Universidad de Santiago de Chile
Facultad de Ingeniería
Departamento de Ingeniería Informática

APUNTES DE LA ASIGNATURA

ANÁLISIS DE ALGORITMOS Y ESTRUCTURA DE DATOS

Mag. Jacqueline Köhler C.

TABLA DE CONTENIDOS

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

ÍNDICE DE ALGORITMOS

1 ANÁLISIS DE ALGORITMOS

1.1 CONCEPTOS PREVIOS DE ALGORITMOS

Un algoritmo es una secuencia de instrucciones que se debe seguir para resolver un determinado problema, que además garantiza la obtención de una respuesta al terminar. Debe ser preciso y detallado, es decir, no usa intuición ni creatividad, ni tampoco requiere de decisiones subjetivas. Los algoritmos se remontan hasta muy atrás en la historia. El más antiguo conocido, el algoritmo para determinar el máximo común divisor de dos números, se atribuye a Euclides.

En el párrafo anterior se mencionó que los algoritmos garantizan la obtención de una respuesta al terminar de seguir sus pasos para resolver un problema dado. Esto se traduce, entonces, en que los algoritmos ofrecen una solución general para un problema dado que esté acotado a un dominio determinado. Se denomina instancia de un problema a un caso particular de éste, y un algoritmo debe funcionar correctamente para todas las instancias del problema, sin considerar las restricciones propias de la implementación: capacidad de memoria, rangos de valores, etc. No obstante, debe ser escrito de forma tal que sea claro determinar la forma de implementarlo en algún lenguaje de programación.

1.2 COMPLEJIDAD DE ALGORITMOS

En muchas ocasiones ocurre que un problema puede ser resuelto por muchos algoritmos. Entonces surge el problema de escoger cuál de ellos utilizar. Si solo se requiere resolver unas pocas instancias sencillas del problema, no importa cuál sea el algoritmo seleccionado. No obstante, si la instancia a resolver es muy difícil o bien se requiere resolver muchas instancias, es importante escoger el mejor. Existen dos enfoques para determinar cuál algoritmo escoger:

- Empírico (*a posteriori*): consiste en programar todas las alternativas, probarlas con diversas instancias y comparar su desempeño. Por supuesto, este enfoque resulta muy poco práctico.
- Teórico (*a priori*): consiste en determinar matemáticamente cuántos recursos se necesitan para cada algoritmo. Esta tarea se denomina estudio de la complejidad y consiste en la obtención de una expresión matemática que refleja el comportamiento del algoritmo en función de los datos de entrada.

La complejidad o eficiencia de un algoritmo debe ser estudiada con respecto a algún recurso crítico: el tiempo de computación (cuánto tiempo toma ejecutar la implementación de un algoritmo), la cantidad de espacio necesaria para almacenar los datos requeridos, la cantidad de procesadores necesarios, etc. En general, la complejidad de tiempo es mayor que la complejidad de espacio, ya que cualquier información almacenada debe ser leída o guardada en algún instante. Por ese motivo, generalmente se trabaja con la complejidad de tiempo.

Se señaló anteriormente que la complejidad de un algoritmo se determina en función de los datos de entrada. En efecto, se debe buscar una expresión matemática en función del tamaño de dichos datos. Formalmente, el tamaño de un dato se refiere a la cantidad de bits necesarios para su almacenamiento. No obstante, esa noción depende de la implementación del algoritmo, por lo que no es adecuada para una evaluación teórica. En consecuencia, se considera simplemente que el tamaño es un número natural que

indica la cantidad de elementos de un objeto, por ejemplo, la cantidad de elementos de un arreglo o bien el número de nodos y aristas de un grafo.

El comportamiento real de un algoritmo depende de muchos factores propios de su implementación. Entre ellos se encuentran las restricciones de arquitectura, es decir, el compilador o intérprete utilizado, el lenguaje de programación y la máquina en que se ejecute. También influyen algunos factores humanos, como la forma de escribir el código fuente o las estructuras de datos seleccionadas. En consecuencia, se hace necesario evaluar el comportamiento del algoritmo en forma independiente de factores humanos y de máquina.

Ahora surge el problema de determinar la eficiencia de un algoritmo de algún modo estándar. Si el tiempo de ejecución depende de la implementación, no es posible medirlo en segundos, pues no todos los computadores tardan lo mismo en ejecutar un mismo programa. Este problema puede ser resuelto gracias al principio de invariancia, que indica que dos implementaciones diferentes de un mismo algoritmo no difieren en su eficiencia más que en una constante multiplicativa c. Así, es posible expresar la complejidad en términos del tamaño de la entrada como $c \cdot t(n)$. Más adelante, con el estudio de la notación asintótica, se verá que es posible ignorar la constante y decir, simplemente, que el algoritmo está en el orden de $t(n)$. En algunos casos, la complejidad recibe un nombre especial. Un algoritmo es:

- Lineal si está en el orden de n.
- Cuadrático se está en el orden de n^2.
- Polinomial si está en el orden de n^k.
- Exponencial si está en el orden de c^n.

Otro aspecto importante al evaluar la complejidad de un algoritmo es que frecuentemente los problemas tienen diversas instancias de un mismo tamaño y el tiempo que tarda un algoritmo en resolverlas puede variar mucho. En consecuencia, no tiene sentido evaluar la eficiencia de un algoritmo solo en función del tamaño de la instancia, pues el algoritmo debe funcionar correctamente para todas las instancias. En el mejor escenario, el tiempo requerido suele ser poco (y no será considerado en este curso). Sin embargo, si el tiempo de respuesta es crítico, es necesario considerar el tiempo requerido para el peor escenario posible. También puede ser útil analizar el caso promedio, aunque resulta muy difícil pues esto requiere conocer a priori la distribución de los casos a resolver, por lo que el análisis de la complejidad del caso medio no será considerado en este curso.

1.2.1 LA BÚSQUEDA DE LA EFICIENCIA

Es sabido que los computadores son cada vez más rápidos. Esto podría conducir a la idea de que la eficiencia de los algoritmos es cada vez menos importante. Sin embargo, no necesariamente es así. Suponga que se tiene un algoritmo cuadrático exponencial A_1 para resolver un problema, una máquina M_1 que tarda $2n^2$ segundos en resolver una instancia de tamaño n y una máquina M_2 que tarda n^2 segundos en resolver la misma instancia. La figura 1.1 muestra el incremento del tiempo de ejecución del algoritmo A_1 para M_1 (en azul) y M_2 (en rojo) a medida que aumenta el tamaño de la instancia. Se puede observar el mismo tipo de curva para el incremento del tiempo de ejecución en ambas máquinas.

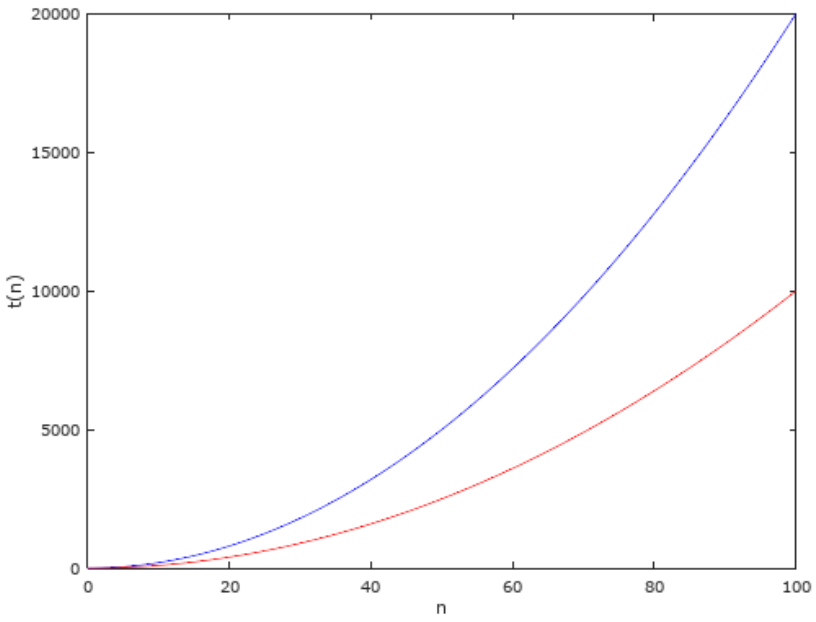


FIGURA 1.1: Comparación del desempeño de un algoritmo en dos máquinas diferentes.

Ahora considere que se descubre un algoritmo lineal A_2 para resolver el mismo problema, y que M_1 tarda $4n + 23$ segundos en resolver una instancia de tamaño n. La figura 1.2 muestra el tiempo de ejecución de los algoritmos A_1 (en azul) y A_2 (en rojo) en la máquina M_1. Es evidente que, en general, el algoritmo lineal es más eficiente que el cuadrático. No obstante, en instancias del problema en que $n \leq 4$ resulta más veloz el algoritmo cuadrático. En situaciones críticas, podría ser adecuado combinar ambas alternativas para usar la mejor de acuerdo al tamaño de la instancia del problema a resolver.

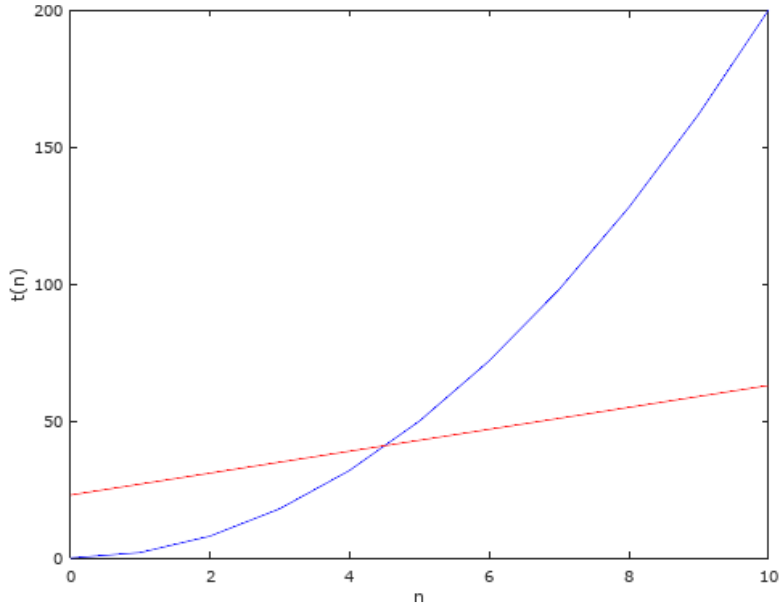


FIGURA 1.2: Comparación del desempeño de dos algoritmos en una misma máquina.

