llave K , que da como resultado el criptograma C
 - e) $T=D(C, K)$ la función de desencriptación del criptograma C , con la llave K , que da como resultado el texto T
 - f) Entonces, para RSA se cumple que:
 - 1) si $C=E(T, X_{prv})$ entonces $T=D(C, X_{pbl})$, es decir lo que se encripta con la llave privada, sólo puede desencriptarse con la llave pública
 - 2) si $C=E(T, X_{pbl})$ entonces $T=D(C, X_{prv})$, es decir lo que se encripta con la llave pública, sólo puede desencriptarse con la llave privada

Estas propiedades eliminan los problemas de la paradoja del canal seguro, por el uso de la llave pública, y la multiplicidad de intercambios, ya que para que n personas se comunican, sólo se necesitan n intercambios de llaves públicas.

Además ya se puede realizar una comunicación privada así:

- Si X envía un mensaje a Y, encriptado con X_{prv} , Y lo desencriptará con X_{pbl} , con lo cual Y estará seguro que X es el remitente, ya que el mensaje fue encriptado con X_{prv}
- Si X envía un mensaje a Y, encriptado con Y_{pbl} , Y lo desencriptará con Y_{prv} , con lo cual X estará seguro que sólo Y podrá leerlo, ya que sólo Y conoce Y_{prv}
- Al combinar ambos mecanismos, X puede crear un criptograma, en el cual se X se asegure que sólo Y puede verlo y Y estará seguro que X es el remitente, así:
 - X encriptará así:
 - $C_1 = E(T, X_{prv})$
 - $C_2 = E(C_1, Y_{pbl})$
 - Se envía C_2 a Y
 - Y desencriptará así:
 - $C_1 = D(C_2, Y_{prv})$
 - $T = D(C_1, X_{pbl})$

Estas propiedades han hecho de RSA la base de todos los protocolos de seguridad que conocemos, como PGP, HTTP, SSL, etc. Sin embargo, debido a su complejidad matemática, que involucra operaciones de exponentes y módulos con número grandes (de hasta 1024 bits o sea 2^{1024}), su rendimiento es mucho menor que el de un algoritmo simétrico. Por lo tanto, los protocolos mencionados, hacen una combinación de métodos simétricos y asimétricos, en los que éstos últimos, se utilizan sólo para intercambiar una llave simétrica temporal, generada en tiempo de corrida, al principio de la comunicación y luego la información se intercambia utilizando un algoritmo simétrico.

De esta forma se logra una privacidad completa, a través de la asimetría, con un rendimiento aceptable, a través de la simetría.

5.3.3. Criptotutorial

<http://www.ti89.com/cryptotut/rsa2.htm>

5.3.4. RSA Theory

http://www.di-mgt.com.au/rsa_theory.html

Capítulo 6

Ejercicios de examen

Alguno ejercicios resueltos de examen. Otros solo se presentan, más carecen de solución.

6.1. Deducción de $O(n)$ 1

Deduzca formalmente $O(n)$, mostrando claramente su procedimiento, para la siguiente función:

```
int recursiva1(int n)
{
    if (n<=1)
        return 5;
    else
        return recursiva1(n/2) + recursiva1(n/2);
}
```

$$T(\text{recursiva1}(n)) = T(n)$$

$$T(n) = \begin{cases} \text{si } n \leq 1 : & T(\text{comp}_{if}) + T(\text{return}) = 2t \\ \text{si } n > 1 : & T(\text{comp}_{if}) + T(\text{return}) + T(n/2) + T(n/2) \\ & = t + t + 2T(n/2) \\ & = 2t + 2T(n/2) \end{cases}$$