1.2.2 CÁLCULO DE LA COMPLEJIDAD

Se denomina operación elemental o instrucción básica a una instrucción cuyo tiempo de ejecución está acotado superiormente por una constante que solo depende de la implementación usada (la máquina que ejecute el programa, el lenguaje de programación empleado, etc.), por lo que se asume que tienen un tiempo de ejecución unitario. El ejemplo 1.1 muestra una justificación para esta simplificación.

EJEMPLO 1.1:

> Considere que un algoritmo requiere efectuar a operaciones de adición, m operaciones de multiplicación y s operaciones de asignación. Además, se sabe que una suma nunca requiere más de t_a microsegundos, una multiplicación nunca requiere más de t_m microsegundos y una asignación nunca requiere más de t_2 segundos.
>
> Por la definición anterior se tiene que el tiempo t de ejecución del algoritmo está acotado por $t \leq at_a + mt_m + st_s$. Esto quiere decir que $t \leq \max\{t_a, t_m, t_s\} \times (a + m + s)$. Pero se sabe que el tiempo requerido por cada operación elemental no es importante pues se descartan las constantes, por lo que se asume que pueden ser ejecutadas en tiempo unitario. Por tanto, se puede afirmar que el algoritmo en cuestión es del orden de $a + m + s$.

Formalizando la idea del ejemplo 1.1, se tiene que, para un algoritmo dado, la complejidad de tiempo queda determinada por el tiempo de ejecución de cada instrucción básica y el número de veces que se ejecuta esa instrucción.

Sean:

- A: un algoritmo para resolver un problema dado,
- $I_1, \ldots, I_k$: una secuencia de instrucciones básicas del algoritmo A,
- t_j: el tiempo de ejecución de la instrucción I_j, $1 \leq j \leq k$,
- f_j: el número de veces que se ejecuta I_j, $1 \leq j \leq k$.

La complejidad de tiempo del algoritmo A está dada por:

$$T_A = \sum_j f_j \cdot t_j$$

Como todas las instrucciones básicas requieren un tiempo constante, se puede suponer que esos tiempos son iguales, es decir: $t_1 = t_2 = \cdots = t_k = 1$. Luego, se tiene:

$$T_A = \sum_j f_j$$

Es decir, la complejidad de tiempo corresponde al número total de instrucciones básicas necesarias de ejecutar para una computación completa del algoritmo.

EJEMPLO 1.2:

El problema de la suma de matrices consiste en, dadas $A, B \in M_{(n\times m)}$, calcular $C = A + B \in M_{(n\times m)}$. Podemos resolver este problema usando el algoritmo 1.1.

Complejidad:

- Aquí se observan dos ciclos Para y una asignación, que es una instrucción básica.
- El primer ciclo Para se ejecuta n veces.
- Además, el ciclo Para anidado dentro del primero se ejecuta m veces.
- Luego, la instrucción básica anidada en los dos ciclos se ejecuta m veces (por el segundo ciclo) por las n veces que se ejecuta el primer ciclo.
- La asignación tiene lugar en un tiempo t.

ALGORITMO 1.1: Suma matricial.

```
SUMA_MATRICIAL(A[n][m]: número, B[n][m]: número): número[n][m]
Para i desde 1 hasta n:
            Para j desde 1 hasta m:
                  C[i][j] = A[i][j] + B[i][j]

Devolver C
```

La complejidad de tiempo del algoritmo dado queda definida por:

$$T_A = m \times n \times t$$

Asumiendo $t = 1$:

$$T_A = m \times n$$

En particular, si $m = n$:

$$T_A = n^2$$

Del ejemplo 1.2 se desprenden las siguientes conclusiones, que se resumen en la figura 1.3:

- Las instrucciones simples se ejecutan en un tiempo constante que se aproxima a 1.
- Las instrucciones condicionales se consideran como una instrucción simple.
- Los ciclos iterativos se ejecutan la cantidad de veces que tienen establecido repetirse. Esto es, si el ciclo Para repite sus instrucciones desde 1 a n, su tiempo es n.
- Los ciclos iterativos anidados se ejecutan en su tiempo correspondiente, multiplicado por las veces que se repita el ciclo en el cual están anidados.

EJEMPLO 1.3:

Problema: buscar un valor en un arreglo de enteros. Podemos resolver este problema usando el algoritmo 1.2.

Para determinar el tiempo de ejecución del algoritmo, es necesario calcular la cantidad de operaciones elementales que se realizan:

- En la línea 1 se ejecuta 1 operación (asignación).

- En la línea 2 se ejecuta la condición del ciclo, conformada por un total de 4 operaciones: dos comparaciones, un acceso al arreglo y un y lógico.
- La línea 3 cuenta con dos operaciones: un incremento y una asignación.
- La línea 4 está conformada por una comparación y un acceso al vector arreglo (2 operaciones).
- La línea 5 contiene una operación de retorno si la condición se cumple.
- Análogamente, la línea 7 contiene una operación de retorno si la condición no se cumple.

En el mejor caso, solo se ejecuta la línea 1 y la primera mitad de la condición de la línea 2, es decir, solo 2 operaciones. A continuación, se ejecutan las líneas 5 a 7. Así, para el mejor caso:

$$T(n) = 1 + 2 + 3 = 6$$

En el peor caso, se ejecuta la línea 1 y el ciclo se repite hasta que se cumple la segunda condición. Cada iteración del ciclo está dada por las instrucciones de las líneas 2 y 3 más una ejecución adicional de la línea 2 (para salir del ciclo). La ejecución termina con la línea 7. Así, para el peor caso se tiene que:

$$T(n) = 1 + \left(\left(\sum_{i=1}^{n-1}(4+2)\right) + 4\right) + 2 + 1 = 6n + 2$$

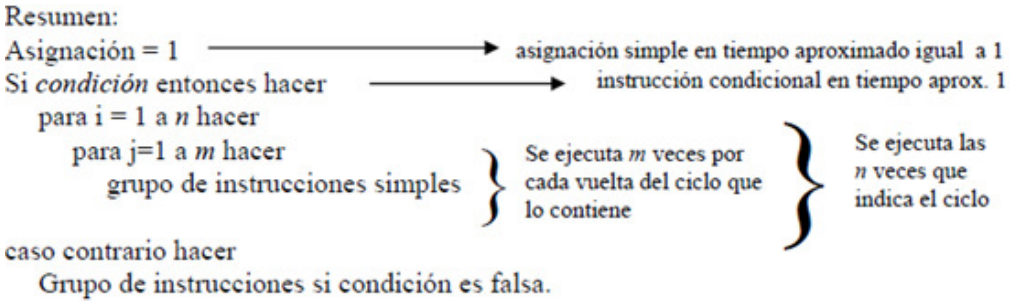


FIGURA 1.3: Resumen para el cálculo de complejidad computacional.

ALGORITMO 1.2: Búsqueda de un entero en un arreglo.

```
BUSCAR (A[n]: entero, c: entero): entero
      j ← 1
      Mientras A[j] < c y j < n hacer:
            j ← j+1
      Si A[j] = c hacer:
            devolver j
      En otro caso hacer:
            devolver 0
```

1.3 MODELO DE MÁQUINA

Se utiliza el modelo RAM (Máquina de Acceso Aleatorio, *Random Access Machine* en inglés), definido por el esquema de la figura 1.4.

Se denomina entrada a los datos del problema que se entregan al algoritmo y salida a la respuesta entregada por éste.

El análisis de complejidad de un algoritmo se hace respecto al mejor caso (sus entradas más favorables en cuanto al tiempo requerido), al peor caso (las entradas menos favorables) y al caso medio.
Antes de seguir adelante, es necesario conocer algunas definiciones:

- Paso: tiempo de una instrucción básica.
- Complejidad local: número total de pasos de la ejecución completa del algoritmo para una entrada dada.
- Complejidad asintótica: menor cota superior de la complejidad local.
- Complejidad del mejor caso: tiempo para ejecutar el algoritmo con la entrada más favorable.
- Complejidad del peor caso: tiempo en ejecutar el algoritmo con la entrada más desfavorable.
- Complejidad del caso medio: tiempo promedio en ejecutar el algoritmo con todas las entradas posibles.

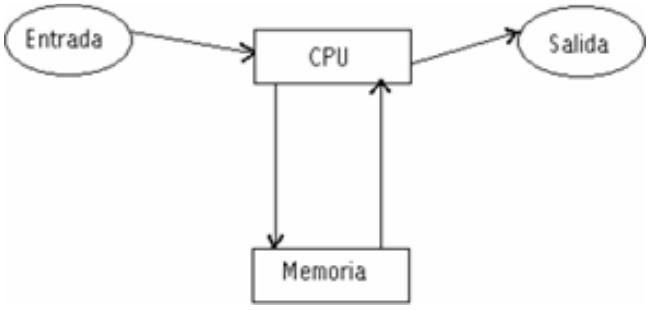


FIGURA 1.4: Modelo RAM.

Dado un algoritmo A para resolver un problema π, sean:

- $E_1, E_2, E_3, \dots, E_k$ todas las posibles entradas,
- $T_1, T_2, T_3, \dots, T_k$ tiempo de ejecutar A para la entrada j, $1 \leq j \leq k$,
- $P_1, P_2, P_3, \dots, P_k$ probabilidad de que ocurra la entrada j, $1 \leq j \leq k$.

La complejidad del mejor caso es $C_{MC} = \min\{T_j : 1 \leq j \leq k\}$,

La complejidad del peor caso es $C_{PC} = \max\{T_j : 1 \leq j \leq k\}$,

La complejidad del caso medio está dada por:

$$\sum_j P_j T_j$$

La complejidad más utilizada para confrontar los rendimientos de los algoritmos es la del peor caso. Una de las razones es que da una cota para todas las entradas, incluyendo aquellas muy malas, que el análisis del caso promedio no puede ofrecer. Otra razón es que la complejidad del caso medio es más difícil de calcular, dado que es necesario conocer todas las posibles entradas del algoritmo y sus probabilidades.

1.4 NOTACIÓN O

Dadas $f: \mathbb{R} \to \mathbb{R}^+$ y $g: \mathbb{R} \to \mathbb{R}^+$, se dice que f es del orden de g, denotado por $f \sim O(g)$, si existen constantes c y n_0 tales que $f(n) \leq c \cdot g(n) \forall n \geq n_0$. En otras palabras, f está acotada inferiormente por $c \cdot g(n)$ para todo valor $n \geq n_0$. La figura 1.5 muestra esta explicación de manera gráfica.

EJEMPLO 1.4:

$3n + 1 \sim O(n)$
$3n^2 + n \sim O(n^2)$
$5n^4 \sim O(n^4)$
$10 \cdot 2^n \sim O(2^n)$

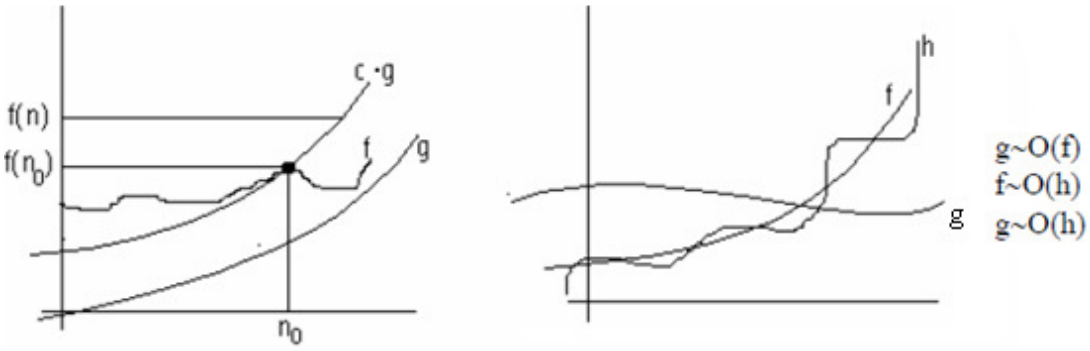


FIGURA 1.5: Ejemplo de notación O.

Propiedades de la notación O:

1. $O(f) + O(g) = O(f + g)$ — Regla de la suma
2. $O(k \cdot f) = k \cdot O(f) = O(f), k\ cte.$ — Constantes multiplicativas
3. $O(f) \cdot O(g) = O(f \cdot g)$ — Regla de la multiplicación
4. $f \geq g \Rightarrow O(f + g) = O(f)$ — Cotas
5. $f \geq g \Rightarrow \big(h \sim O(g)\ entonces\ h \sim O(f)\big)$ — Transitividad

Observaciones:

- La notación O desprecia las constantes aditivas y multiplicativas.
- La medida de la complejidad es una aproximación al tiempo real que el algoritmo requiere para encontrar la solución. Por ello aparece el símbolo ~ en lugar de =.

1.5 OTRAS NOTACIONES

Además de la notación O, existen otras notaciones asintóticas que ayudan a acotar la complejidad de un algoritmo. Si bien no serán consideradas en este curso, algunas de ellas son ampliamente utilizadas en el ámbito académico y de investigación, por lo que es importante darlas a conocer como base para cursos más avanzados y trabajos futuros.

- Notación Ω: Dadas $f: \mathbb{R} \to \mathbb{R}^+$ y $g: \mathbb{R} \to \mathbb{R}^+$, se dice que f es omega de g, denotado por $f \sim \Omega(g)$, si existen constantes c y n_0 tales que $f(n) \geq c \cdot g(n) \forall\, n \geq n_0$. En otras palabras, f está acotada inferiormente por $c \cdot g(n)$ para todo valor $n \geq n_0$.

- Notación Θ: Dadas $f: \mathbb{R} \to \mathbb{R}^+$ y $g: \mathbb{R} \to \mathbb{R}^+$, si $f \sim O(g)$ y $f \sim \Omega(g)$ a la vez, se dice que $f \sim \Theta(g)$. Expresado de otro modo, $f \sim \Theta(g)$ si existen constantes c_1, c_2 y n_0 tales que $c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)\ \forall\, n \geq n_0$. Es decir, f está acotada inferiormente por $c_2 \cdot g(n)$ y superiormente por $c_1 \cdot g(n)$ para todo valor $n \geq n_0$.
- Notación o: Dadas $f: \mathbb{R} \to \mathbb{R}^+$ y $g: \mathbb{R} \to \mathbb{R}^+$, se dice que f es o (también llamada o chica u o minúscula) de g, denotado por $f \sim o(g)$, si $f(n) \leq c \cdot g(n) \forall\, n \geq n\_0$ y $\forall c > 0$. En otras palabras, f está acotada inferiormente por $c \cdot g(n)$ para todo valor $n \geq n_0$.

1.6 COMPLEJIDAD DE ALGORITMOS SECUENCIALES E ITERATIVOS

Anteriormente se estudió la forma de estimar el tiempo de ejecución para un algoritmo dado mediante la contabilización de las instrucciones que se ejecutan ante un escenario dado. Para determinar la complejidad del algoritmo, basta con aplicar las cotas descritas sobre la fórmula obtenida para el tiempo de ejecución.

Así, para el ejemplo 1.3 se tiene que:

$C_{MC} \sim O(6) = 1$
$C_{PC} \sim O(6n + 2) = O(n)$

EJEMPLO 1.5:

El problema de multiplicar dos matrices $A \in M_{(n \times p)}$ y $A \in M_{(p \times m)}$ puede ser resuelto por medio del algoritmo 1.3.

ALGORITMO 1.3: Multiplicación de dos matrices.

```
MULTIPLICACION_MATRICIAL(A[n][p]: número, B[p][m]: número): número[n][m]
Para i desde 1 hasta n:
            Para j desde 1 hasta m:
                  C[i][j] = A[i][1] + B[1][j]

                  Para k desde 2 hasta p:
                        C[i][j] = C[i][j] + A[i][k] + B[k][j]

Devolver C
```

Complejidad:

- Aquí se observan tres ciclos Para anidados.
- El primer ciclo Para se ejecuta n veces.
- Por cada vez que se ejecuta el primer ciclo, el segundo ciclo Para se ejecuta m veces.
- Por cada vez que se ejecuta el segundo ciclo, se tiene una instrucción básica y un tercer ciclo que se ejecuta $p - 1$ veces.

Así, la complejidad de tiempo queda definida por $T(m, n, p) \sim O(m \cdot n \cdot p)$. En el caso particular en que $m = n = p$, se tiene entonces $T(n) \sim O(n^3)$.

En este caso, la complejidad del mejor caso es igual a la complejidad del peor caso y a la complejidad del caso medio, ya que cualquiera sea la entrada el algoritmo ejecuta todos los pasos.

Hay un aspecto del cálculo de la complejidad que suele dar origen a confusión, y está ligado a la noción de instrucción básica. Había quedado establecido que las instrucciones básicas son aquellas cuyo tiempo de ejecución está acotado superiormente por una constante que solo depende de la implementación usada. En otras palabras, pueden asociarse a las operaciones unarias y binarias que incorpora el procesador (asignación, cambio de signo, operaciones aritméticas, operaciones lógicas, comparaciones, etc.). Sin embargo, en los ejemplos anteriores pareciera que no se hubiese considerado la condición de la sentencia Si o de los ciclos Para y Mientras, o bien que se hubiese considerado como instrucción simple a alguna sentencia que en realidad contiene varias operaciones.

En efecto, si se tiene una sentencia como:

```
x = 3 * (5 + x) / (-y - 6)
```

Al momento de compilar o interpretar el programa se traduce en una serie de sentencias (los términos t_j corresponden a variables temporales creadas por el compilador o intérprete):

```
t0 = 5 + x
t1 = 3 * t0
t2 = -y
t3 = t2 - 6
x = t1 / t3
```

No obstante, como se asume que cada una de ellas tiene un tiempo de ejecución unitario y la notación asintótica descarta las constantes aditivas y multiplicativas, es posible ignorar este aspecto.

En el caso de las comparaciones y operaciones lógicas se observa el mismo fenómeno. Considere la expresión:

```
x != 3 && x < 8
```

Cuando se compila o interpreta el programa, esto se traduce en una secuencia de operaciones de la forma (los números agregados al comienzo de cada instrucción simplemente permiten identificar cada línea para dirigir los saltos):

```
0: si x != 3 saltar a 3
1: t0 = falso
2: saltar a 4
3: t0 = verdadero
4: si x < 8 saltar a 7
5: t1 = falso
6: saltar a 8
7: t1 = verdadero
8: t2 =t0 and t1
```

No obstante, como aún no han sido abordadas las instrucciones de máquina en otras asignaturas, las constantes aditivas o multiplicativas serán en adelante reemplazados por una c.

EJEMPLO 1.6:

El factorial de un número natural n se define por la multiplicación de números naturales sucesivos desde 1 hasta n: $n! = 1 \cdot 2 \cdot 3 \cdot ... \cdot n$. Puede ser calculado iterativamente como se muestra en el algoritmo 1.4.

ALGORITMO 1.4: Algoritmo iterativo para el cálculo del factorial de un entero no negativo.

```
FACTORIAL(n: entero no negativo): entero positivo
      factorial = 1

      Para i desde 2 hasta n:
            factorial = factorial * i

      Devolver factorial
```

Complejidad:

- Asignación con tiempo constante.
- Ciclo Para que se ejecuta $n - 1$ veces.
- Dentro del ciclo, una asignación con tiempo constante.
- Sentencia de retorno con tiempo constante.