Expandiendo parte recursiva:

$$\begin{aligned} T(n) &= 2t + 2T(n/2) \\ &= 2t + 2[2t + 2T(n/4)] = 2t + 4t + 4T(n/4) = 6t + 4T(n/4) \\ &= 6t + 4[2t + 2T(n/8)] = 6t + 8t + 8T(n/8) = 14t + 8T(n/8) \\ &= 14t + 8[2t + 2T(n/16)] = 14t + 16t + 16T(n/16) = 30t + 16T(n/16) \\ &= 30t + 16[2t + 2T(n/32)] = 30t + 32t + 32T(n/32) = 62t + 32T(n/32) \end{aligned}$$

las partes en negrita de las expansiones puede escribirse tambien como

$$\begin{aligned} 2t &= 2t \\ 6t &= 2t + 4t \\ 14t &= 2t + 4t + 8t \\ 30t &= 2t + 4t + 8t + 16t \\ 62t &= 2t + 4t + 8t + 16t + 32t \end{aligned}$$

y para un k-ésimo término

$$t \sum_{x=1}^k 2^x$$

dado que para cada número natural n , $2^0 + 2^1 + 2^2 \dots + 2^n = 2^{n+1} - 1$, la anterior sumatoria queda

$$\begin{aligned}
 t \sum_{x=1}^k 2^x &= t(2^0 + 2^1 + 2^2 + 2^3 \dots + 2^k - 1) \\
 &= t(2^{k+1} - 1 - 1) \\
 &= t(2^{k+1} - 2) \\
 &= t(2^k 2 - 2) \\
 &= 2t(2^k - 1)
 \end{aligned}$$

el -1 restado a la ecuacion es para anular el 2^0 de la secuencia y así quede igual a la secuencia que es necesaria.

ahora para un k -ésimo término de la funcion $T(n)$ completa sería

$$T(n) = 2t(2^k - 1) + 2^k T(n/2^k) \quad (6.1)$$

condicion de salida

$$\begin{aligned}
 \frac{n}{2^k} &= 1 \\
 n &= 2^k \\
 \log_2 n &= \log_2 2^k \\
 k &= \log_2 n
 \end{aligned}$$

sustituyendo k en 6.1

$$\begin{aligned}
 T(n) &= 2t(2^{\log_2 n} - 1) + 2^{\log_2 n} T(n/2^{\log_2 n}) \\
 &= 2t(n - 1) + n T(n/n) \\
 &= 2t(n - 1) + n T(1) \\
 &= 2t(n - 1) + n (2t) \quad \backslash \text{ usando la parte no recursiva de } T(n) \\
 &= 2t(n - 1 + n) \\
 &= 2t(2n - 1) \quad \backslash \text{ por definición, tiempo minimo igual a 1 (t=1)} \\
 &= 4n - 2
 \end{aligned}$$

deduciendo $O(n)$

$$\begin{aligned}
 O[T(\text{recursiva1}(n))] &= O[4n - 2] \\
 &= \max(O(4n), O(-2)) \\
 &= O(4n) \\
 &= n
 \end{aligned}$$

es lineal.

6.2. Deducción de $O(n)$ 2

Deduzca formalmente $O(n)$, mostrando claramente su procedimiento, para la siguiente función:

```
int recursiva2(int n) {
    if (n<=1)
        return 5;
    else
        return recursiva2(n/3)
}
```

$$T(\text{recursiva2}(n)) = T(n)$$

$$T(n) = \begin{cases} \text{si } n \leq 1 : & T(\text{comp}_{if}) + T(\text{return}) = 2t \\ \text{si } n > 1 : & T(\text{comp}_{if}) + T(\text{return}) + T(\text{llamadaRecursiva}) = 2t + T(n/3) \end{cases}$$

Expansión de parte recursiva:

$$\begin{aligned} T(n) &= 2t + T(n/3) \\ &= 2t + [2t + T((n/3)/3)] = 2t + 2t + T(n/9) = 4t + T(n/9) \\ &= 4t + [2t + T((n/9)/3)] = 4t + 2t + T(n/27) = 6t + T(n/27) \\ &= 6t + [2t + T((n/27)/3)] = 6t + 2t + T(n/81) = 8t + T(n/81) \end{aligned}$$

para un k-esimo termino

$$T(n) = 2kt + T(n/3^k) \quad (6.2)$$

condicion de salida

$$\begin{aligned} \frac{n}{3^k} &= 1 \\ n &= 3^k \\ \log_3(n) &= \log_3(3^k) \\ k &= \log_3(n) \end{aligned}$$