Con los tiempos de cada instrucción y por las propiedades de la notación O, la complejidad del algoritmo 1.4 es $T(n) = c + c \cdot (n - 1) + c = c \cdot (n - 1) + c$. Aplicando las propiedades de la notación O se tiene que $T(n) \sim O(n)$.

Otro aspecto importante a tener en cuenta al determinar la complejidad de un algoritmo tiene que ver con la llamada a funciones.

EJEMPLO 1.7:

El algoritmo 1.5 calcula la sumatoria de los factoriales de los n primeros números naturales, usando para ello la función definida en el algoritmo 1.4.

ALGORITMO 1.5: Algoritmo que suma los factoriales de los n primeros naturales.

```
SUMATORIA_DE_FACTORIALES(n: entero no negativo): entero positivo
      suma = 0

      Para i desde 1 hasta n:
            suma = suma + FACTORIAL(i)

      Devolver suma
```

Complejidad:

- Asignación con tiempo constante.
- Ciclo Para que se ejecuta n veces.
- Dentro del ciclo, una suma con tiempo constante y una llamada a la función que calcula el factorial para la que es sabido que su complejidad es $O(n)$ (por lo que se puede considerar que su tiempo de ejecución es n).
- Sentencia de retorno con tiempo constante.

Así, la complejidad del algoritmo 1.5 es $T(n) = c + n \cdot (c + n) + c = n^2 + c \cdot n + c$. Aplicando las propiedades de la notación O se tiene que $T(n) \sim O(n^2)$.

La utilidad del uso de la complejidad aparece en su mayor expresión cuando se necesita optar por alguna alternativa entre diferentes algoritmos para resolver un problema determinado. Para esto, se debe saber discriminar entre un algoritmo y otro, para poder tomar la mejor decisión.

EJEMPLO 1.8:

Suponga que la entrada de cierto problema π tiene tamaño n y que, para resolverlo se encuentra en la literatura las siguientes alternativas:

a. Algoritmo de tiempo $O(n^3)$.
b. Algoritmo de tiempo $O(n) + O(n \log n)$.
c. Algoritmo de tiempo $O(n^2)$.

Se debe elegir la alternativa cuyo valor de función sea menor, esto es, porque el algoritmo elegido debe ser el más rápido, es decir que ocupe menor tiempo de ejecución. Luego, la alternativa elegida es la b.

Ahora que ya se tienen las herramientas para analizar la complejidad de un algoritmo, es pertinente dar algunas definiciones adicionales relativas a la eficiencia de algoritmos:

- Un algoritmo es eficiente si su complejidad es un polinomio de la entrada, es decir, es de la forma $T(n) \sim O(n \cdot k)$, donde k es una constante.
- Para algunos problemas es posible determinar el límite inferior del problema que corresponde al tiempo mínimo para resolverlo. Este límite es una característica del problema y refleja su dificultad intrínseca para resolverlo. Cualquier algoritmo que lo resuelva utilizará por lo menos ese tiempo.
- Un algoritmo es óptimo si su complejidad es $\sim O(L)$, donde L es el límite inferior del problema.

1.7 COMPLEJIDAD DE ALGORITMOS RECURSIVOS

Se denomina recursión al proceso de resolver un problema grande descomponiéndolo en uno o más subproblemas que son de estructura idéntica a la del problema original pero más simples de resolver que aquel. Cada subproblema se resuelve de forma similar, repitiendo la descomposición hasta generar subproblemas tan simples que se resuelven sin necesidad de descomponerlos. La solución del problema original se obtiene combinando las soluciones de los subproblemas en un proceso inverso a la descomposición realizada.

La recursión está muy ligada a la recurrencia matemática, que es cuando se define una función f en términos de ella misma. Dos ejemplos clásicos de recurrencia matemática son el cálculo del factorial y los números de Fibonacci, que se muestran en los ejemplos 1.9 y 1.10, respectivamente.

Cabe destacar que la recursividad es una propiedad de la estrategia para resolver un problema y no del problema en sí. Así pues, se dice que un algoritmo es recursivo si corresponde a un procedimiento que se invoca a sí mismo hasta recibir como entrada un caso base.
En programación, la recursión puede darse de dos formas:

- Directa: en algún paso del conjunto de instrucciones aparece una invocación al propio procedimiento.
- Indirecta: el procedimiento llama a otro procedimiento, y este último llama, a su vez, al primero. Asimismo, si un procedimiento llama a otros y, en algún momento, alguno de ellos llama al primero, también es un caso de recursión indirecta.

En toda definición recursiva de un problema se debe establecer una condición de borde, es decir, una condición que no requiera otra definición recursiva para su resolución. Ésta condición determina el criterio de parada del algoritmo recursivo. Otro aspecto importante a considerar es que la complejidad de un algoritmo recursivo se expresa mediante una ecuación de recurrencia.

EJEMPLO 1.9:

La función recurrente para el factorial es:

$$Factorial(n) = \begin{cases} n \cdot (n-1)! & n \geq 1 \\ 1 & n = 0 \end{cases}$$

Puede implementarse recursivamente como muestra el algoritmo 1.6.

ALGORITMO 1.6: Algoritmo recursivo para el cálculo del factorial de un entero no negativo.

```
FACTORIAL(n: entero no negativo): entero positivo
      Si n == 0 ó n == 1:
            Devolver 1
      Sino:
            Devolver n * FACTORIAL(n - 1)
```

La ecuación de recurrencia para el tiempo de ejecución del algoritmo 1.6 es:

$$T(n) = \begin{cases} T(n-1) + c & n > 0 \\ c & n = 0 \end{cases}$$

Si se resuelve esa ecuación por sustitución se encuentra:

$$
\begin{aligned}
T(n) &= T(n-1) + c = \\
&= (T(n-2) + c) + c = T(n-2) + c = \\
&= \big((T(n-3) + c) + c\big) + c = T(n-3) + c = \\
&\quad \ldots \\
&= T(n-k) + c = \\
&\quad \ldots \\
&= T(1) + n - 1 = \\
&= T(0) + n = n + c
\end{aligned}
$$

Luego, la complejidad del Algoritmo Factorial Recursivo es $\sim O(n)$.

En el caso del factorial, la complejidad del algoritmo iterativo es igual a la del algoritmo recursivo. No obstante, esto no siempre es así. Para algunos problemas es más eficiente el algoritmo recursivo; para otros, el algoritmo iterativo es mejor. Más aún, existen problemas para los cuales es imposible desarrollar un algoritmo iterativo que no sea una simulación de la recursión. La secuencia de Fibonacci del ejemplo 1.10 es un caso en que el algoritmo iterativo es más eficiente.

EJEMPLO 1.10:

La función recurrente para determinar los términos de la sucesión de Fibonacci es:

$$
Fib(n) = \begin{cases} Fib(n-1) + Fib(n-2) & n \geq 2 \\ 1 & n = 1 \\ 1 & n = 0 \end{cases}
$$

Puede implementarse recursivamente como muestra el algoritmo 1.7.

ALGORITMO 1.7: Algoritmo recursivo para el enésimo término de la sucesión de Fibonacci.

```
FIBONACCI(n: entero no negativo): entero positivo
      Si n == 0 ó n == 1:
            Devolver 1
      Sino:
            Devolver FIBONACCI(n - 2) + FIBONACCI(n - 1)
```

La ecuación de recurrencia para el tiempo de ejecución del algoritmo 1.7 es:

$$
T(n) = \begin{cases} T(n-1) + T(n-2) & n > 0 \\ c & n = 0 \end{cases}
$$

La resolución de dicha ecuación resulta bastante compleja, por lo que la complejidad de este algoritmo será determinada más adelante.

A modo de contraste, el algoritmo 1.8 muestra una solución iterativa para el enésimo término de la sucesión de Fibonacci.

ALGORITMO 1.8: Algoritmo iterativo para el enésimo término de la sucesión de Fibonacci.

```
FIBONACCI(n: entero no negativo): entero positivo
      Si n == 0 ó n == 1:
            Devolver 1

      penúltimo = 1
      último = 1

      Para i desde 2 hasta n:
            término = penúltimo + último
            penúltimo = último
            último = término

      Devolver término
```

De acuerdo a los ejemplos realizados hasta ahora, se puede establecer que el tiempo de ejecución para el algoritmo 1.8 está dado por:

$$T(n) = \begin{cases} T(n-1)+1 & n > 0 \\ 1 & n = 0 \end{cases}$$

Si se resuelve esa ecuación por sustitución se encuentra:

$T(n) = c \cdot (n-c) + c = c \cdot n + c \sim O(n)$.

Dado que un algoritmo recursivo se invoca a sí mismo, al expresar el orden $T(n)$ de su complejidad se obtiene una ecuación de recurrencia. Esta ecuación depende del proceso de descomposición del problema en subproblemas.

Reducción por sustracción: si el tamaño n del problema decrece en una cantidad constante b en cada llamada, se realizan a llamadas recursivas y las operaciones correspondientes a la parte no recursiva del algoritmo toman un tiempo $O(n^k)$, entonces:

$T(n) = aT(n-b) + O(n^k)$ si $n \geq b$

Una observación importante es que se considera un tiempo polinomial para la resolución de un problema solamente porque interesa que la solución recursiva sea eficiente. Un tiempo exponencial, por ejemplo, sería mucho más costoso que un caso iterativo.

La solución de esta ecuación de recurrencia es de la forma:

$$O(\quad) = \begin{cases} n^k & si\ a < 1 \\ n^{k+1} & si\ a = 1 \\ a^{\frac{n}{b}} & si\ a > 1 \end{cases}$$

EJEMPLO 1.11:

La ecuación de recurrencia para el algoritmo 1.7 puede expresarse como:

$$T(n) = T(n-1) + T(n-2) + O(1)$$

Pero es sabido que $T(n-2) \leq T(n-1)$, por lo que es válido expresar la ecuación como:

$$T(n) = 2T(n-1) + O(n^0)$$

De donde se tiene que: $a = 2$, $b = 1$ y $k = 0$. Así, aplicando la fórmula, $T(n) \sim O(2^n)$. Esto demuestra que el algoritmo iterativo es significativamente más eficiente, pues es de orden lineal, mientras que el recursivo es exponencial.

Reducción por división: si el algoritmo con un problema de tamaño n realiza a llamadas recursivas con subproblemas de tamaño n/b y las operaciones correspondientes a la parte no recursiva, que corresponde a descomponer el problema en los subproblemas y luego combinar las soluciones de éstos, toman un tiempo $O(n^k)$ entonces:

$T(n) = aT\left(\frac{n}{b}\right) + O(n^k)$ si $n \geq b$

La solución de esta ecuación de recurrencia es de la forma:

$$O(\quad) = \begin{cases} n^k & si\ a < b^k \\ n^k \log n & si\ a = b^k \\ n^{\log_b a} & si\ a > b^k \end{cases}$$

2 NOCIONES DE C: ARREGLOS, PUNTEROS Y ESTRUCTURAS

Schildt (1999) define un arreglo como "una colección de variables del mismo tipo que se referencian por un nombre común". Cada elemento está contenido en una posición específica del arreglo, por lo que esta estructura puede ser recorrida mediante un índice.

Habitualmente los elementos de un arreglo son almacenados en posiciones contiguas de memoria, donde la posición más baja de memoria corresponde al primer elemento del arreglo y la posición más alta, al último.

Se pueden implementar arreglos tanto de forma estática como dinámica. Desde luego, ambos enfoques tienen sus ventajas y desventajas.

2.1 ARREGLOS ESTÁTICOS

Este tipo de implementación almacena el arreglo en una posición fija de memoria, por lo que es necesario conocer la máxima cantidad de elementos que puede contener (tamaño) y la cantidad de memoria que ocupa cada uno de sus elementos. En consecuencia, es necesario conocer también el tipo de datos que contendrá el arreglo. En este tipo de implementación, la creación de la estructura de datos y la gestión de memoria son transparentes para el programador. Además, la cantidad de memoria necesaria para almacenar el arreglo se asigna en tiempo de compilación y estará contenida en el bloque de memoria estática asignado al programa.

En C, la declaración de arreglos se realiza de la siguiente manera:

```
tipo nombreDeVariable[tamaño];
```

En C (y en muchos otros lenguajes), el índice para el primer elemento es el 0, por lo que el último elemento de un arreglo de tamaño 10 estará indexado en la posición 9.

El ejemplo 2.1 muestra un programa sencillo en C (al que se han agregado números de línea) donde se muestra el trabajo con un arreglo de enteros implementado estáticamente. Algunos elementos interesantes del programa son los siguientes:

- En la línea 1 se importa la biblioteca de entrada y salida estándar de C.
- En la línea 2 se hace uso de la directiva #define para declarar la constante MAX.
- La línea 5 muestra la declaración de un arreglo de enteros de tamaño MAX.
- El ciclo `for` de las líneas 8 a 10 recorre el arreglo e ingresa en cada posición los enteros del 1 al 5, respectivamente.
- La línea 12 contiene una sentencia para mostrar por pantalla el tamaño (en bytes) del arreglo. Para este fin se hace uso de la función `sizeof()`. Note que, si bien el arreglo contiene 5 elementos, su tamaño en memoria es de 20 bytes (considerando enteros de 4 bytes cada uno).
- Las líneas 13 a 17 muestran por pantalla los elementos del arreglo, dejando a continuación una línea en blanco.

EJEMPLO 2.1:

```
 1 #include <stdio.h>
 2 #define MAX 5
 3 void main() {
 4
 5     int ejemplo[MAX];
 6     int i;
 7
 8     for(i=0; i<MAX; i++) {
 9         ejemplo[i] = i+1;
10     }
11
12     printf("Tamano en memoria del arreglo: %d\n\n", sizeof(ejemplo));
13     printf("%d", ejemplo[0]);
14
15     for(i=1; i<MAX; i++) {
16         printf(" %d", ejemplo[i]);
17     }
18
19     printf("\n\n");
20 }
```

La figura 2.1 muestra cómo se vería en memoria el arreglo del ejemplo 2.1 si comienza en la posición 1500 de la memoria.

Elemento	ejemplo[0]	ejemplo[1]	ejemplo[2]	ejemplo[3]	ejemplo[4]
Contenido	1	2	3	4	5
Dirección	1500	1504	1508	1512	1516

FIGURA 2.1: Arreglo de enteros del ejemplo 2.1.

2.2 PUNTEROS

Antes de comenzar a estudiar las siguientes estructuras de datos básicas es necesario conocer el funcionamiento de los punteros. En el caso de C, el manejo de punteros y de la memoria asociada debe ser explicitado por el programador, a diferencia de lo que ocurre en otros lenguajes de programación como Java y Python.

Hasta ahora, los ejemplos estudiados solo contenían variables estáticas, vale decir, el nombre de la variable está asociado a una posición fija de la memoria de un tamaño previamente definido. Así, por ejemplo, para un carácter se emplea 1 byte y para un entero, 4 bytes.

Un puntero, en cambio, es una variable estática cuyo contenido es una posición de memoria, como muestra la figura 2.2. Inicialmente, un puntero no apunta a ninguna posición de memoria, por lo que debe ser inicializado en nulo (NULL). Esta inicialización es imprescindible puesto que, al declarar una variable de tipo puntero, C no elimina lo que pueda haber existido previamente en dicha celda de memoria, por lo que en la práctica el puntero podría estar apuntando a cualquier parte y, en consecuencia, provocar errores graves durante la ejecución del programa.

Otra cosa importante de señalar es que un puntero siempre tiene un tipo de dato asociado, por lo que solo puede hacer referencia (apuntar) a valores de ese tipo.

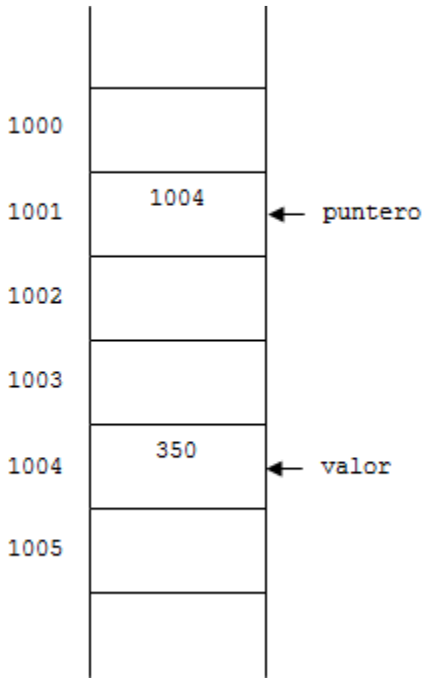


FIGURA 2.2: Almacenamiento en memoria de una variable y un puntero a ella.

Los punteros tienen algunos operadores asociados en C:
- Dirección de (&): permite obtener la posición en memoria de un elemento.
- Indirección (*): muestra lo que está contenido en una determinada posición de memoria.

El ejemplo 2.2 muestra un código fuente en C que permite declarar y asignar las variables de la figura 2.2, además de mostrar algunos elementos por pantalla. Para indicar que una variable es un puntero a algún tipo de dato, basta con anteponer el operador * a su nombre.

EJEMPLO 2.2:

```
#include <stdio.h>

void main() {
    int valor, *puntero;
    valor = 350;
    puntero = &valor;

    printf("valor: %d. puntero: %d\n", valor, puntero);
    printf("El valor apuntado por puntero: %d\n", *puntero);
}
```

Otro uso de los punteros es para trabajar con arreglos dinámicos. Al trabajar con arreglos estáticos, el nombre del arreglo está asociado a la dirección en memoria del primer elemento del arreglo. Así, por ejemplo, al hacer la declaración `int arreglo[20];` el nombre arreglo está asociado a la posición de

memoria donde comienza el elemento `arreglo[0]`, es decir, la relación `arreglo == arreglo[0]` es verdadera. Esto se abarca con mayor profundidad en la sección siguiente.

2.3 ARREGLOS DINÁMICOS

En ocasiones puede ser necesario crear arreglos asignando memoria en forma dinámica, es decir, que la cantidad de memoria utilizada pueda ser solicitada (y liberada) explícitamente por el programador y pueda variar durante la ejecución del programa. Para esto se hace necesario usar punteros en la declaración del arreglo, aunque para trabajar con sus elementos se utiliza la indexación del mismo modo que en arreglos estáticos.

Otro aspecto relevante es que, al trabajar con arreglos dinámicos, la cantidad de memoria se asigna en tiempo de ejecución y no está contenida en la porción de memoria asignada al programa, sino en la memoria dinámica o *heap*, la cual contiene elementos de tamaño variable.

El ejemplo 2.3 muestra un programa en C que trabaja con arreglos dinámicos. Algunos elementos relevantes del programa del ejemplo 2.3 son:

- En la línea 2 se importa la biblioteca estándar de C, que permite hacer uso de las funciones para gestión de memoria.
- La línea 10 define un puntero a un entero, pero le asigna memoria de acuerdo al tamaño de arreglo ingresado por el usuario (`tamanoInicial`). La función encargada de asignar memoria es `malloc()`, cuyo argumento corresponde a la cantidad de bytes a asignar. La función `sizeof()` determina la cantidad de memoria que utiliza cada entero, por lo que para almacenar `tamanoInicial` enteros se requieren `tamanoInicial * sizeof(int)` bytes. una alternativa para `malloc()` es `calloc()`, que además de asignar memoria, inicializa los campos con ceros.
- El código de las líneas 14 a 17 comprueba que haya sido posible asignar al puntero p la cantidad de memoria solicitada. En caso de no ser exitosa la asignación, envía un mensaje al usuario y termina la ejecución del programa. Lo mismo ocurre con el código de las líneas 34 a 37.
- La línea 32 reasigna memoria al arreglo p, cambiando la cantidad de memoria asignada de acuerdo a un nuevo tamaño de arreglo establecido por el usuario. para esto es necesario cambiar el segmento de memoria en que está contenido el arreglo.
- Una buena práctica al trabajar con memoria dinámica (y punteros) es explicitar que un puntero a una posición liberada de memoria queda nulo (líneas 50 y 51) para evitar errores en tiempo de ejecución.