sustituyendo k en (6.2)

$$\begin{aligned} T(n) &= 2\log_3(n)t + T(n/3^{\log_3(n)}) \\ &= 2\log_3(n)t + T(n/n) \\ &= 2\log_3(n)t + T(1) \\ &= 2\log_3(n)t + 2t \quad \backslash \text{ usando la parte no recursiva de } T(n) \\ &= 2t(\log_3(n) + 1) \\ &= 2(\log_3(n) + 1) \quad \backslash \text{ por definición, tiempo minimo igual a 1 (t=1)} \end{aligned}$$

deduciendo $O(n)$

$$\begin{aligned}
 O[T(\text{recursiva2}(n))] &= O[T(n)] \\
 &= O[2 * (\log_3 n + 1)] \\
 &= O[\log_3 n + 1] \\
 &= \max[O(\log_3 n), O(1)] \\
 &= O[\log_3 n] \\
 &= \log_3 n
 \end{aligned}$$

tiene forma logaritmica

6.3. Deducción de $O(n)$ 3

Deduzca formalmente $O(n)$, mostrando claramente su procedimiento, para la siguiente función:

```

int iterativa(int n){
    int x=1; //s1
    while (x<=n){ //s2
        x*=3;
    }
    return x; //s3
}

```

$$\begin{aligned}
 T[\text{iterativa}(n)] &= T(n) \\
 T(n) &= T(s1) + T(s2) + T(s3) \\
 T(n) &= t + T(s2) + T(s3) \\
 T(n) &= t + T(s2) + T(\text{return}) \\
 T(n) &= t + T(s2) + t \\
 T(n) &= 2t + T(s2) \\
 T(n) &= 2t + T(\text{while}) \\
 T(n) &= 2t + v * [T(\text{condicion}) + T(\text{cuerpoDelWhile})] \\
 T(n) &= 2t + v * (t + t) \\
 T(n) &= 2t + 2t * v \\
 T(n) &= 2t(1 + v)
 \end{aligned}$$

v =número de veces que se ejecuta el cuerpo y condición del while.

¿cuánto es v en términos de n ¹?

Podemos hacer un algoritmo equivalente

```
int iterativaEq(int n) {
    x=1 ;
    v=0 ;
    while (x <= n) {
        x *= 3 ;
        v = v + 1 ;
    }
    cout << "v=" << v ;
    return x;
}
```

Valores de v y x respecto a n

n	v	$x \leq n$	iteraciones
0	0	$1 \leq 0$	
1	0	$1 \leq 1, 3 \leq 1$	I
2	1	$1 \leq 2, 3 \leq 2$	I
3	2	$1 \leq 3, 3 \leq 3, 9 \leq 3$	II
4	2	$1 \leq 4, 3 \leq 4, 9 \leq 4$	II
5	2		II
6	2		II
7	2		II
8	2		II
9	3	$1 \leq 9, 3 \leq 9, 9 \leq 9, 27 \leq 9$	III
10	3	$1 \leq 10, 3 \leq 10, 9 \leq 10, 27 \leq 10$	III
11	3	$1 \leq 11, 3 \leq 11, 9 \leq 11, 27 \leq 11$	III
\vdots	\vdots	\vdots	\vdots
27	4	$1 \leq 27, 3 \leq 27, 9 \leq 27, 27 \leq 27, 81 \leq 27$	IV

Podemos deducir que existe una relación entre v y n así:

$$n < 3^v$$

por tanto

$$\log_3 n < \log_3 3^v$$

$$\log_3 n < v$$

retomando la función del tiempo anterior

$$T(n) = 2t(1 + v)$$

$$T(n) = 2t(1 + \log_3 n)$$

$$T(n) = 2(1 + \log_3 n) \quad \backslash \text{por definición, tiempo mínimo igual a 1 (t=1)}$$

¹ n representa el número de datos que recibe el algoritmo

la función $O(n)$ sería

$$\begin{aligned}
 O[\text{iterativa}(n)] &= O[T(n)] \\
 &= O[2(1 + \log_3 n)] \\
 &= O[1 + \log_3 n] \quad \backslash \text{ regla de las constantes} \\
 &= \max[O(1), O(\log_3 n)] \\
 &= O(\log_3 n) \\
 &= \log_3 n
 \end{aligned}$$

tiene forma logarítmica.