2.4 ESTRUCTURAS

Es frecuente en programación querer representar datos complejos, que están conformados por dos o más datos que pueden ser de distintos tipos. Estos datos compuestos suelen denominarse registros o estructuras. En el caso de C, esto se logra por medio de las palabras reservadas `struct`, para definir los datos involucrados, y `typedef` para asignarle un nombre al nuevo tipo de dato. El ejemplo 2.4 muestra la creación del tipo de dato persona, cuyos datos son nombre, RUT y edad, y un programa principal que permite crear una nueva persona y luego mostrarla por pantalla.

EJEMPLO 2.3:

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3
 4 void main() {
 5     int tamanoInicial;
 6     int tamanoFinal;
 7     int i;
 8     int *p;
 9
10     printf("Ingrese tamano del arreglo: ");
11     scanf("%d", &tamanoInicial);
12     p = (int *) malloc(tamanoInicial * sizeof(int));
13
14     if(!p) {
15         printf("ERROR: No se pudo asignar memoria.");
16         exit(0);
17     }
18
19     for(i = 0; i < tamanoInicial; i++) {
20         printf("Ingrese un valor entero: ");
21         scanf("%d", &p[i]);
22     }
23
24     printf("\n");
25     for(i = 0; i < tamanoInicial;i++) {
26         printf("%d ", p[i]);
27     }
28
29     printf("\n\n");
30     printf("Ingrese nuevo tamano del arreglo: ");
31     scanf("%d", &tamanoFinal);
32     realloc(p, tamanoFinal * sizeof(int));
33
34     if(!p) {
35         printf("ERROR: No se pudo asignar memoria.");
36         exit(0);
37     }
38
39     for(i = tamanoInicial; i < tamanoFinal; i++) {
40         printf("Ingrese un valor entero: ");
41         scanf("%d", &p[i]);
42     }
43
44     printf("\n");
45     for(i = 0; i < tamanoFinal; i++) {
46         printf("%d ", p[i]);
47     }
48
49     printf("\n\n");
50     free(p);
51     p = NULL;
52 }
```

EJEMPLO 2.4:

Crear un tipo de dato persona, que tenga los campos nombre, RUT y edad. Construya un programa principal que cree una nueva persona y luego muestre sus datos por pantalla.

En términos conceptuales, la solución para este problema puede representarse en pseudocódigo como muestra el algoritmo 2.1.

ALGORITMO 2.1: Creación de un tipo de dato compuesto y función de prueba.

```
Tipo persona:
      nombre: string
      rut: string
      edad: entero

PRINCIPAL():
      persona p
      p.nombre = pedir("Ingrese el nombre de la persona: ")
      p.rut = pedir("Ingrese el RUT de la persona: ")
      p.edad = pedir("Ingrese la edad de la persona: ")
      imprimir(p.nombre, " de RUT ", p.rut, " tiene ", p.edad, " años.")
```

En C, la implementación del algoritmo 2.1 se realiza de la siguiente manera:

```
#include <stdio.h>
#include <stdlib.h>

typedef struct s {
    char nombre[15];
    char rut[14];
    int edad;
} persona;

void main() {
    persona p;
    printf("Ingrese el nombre de la persona: ");
    scanf("%s", &p.nombre);
    printf("Ingrese el RUT de la persona: ");
    scanf("%s", &p.rut);
    printf("Ingrese la edad de la persona: ");
    scanf("%d", &p.edad);
    printf("%s, de RUT %s, tiene %d agnos.", p.nombre, p.rut, p.edad);
}
```

Una observación importante es que la definición del tipo persona podría también hacerse como sigue:

```
struct s {
    char nombre[15];
    char rut[14];
    int edad;
};

Typedef struct s persona;
```

En este último caso, se separan las tareas de crear la estructura de datos y de asignarle un tipo de dato a dicha estructura.

3 ALGORITMOS ELEMENTALES

Las tareas de ordenar información de acuerdo a algún criterio o de buscar algún elemento en alguna colección lineal de datos (tales como arreglos y listas) son muy frecuentes, por lo que es fundamental contar algoritmos que resuelvan estos problemas.

3.1 ALGORITMOS DE ORDENAMIENTO

Un problema fundamental (y muy frecuente, por cierto) es el de ordenar una lista (o arreglo) de datos según algún criterio lineal (orden alfabético, de menor a mayor, etc.). En el caso de un arreglo de datos atómicos (enteros, caracteres, etc.), resulta sencillo establecer el criterio de ordenamiento. No obstante, esta tarea puede resultar más compleja al trabajar con tipos de datos compuestos (estructuras). En estos casos, una estrategia habitual es la de escoger un campo como clave (por ejemplo, la edad de una persona) y ordenar de acuerdo a dicho campo (de más joven a más anciana).

Si bien el problema de ordenamiento es bastante sencillo en su definición, puede abordarse de maneras diferentes dependiendo de si el listado de datos a ordenar se encuentra en la memoria principal (ordenamiento interno), con la ventaja de su capacidad de acceso aleatorio, o si se encuentra en algún tipo de almacenamiento externo (ordenamiento externo), con la desventaja de tener que mover grandes volúmenes de datos entre la memoria principal y el otro dispositivo de almacenamiento.

Al momento de evaluar el tiempo de ejecución de los algoritmos de ordenamiento se pueden emplear diversos criterios, los cuales deben ser escogidos de acuerdo a la aplicación que requiera del ordenamiento.

- Contar la cantidad de pasos del algoritmo para ordenar n registros.
- Contar la cantidad de comparaciones entre llaves que se requieren para ordenar n registros. Este esquema es adecuado cuando la comparación entre claves es una operación costosa en tiempo (por ejemplo, si son cadenas muy largas de caracteres).
- Si el tamaño de cada registro es muy grande, también se debe tener en cuenta cuántas veces debe moverse un registro.

Se presentan aquí los algoritmos más sencillos para llevar a cabo el proceso de ordenamiento interno, aunque existen otros muchos.

3.1.1 ORDENAMIENTO DE BURBUJA

Suele considerarse como el método más simple para ordenar una lista de datos. Su nombre viene de la metáfora de las burbujas de jabón, donde las más grandes suben más rápido que las más pequeñas. Esta metáfora, llevada a la noción de ordenar una lista de datos, requiere en primer lugar imaginar que la lista está representada verticalmente. Así, el elemento más grande es el primero en "subir" hasta el final de la lista. A continuación, el segundo elemento más grande queda en la penúltima posición, y así sucesivamente. Para esto se recorre el arreglo repetidas veces, intercambiando elementos adyacentes que se encuentren en desorden, como se puede apreciar en el algoritmo 3.1.

ALGORITMO 3.1: Ordenamiento de burbuja.

```
BURBUJA(arreglo[n]: tipoDato): tipoDato[n]
      Para i desde n - 1 hasta 1:
            Para j desde 0 hasta i - 1:
                  Si arreglo[j] > arreglo[j + 1]:
                        arreglo = INTERCAMBIAR(arreglo, j, j + 1)

      Devolver arreglo

INTERCAMBIAR(arreglo[n]: tipoDato, i: entero, j: entero): tipoDato[n]
      Si i != j:
            auxiliar = arreglo[i]
            arreglo[i] = arreglo[j]
            arreglo[j] = auxiliar

      Devolver arreglo
```

La complejidad del peor caso para el ordenamiento de burbuja se determina considerando lo siguiente:

- En el peor caso de la sentencia Si se ejecuta una cantidad constante de operaciones: $O(1)$.
- El ciclo interior se ejecuta $n - i$ veces e incluye la sentencia Si: $O(n) \cdot O(1) = O(n)$.
- El ciclo exterior se ejecuta $n - 1$ veces e incluye al ciclo interior: $O(n) \cdot O(n) = O(n^2)$.

Así, se tiene que la complejidad del peor caso para el ordenamiento por burbuja es $O(n^2)$.

El ejemplo 3.1 presenta la traza de la ejecución del código del ejemplo 3.1 para un arreglo dado. Se subrayan en cada caso los elementos que podrían aparecer intercambiados al término de la iteración siguiente.

EJEMPLO 3.1:

```
arreglo inicial: 8 2 5 7 1
i: 4, j: 0, arreglo: 2 8 5 7 1
i: 4, j: 1, arreglo: 2 5 8 7 1
i: 4, j: 2, arreglo: 2 5 7 8 1
i: 4, j: 3, arreglo: 2 5 7 1 8
i: 3, j: 0, arreglo: 2 5 7 1 8
i: 3, j: 1, arreglo: 2 5 7 1 8
i: 3, j: 2, arreglo: 2 5 1 7 8
i: 2, j: 0, arreglo: 2 5 1 7 8
i: 2, j: 1, arreglo: 2 1 5 7 8
i: 1, j: 0, arreglo: 1 2 5 7 8
```

3.1.2 ORDENAMIENTO POR SELECCIÓN

Este método se origina en la idea de que, para cada posición de la lista o arreglo de datos, se escoge el mejor valor posible. En otras palabras, selecciona el elemento más adecuado para situarlo en la posición que le corresponde. El algoritmo 3.2 muestra el ordenamiento por selección, haciendo uso de la operación intercambiar definida en el algoritmo 3.1.

ALGORITMO 3.2: Ordenamiento por selección.

```
SELECCIÓN(arreglo[n]: tipoDato): tipoDato[n]
      Para i desde 0 hasta n – 2:
            índiceMejor = i
            valorMejor = arreglo[i]

            Para j desde i + 1 hasta n – 1:
                  Si arreglo[j] < valorMejor:
                        índiceMejor = j
                        valorMejor = L[j]

            Si índiceMejor != i:
                  arreglo = INTERCAMBIAR(arreglo, i, índiceMejor)

      Devolver arreglo
```

Tras analizar el algoritmo, se tiene que la complejidad del peor caso para el ordenamiento por selección es $O(n^2)$, debido a los ciclos PARA anidados.

El ejemplo 3.2 presenta la traza de la ejecución del ordenamiento por selección para un arreglo dado.

EJEMPLO 3.2:

```
arreglo inicial: 8 7 5 2 1
i: 0, j: 1, índiceMejor: 1, valorMejor: 7, arreglo: 8 7 5 2 1
i: 0, j: 2, índiceMejor: 2, valorMejor: 5, arreglo: 8 7 5 2 1
i: 0, j: 3, índiceMejor: 3, valorMejor: 2, arreglo: 8 7 5 2 1
i: 0, j: 4, índiceMejor: 4, valorMejor: 1, arreglo: 1 7 5 2 8
i: 1, j: 2, índiceMejor: 2, valorMejor: 5, arreglo: 1 7 5 2 8
i: 1, j: 3, índiceMejor: 3, valorMejor: 2, arreglo: 1 7 5 2 8
i: 1, j: 4, índiceMejor: 3, valorMejor: 2, arreglo: 1 2 5 7 8
i: 2, j: 3, índiceMejor: 2, valorMejor: 5, arreglo: 1 2 5 7 8
i: 2, j: 4, índiceMejor: 2, valorMejor: 5, arreglo: 1 2 5 7 8
i: 3, j: 4, índiceMejor: 3, valorMejor: 7, arreglo: 1 2 5 7 8
```

3.1.3 ORDENAMIENTO POR INSERCIÓN

Este método de ordenamiento opera separando la lista de elementos en dos partes: una ya ordenada y la otra, desordenada. Inicialmente, se considera que solo el primer elemento está ordenado. Luego, en cada iteración, toma el primer elemento del fragmento desordenado y lo inserta (moviéndolo por medio de intercambios) en la posición que le corresponde del fragmento ordenado, como muestra el algoritmo 3.3.

Tras analizar el algoritmo, se tiene que la complejidad del peor caso para el ordenamiento por inserción es $O(n^2)$, debido al ciclo Mientras anidado dentro del ciclo Para.

El ejemplo 3.3 muestra la traza del ordenamiento por inserción.

ALGORITMO 3.3: Ordenamiento por inserción.

```
INSERCIÓN(arreglo[n]: tipoDato): tipoDato[n]
      Para i desde 1 hasta n – 1:
            j = i
            Mientras arreglo[j] < arreglo[j-1] y j >= 1:
                  arreglo = INTERCAMBIAR(arreglo, j, j – 1)
                  j = j-1
      Devolver arreglo
```

EJEMPLO 3.3:

```
arreglo inicial: 8 7 5 2 1
i: 1, j: 1, arreglo: 7 8 5 2 1
i: 2, j: 2, arreglo: 7 5 8 2 1
i: 2, j: 1, arreglo: 5 7 8 2 1
i: 3, j: 3, arreglo: 5 7 2 8 1
i: 3, j: 2, arreglo: 5 2 7 8 1
i: 3, j: 1, arreglo: 2 5 7 8 1
i: 4, j: 4, arreglo: 2 5 7 1 8
i: 4, j: 3, arreglo: 2 5 1 7 8
i: 4, j: 2, arreglo: 2 1 5 7 8
i: 4, j: 1, arreglo: 1 2 5 7 8
```

3.1.4 QUICKSORT

Ampliamente considerado como el mejor algoritmo de ordenamiento interno, el ordenamiento rápido o *quicksort* se basa en escoger un valor dentro del arreglo como pivote en torno al cual se ordenan los elementos restantes: luego de escoger el pivote, se intercambian elementos de forma tal que todos los valores menores que el pivote queden a su izquierda y los mayores, a su derecha. Por último, se aplica *quicksort* de manera recursiva a los subarreglos situados a cada lado del pivote. El caso base es cuando se tienen arreglos con menos de dos elementos. El algoritmo 3.4 muestra una forma de implementar éste algoritmo de ordenamiento.

Tras analizar el algoritmo 3.4, puede verse que se trata de una reducción por reducción. Se puede suponer que ambos subarreglos tienen el mismo tamaño. Al evaluar este algoritmo, se tiene que la complejidad del peor caso es $O(n \log n)$.

Otra consideración importante es que no existe un único mecanismo para seleccionar el pivote. No obstante, es importante que la estrategia empleada no entorpezca el rendimiento general del algoritmo. Hay versiones que usan como pivote el último elemento del arreglo. En esta, la estrategia es algo más compleja y busca asegurar que exista al menos un elemento menor que el pivote.

El ejemplo 3.4 muestra la traza de *quicksort* para un arreglo dado.

ALGORITMO 3.4: Ordenamiento rápido.

```
QUICKSORT(arreglo[n]: tipoDato, inicio: entero, fin: entero): tipoDato[n]
      Si inicio < fin:
            índicePivote = BUSCAR_PIVOTE(arreglo, inicio, fin)
            corte = PARTICIONAR(arreglo, inicio, fin, índicePivote)
            arreglo = QUICKSORT(arreglo, inicio, corte - 1)
            arreglo = QUICKSORT(arreglo, corte + 1, fin)
            Devolver arreglo

BUSCAR_PIVOTE(arreglo[n]: tipoDato, inicio: entero, fin: entero): entero
      Para i desde inicio + 1 hasta fin:
            Si arreglo[i] > arreglo[inicio]:
                  Devolver i
            Sino si arreglo[i] < arreglo[inicio]:
                  Devolver inicio

      Devolver fin

PARTICIONAR(arreglo[n]: tipoDato, inicio: entero, fin: entero,
            índicePivote: entero): entero
      pivote = arreglo[índicePivote]
      arreglo = INTERCAMBIAR(arreglo, índicePivote, fin)
      índicePivote = fin
      izquierda = inicio
      derecha = fin - 1

      Hacer:
            Arreglo = INTERCAMBIAR(arreglo, izquierda, derecha)

            Mientras arreglo[izquierda] < pivote:
                  izquierda = izquierda + 1
            Mientras arreglo[derecha] >= pivote:
                  derecha = derecha - 1
      Mientras izquierda < derecha

      arreglo = INTERCAMBIAR(arreglo, izquierda, índicePivote)
      Devolver izquierda
```

3.2 ALGORITMOS BÁSICOS DE BÚSQUEDA

Otro problema frecuente es el de buscar un elemento en una lista de datos. No obstante, la solución de este problema puede mejorar considerablemente cuando se tienen datos ordenados.

3.2.1 BÚSQUEDA SECUENCIAL

El algoritmo más elemental para buscar un elemento en una lista (algoritmo 3.5), y el único que sirve cuando ésta se encuentra desordenada, es la búsqueda secuencial, vale decir, recorrer la lista elemento por elemento hasta encontrar la primera aparición del elemento buscado. Se entrega como respuesta la

posición del elemento. (si se alcanza el final de la lista sin encontrar el elemento buscado, se devuelve `NULO`).

EJEMPLO 3.4:

```
arreglo inicial:  6 2 7 0 9 4 5 3 8 1
inicio: 0, fin: 9, pivote: 6, arreglo 1 2 7 0 9 4 5 3 8 6
inicio: 0, fin: 9, pivote: 6, arreglo 8 2 7 0 9 4 5 3 1 6
inicio: 0, fin: 9, pivote: 6, arreglo 1 2 7 0 9 4 5 3 8 6
inicio: 0, fin: 9, pivote: 6, arreglo 1 2 3 0 9 4 5 7 8 6
inicio: 0, fin: 9, pivote: 6, arreglo 1 2 3 0 5 4 9 7 8 6
inicio: 0, fin: 9, pivote: 6, arreglo 1 2 3 0 5 4 6 7 8 9
inicio: 0, fin: 5, pivote: 2, arreglo 1 4 3 0 5 2
inicio: 0, fin: 5, pivote: 2, arreglo 5 4 3 0 1 2
inicio: 0, fin: 5, pivote: 2, arreglo 1 4 3 0 5 2
inicio: 0, fin: 5, pivote: 2, arreglo 1 0 3 4 5 2
inicio: 0, fin: 5, pivote: 2, arreglo 1 0 2 4 5 3
inicio: 0, fin: 1, pivote: 1, arreglo 0 1
inicio: 0, fin: 1, pivote: 1, arreglo 0 1
inicio: 0, fin: 1, pivote: 1, arreglo 0 1
inicio: 3, fin: 5, pivote: 5, arreglo 4 3 5
inicio: 3, fin: 5, pivote: 5, arreglo 3 4 5
inicio: 3, fin: 5, pivote: 5, arreglo 3 4 5
inicio: 3, fin: 4, pivote: 4, arreglo 3 4
inicio: 3, fin: 4, pivote: 4, arreglo 3 4
inicio: 3, fin: 4, pivote: 4, arreglo 3 4
inicio: 7, fin: 9, pivote: 8, arreglo 7 9 8
inicio: 7, fin: 9, pivote: 8, arreglo 9 7 8
inicio: 7, fin: 9, pivote: 8, arreglo 7 9 8
inicio: 7, fin: 9, pivote: 8, arreglo 7 8 9
arreglo final:  0 1 2 3 4 5 6 7 8 9
```

ALGORITMO 3.5: Búsqueda secuencial.

```
BÚSQUEDA_SECUENCIAL(arreglo[n]: tipoDato, buscado: tipoDato): entero
      Para i desde 0 hasta n - 1:
            Si arreglo[i] == buscado:
                  Devolver i
      Devolver NULO
```

El análisis de este algoritmo indica que, en el peor de los casos, su complejidad es $O(n)$. Cabe destacar que `NULO` no es un entero, pero al momento de la implementación basta con escoger un valor fuera de rango (por ejemplo, un número negativo).

3.2.2 BÚSQUEDA SECUENCIAL ORDENADA

El método anterior puede ser levemente mejorado cuando la lista está ordenada. Si, por ejemplo, el orden es de menor a mayor, la búsqueda debe detenerse al momento de encontrar el primer elemento mayor al buscado (cuando este último no está en la lista).

ALGORITMO 3.6: Búsqueda secuencial ordenada.