6.4. Mapeo de matriz triangular inversa

Una matriz triangular inversa se define como aquella matriz en la que las posiciones $A[i, j]$ no existan cuando $j > n - i + 1$, de modo que sólo se utiliza la memoria necesaria para los elementos que sí se usan. Deduzca formalmente la fórmula de mapeo lexicográfico $LOC(A[i, j])$ para una matriz triangular inversa de pares límites $1 \dots n$ (índice máximo n e índice mínimo 1). Por ejemplo, para el caso $n = 3$.

$A[1, 1]$	$A[1, 2]$	$A[1, 3]$
$A[2, 1]$	$A[2, 2]$	
$A[3, 1]$		

Debe representarse en memoria así:

$A[1, 1]$	$A[1, 2]$	$A[1, 3]$	$A[2, 1]$	$A[2, 2]$	$A[3, 1]$
-----------	-----------	-----------	-----------	-----------	-----------

$$\text{Loc}(i, j) = \text{ff} + \text{AvanceFilas} + \text{AvanceColumnas}$$

A simple vista se puede deducir que el avance en columnas es **j-1**.

Dado que si seleccionamos la posición $A[1, 3]$ el avance en filas es 0 y la posición del arreglo correspondiente al mapeo es la posición 2. ($3 - 1 = 2$).

Para deducir el avance en filas debemos observar el patrón, tomando como N el tamaño de la primera fila:

i	Avance
1	0
2	N
3	$N + N - 1$
4	$N + N - 1 + N - 2$
5	$N + N - 2 + N - 2 + N - 3$
k	$N + N - 2 + N - 2 + N - 3 \dots N - (k - 2)$

Lo cual se puede representar con la siguiente sumatoria:

$$\sum_{k=0}^{i-2} N - k$$

Por lo tanto la fórmula de mapeo lexicográfico para una matriz triangular inversa es:

$$Loc(i, j) = \alpha + \sum_{k=0}^{i-2} (N - k) + (j - 1)$$

6.5. Mapeo de matriz triangular

Una matriz triangular se define como aquella matriz en la que las posiciones $A[i, j]$ no existan cuando $j > i$, de modo que sólo se utiliza la memoria necesaria para los elementos que sí se usan. Deduzca la fórmula de mapeo lexicográfico $LOC(A[i, j])$ para una matriz triangulares de índice máximo n e índice mínimo 1. Por ejemplo, para el caso $n = 3$:

$A[1, 1]$		
$A[2, 1]$	$A[2, 2]$	
$A[3, 1]$	$A[3, 2]$	$A[3, 3]$

Debe representarse en memoria así:

$A[1, 1]$	$A[2, 1]$	$A[2, 2]$	$A[3, 1]$	$A[3, 2]$	$A[3, 3]$
-----------	-----------	-----------	-----------	-----------	-----------

$$\begin{aligned} \text{Loc}(A[i, j]) &= \alpha + \text{NumPosicionesAntesDeFila}(i) + \text{AvanzarAColumna}(j) \\ &= \alpha + \text{NumPosicionesAntesDeFila}(i) + (j - 1) \\ &= \alpha + \sum_{x=1}^{i-1} x + (j - 1) \end{aligned}$$

dado que para un entero n lo siguiente es cierto: $1 + 2 + 3 + 4 + \dots + n = n(n + 1)/2$. La funcion de mapeo queda

$$\begin{aligned} \text{Loc}(A[i, j]) &= \alpha + \sum_{x=1}^{i-1} x + (j - 1) \\ &= \alpha + \frac{(i - 1)[(i - 1) + 1]}{2} + (j - 1) \\ &= \alpha + \frac{(i - 1)[i - 1 + 1]}{2} + (j - 1) \\ &= \alpha + \frac{i(i - 1)}{2} + j - 1 \end{aligned}$$

6.6. Multiplicación de una matriz lexicográfica.

Dado el siguiente código:

```
using namespace std;

class Matriz{
public:
    Matriz(int n[], int m[]); /* constructor con los pares
    límites */
    virtual ~Matriz(); /* destructor */

    /* se ponen y obtienen valores */
    int &operator[] (int i[]);

    /* crea una nueva matriz, que la multiplicacion de esta
    matriz por el m */
    Matriz *operator* (Matriz m);
};

int main(){
    cout << "Multiplicacion" << endl;

    /* se define la matriz m1: 10..12 * 1..4 */
    Matriz m1({10,1},{12,4});

    /* se define la matriz m2: 11..14 * 1..5*/
    Matriz m2({11,1}, {14,5});

    /* se inicializan algunas posiciones */
    m1[10,2]=1;
    m1[12,4]=1;
    m2[12,3]=1;
    m2[14,4]=1;

    /* se obtiene la matriz resultado de la multiplicacion
    * m3=10..12 * 1..5
    */

    Matriz *m3=m1*m2;

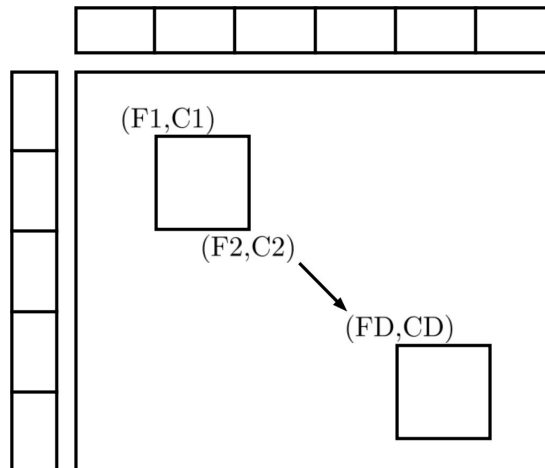
    /* se imprime la matriz resultante */
    for (int i=10; i<=12; i++) {
        for (int j=1; j<=5; j++) {
            cout << "["<<i<<","<<j<<"]="<<m3[{i,j}]; << " -- ";
        }
        cout << endl;
    }
}
```

```
delete m3;  
return 0;  
}
```

- a) Desarrolle los métodos necesarios en la clase Matriz para la implementación de la multiplicación (**operator***) para una matriz lexicográfica.
- b) Deduzca $O(n)$ para el método **operator***.

6.7. Matriz esparcida

Una aplicación de hoja electrónica, maneja la matriz de celdas como una matriz esparcida, con dos vectores de apuntadores a elementos de lista ortogonal. Uno de los vectores se indiza de la A a la Z y sirve para puntos de entrada a las columnas, el otro se indiza de 1 a 256 y sirve para puntos de entrada para las filas. Escriba una rutina **Move(int F1, char C1, int F2, char C2, int FD, char CD)** que mueva el rango especificado por las esquinas **(F1, C1)** y **(F2, C2)** al rango que comienza en **(FD, CD)** solamente moviendo las celdas de lugar, es decir, sin crear nuevos nodos.



6.8. Árbol de expresiones

Dada una clase **ArbolExp**, que representa un árbol de expresiones, escriba un método **valor()** que de como resultado el valor de la expresión representada. Puede suponer que sólo existen los operadores de suma, resta, multiplicación y división. También se supone que todos los operandos son constantes, es decir no hay variables. No debe utilizar ninguna estructura de datos adicional. Suponga que sólo existen las siguientes variables en cada clase:

```
class ArbolExp{
public:
    ...
    int valor();
    ...
private:
    NodoExp *raiz;
    ...
};

class NodoExp{
    ...
    void *valor; //constante u operador
    NodoExp *izq;
    NodoExp *der;
    ...
};
```

La solución es la siguiente:

```
int ArbolExp::valor() {
    return valor(raiz);
}

int ArbolExp::valor(NodoExp* n) {
    char * s;

    if (n==NULL)
        return -1;

    s=(char *) n->valor; //se asume que valor apunta a una
                          cadena

    if (strcmp(s, "*")==0)
        return valor(n->izq) * valor(n->der);
    else if (strcmp(s, "/"==0)
        return valor(n->izq) / valor(n->der);
    else if (strcmp(s, "+")==0)
        return valor(n->izq) + valor(n->der);
    else if (strcmp(s, "-")==0)
        return valor(n->izq) - valor(n->der);
    else
        return atoi(s);
}
```


6.9. Tabla de Hash

Explique los cuatro elementos generales de una tabla de dispersión.

Tabla: En esta se guarda los elementos del universo.

Función Hash: También conocida como funcion de dispersión y da la posición en la tabla para una llave pasada como parámetro.

Colisiones: Ocurre cuando dos llaves distintas dan posiciones de tabla coincidentes. Se debe adoptar alguna política de resolución de colisiones, como encadenamiento simple por ejemplo, para solucionarlo.

Universo: Son elementos que posiblemente se añadirán en la tabla y no necesariamente ocupan espacio de memoria.

6.10. Direcccionamiento abierto de doble dispersión

En una tabla de dispersión de tamaño $M = 13$, con $h(llv) = llv \bmod M$, $S(llv, i) = (llv \bmod 7 + 1) * i$, y utilizando direccionamiento abierto de doble dispersión, realice la inserción de las siguientes llaves: 15,26, 29,2,18,28.

- Tamaño $M=13$
- Función de dispersión $h(llv) = llv \bmod M$
- Función de doble dispersión $S(llv, i) = (llv \bmod 7 + 1)$

LLave	$h(llv)$	$h(llv) + S(llv, 1)$	$h(llv) + S(llv, 2)$	Posición final
16	$15 \bmod 13=2$	No hay colisión	No hay colisión	2
26	$26 \bmod 13=0$	No hay colisión	No hay colisión	0
29	$19 \bmod 13=3$	No hay colisión	No hay colisión	3
2	$2 \bmod 13=2$	$2+(2 \bmod 7 + 1)*1=5$	No hay colisión	5
18	$18 \bmod 13=5$	$5+(18 \bmod 7 + 1)*1=10$	No hay colisión	10
28	$28 \bmod 13=2$	$2+(28 \bmod 7 + 1)*1=3$	$2+(28 \bmod 7 + 1)*2=4$	4

Figura 6.1: Insertando valores

26
15
29
28
2
28

Figura 6.2: Tabla resultante

6.11. Búsqueda de texto

Describe paso a paso, las distintas comparaciones o corrimientos, que se realizaría al buscar el patrón "ABDCDE" en el texto "ABXDEEEBDEAABDE" usando la búsqueda de Booyer-Moore.

6.12. Criptografía

Con una técnica de criptografía pública, definimos $E(P_{pbl}, T)$ como el encriptado del texto T con llave pública P_{pbl} de la persona P y $E(P_{prv}, T)$ como el encriptado del texto T con la llave privada P_{prv} de la persona P . También definimos $D(P_{pbl}, C)$ como el desencriptado del criptograma C con la llave pública P_{pbl} de la persona P y $D(P_{prv}, C)$ como el desencriptado del criptograma C con la llave privada P_{prv} de la persona P . Si una persona X desea enviar un mensaje a un sujeto Y , de modo que X esté seguro que sólo Y podrá leer el mensaje y Y esté seguro que sólo X pudo haber enviado el mensaje. Indique cómo X deberá encriptar el mensaje y cómo Y deberá desencriptarlo para lograr este objetivo.

La **solución** es la siguiente: la persona X encripta así:

$$\begin{aligned} C_1 &= E(X_{prv}, T) \\ C_2 &= E(Y_{pbl}, C_1) \end{aligned}$$

la persona Y desencripta así:

$$\begin{aligned} C_1 &= D(Y_{prv}, C_2) \\ T &= D(X_{pbl}, C_1) \end{aligned}$$