```
BÚSQUEDA_SECUENCIAL_ORDENADA(arreglo[n]: tipoDato, buscado: tipoDato): entero
      Para i desde 0 hasta n – 1:
            Si arreglo[i] == buscado:
                  Devolver i
            Si arreglo[i] > buscado:
                  Devolver NULO
      Devolver NULO
```

El análisis de este algoritmo indica que, en el peor de los casos y a pesar de la mejora introducida, su complejidad es $O(n)$.

3.2.3 BÚSQUEDA BINARIA

Este algoritmo de búsqueda ofrece una mejora sustancial con respecto a la búsqueda lineal ordenada, pues divide el conjunto de búsqueda a la mitad en cada iteración. Para este fin, se compara el elemento buscado con el elemento central, descartando la parte de la estructura de datos en que no se encuentra el elemento buscado.

ALGORITMO 3.7: Búsqueda binaria.

```
BÚSQUEDA_BINARIA(arreglo[n]: tipoDato, buscado: tipoDato): entero
      primero = 0
      último = n – 1

      Mientras(último – primero) > 0:
            centro = ⌊(último + primero) / 2⌋

            Si arreglo[centro] == buscado:
                  Devolver centro
            Sino:
                  Si arreglo[centro] < buscado:
                        primero = centro + 1
                  Sino:
                        último = centro – 1

      Si arreglo[primero] == buscado:
            Devolver primero
      Sino:
            Devolver NULO
```

El análisis de este algoritmo indica que, en el peor de los casos, su complejidad es $O(\log n)$. Esto se debe a que, en cada iteración, se descarta la mitad del arreglo (reducción por división).

4 LISTAS

Según Aho *et al.* (1987), las listas, el primer TDA a estudiar en este curso, son especialmente flexibles y ampliamente utilizadas "porque pueden crecer o achicarse según demanda, y los elementos pueden ser accedidos, insertados o eliminados en cualquier posición al interior de la lista. Además, pueden ser concatenadas para conformar una única lista o separadas en sublistas".

4.1 TIPOS DE DATOS ABSTRACTOS

Anteriormente se definió la noción de estructura de datos como un dato compuesto en que cada elemento no es necesariamente del mismo tipo que los demás. Esta noción sirve para tratar como un único dato a un conjunto de ellos que guardan una relación determinada. Por ejemplo, los datos marca, modelo, año, color y patente pueden considerarse como parte de un dato mayor que represente a un auto. La definición de una estructura de datos es, conceptualmente, muy similar a la idea de definir los atributos de una clase en el paradigma orientado a objetos (POO).

Si bien la noción de estructura de datos resulta muy útil para representar la información en un programa, en incontables ocasiones es necesario definir un conjunto determinado de funciones u operaciones asociadas a dicha estructura (en POO, los métodos de la clase). En el paradigma imperativo, la creación de estas operaciones lleva al concepto de tipo de dato abstracto (TDA). Así, un TDA está conformado por una estructura de datos y por un conjunto de operaciones que definen su funcionamiento.

La importancia de los TDA radica en que, por una parte, permiten construir herramientas para trabajar con grandes cantidades de datos y, por otra, permiten definir datos complejos de manera conceptual, independiente de la implementación, para posteriormente emplearlos en la resolución de problemas más complejos.

4.2 DEFINICIÓN DE LISTA

4.2.1 DEFINICIÓN MATEMÁTICA

Matemáticamente, una lista es una secuencia de 0 o más elementos de un determinado tipo. Habitualmente se denotan mediante una secuencia de elementos separados por comas:

$a_0, a_1, \ldots, a_{n-1} \qquad n \geq 0$

Donde cada a_i es del tipo definido (por ejemplo, si se define una lista de enteros, cada a_i debe ser un entero).

Se dice que el tamaño o longitud de la lista corresponde a la cantidad n de elementos que la conforman. Si $n = 0$, se tiene una lista vacía (sin elementos). Si $n \geq 1$, se dice que a_0 es el primer elemento de la lista y a_{n-1}, el último.

Una propiedad importante de las listas es que sus elementos pueden ser linealmente ordenados de acuerdo a su posición. Se dice que a_i precede a a_{i+1} para $i = 0,1, \dots, n-2$ y que a_i sucede a a_{i-1} para $i = 1,2, \dots, n-1$. Se dice que el elemento a_i está en la posición i. Además de lo anterior, suele manejarse una posición adicional que sucede al último elemento de la lista l (posición n, correspondiente al elemento $n+1$), la cual varía a medida que la lista crece o se encoge, mientras que las demás posiciones mantienen una distancia fija con respecto al principio de la lista. Corresponde al largo de la lista y se asume por defecto.

4.2.2 EL TDA LISTA

Para poder crear un TDA a partir del concepto matemático de lista, es necesario definir un conjunto de operaciones para los elementos de tipo lista. Si bien no existe un único conjunto de operaciones que sea adecuado para todas las aplicaciones, existen algunas de uso frecuente que son representativas:

1. `INSERTAR(l: lista, posición: entero, elemento: tipoDato): lista.`
   Inserta el elemento dado en la posición señalada de la lista l. Esto significa que desplaza hacia la derecha todos los elementos l situados desde la posición dada en adelante. Así, una vez efectuada la operación, se tiene que:

   $l = a_0, a_1, \dots, a_{posición-1}, a_{posición}, \dots, a_{n-1}$

   Tras la operación insertar se obtiene:

   $l = a_0, a_1, \dots, a_{posición-1}, elemento, a_{posición}, \dots, a_{n-1}$

   Cabe destacar que, si no existe la posición dada en la lista l, el resultado es indefinido. Suele devolverse la lista sin cambios.

   Otro aspecto importante es que no existe una única forma de implementar esta operación. Existen múltiples variantes que suelen depender de las necesidades del problema: insertar siempre en un extremo de la lista, o bien agregar los nuevos elementos de modo que la lista esté siempre ordenada. En estas variantes no se requiere como parámetro la posición para el nuevo elemento.

2. `BUSCAR(l: lista, elemento: tipoDato): entero.`
   Entrega la posición del elemento dado en la lista. Si está más de una vez, entonces se entrega la posición de su primera aparición. Si no está contenido en la lista, entonces se retorna nulo o el fin de la lista (más adelante se emplea la misma alternativa.

3. `OBTENER(l: lista, posición: entero): tipoDato.`
   Retorna el elemento situado en la posición dada de la lista. El resultado es indefinido si la posición no es válida.

4. `BORRAR(l: lista, posición: entero): lista.`
   Elimina el elemento en la posición dada de la lista. Si se tiene que:

$l = a_0, a_1, \ldots, a_{posición-1}, a_{posición}, a_{posición+1}, \ldots, a_{n-1}$

Tras la operación borrar se obtiene:

$l = a_0, a_1, \ldots, a_{posición-1}, a_{posición+1}, \ldots, a_{n-1}$

El resultado es indefinido si la posición no es válida. Suele devolverse la lista sin cambios.

5. `ANTERIOR(l: lista, posición: entero): entero.`
   Entrega la posición del elemento que precede al de la posición dada. Si la posición es 0 o no es válida, anterior es indefinido.

6. `SIGUIENTE(l: lista, posición: entero): entero.`
   Entrega la posición del elemento que sigue al de la posición dada. Si es la última posición de la lista o no es válida, siguiente es indefinido.

7. `ANULAR(l: lista): lista.`
   Elimina todos los elementos de la lista.

8. `PRIMERO(l: lista): entero.`
   Devuelve la primera posición de la lista L. Si L está vacía, es indefinido.

9. `MOSTRAR(l: lista).`
   Muestra los elementos de la lista en orden de aparición.

4.3 IMPLEMENTACIÓN DEL TDA LISTA

Si bien la definición del TDA lista es bastante precisa, existen diversas formas de implementarlo en un programa, dependiendo de las herramientas que proporcione el lenguaje de programación y de los requerimientos del programa a implementar. Sin embargo, es fácil notar que todas las operaciones requieren, a lo más, recorrer la lista completa una única vez, con lo que las operaciones más ineficientes tienen complejidad $O(n)$.

Es importante señalar que, aunque por definición los algoritmos deben ser independientes de la implementación, en este caso, al estar creando un nuevo TDA a partir de los tipos de datos y restricciones propias de los lenguajes de programación, no se logra dicha independencia, pues la implementación de las operaciones del TDA varía de acuerdo a su estructura de datos asociada. No obstante, aquellos algoritmos que requieran del uso del TDA sí podrán ser escritos sin necesidad de especificar la implementación de este último. El ejemplo 4.1 ilustra cómo se logra esta independencia.

EJEMPLO 4.1:

Proponga un algoritmo que, dada una lista l, elimine todos los elementos duplicados.

ALGORITMO 4.1: Eliminación de elementos duplicados de una lista.

```
BORRAR_DUPLICADOS(l: lista): lista
      lista nueva

      Mientras l != NULO:
            elemento = OBTENER(l, 0)
            l = BORRAR(l, 0)
            posición = BUSCAR(nueva, elemento)

            Si posición == largo(nueva):
                  nueva = INSERTAR(nueva, largo(nueva))

      Devolver nueva
```

4.3.1 IMPLEMENTACIÓN MEDIANTE ARREGLOS

La primera variante es la implementación por medio de arreglos. Si bien tiene las ventajas de ser la más sencilla de implementar y de ser muy rápida para acceder a los elementos, tiene las limitantes de que es necesario conocer a priori la cantidad máxima de elementos a almacenar (para establecer el tamaño del arreglo) y de ser la menos eficiente para las operaciones de inserción y borrado.

El algoritmo 4.2 muestra el pseudocódigo para implementar el TDA lista por medio de arreglos. Cabe señalar que se crean las operaciones adicional, `CREAR_LISTA()`, que inicializa la estructura de datos para su operación, y `LARGO()`, que entrega la cantidad de elementos de la lista.

ALGORITMO 4.2: Implementación del TDA lista mediante arreglos.

```
tipo lista:
      arreglo[n]: tipoDato
      fin: entero

CREAR_LISTA():
    lista l
    l.fin = 0
    Devolver l

LARGO(l: lista): entero
      Devolver l.fin
```

ALGORITMO 4.2 (continuación): Implementación del TDA lista mediante arreglos.

```
INSERTAR(l: lista, posición: entero, elemento: tipoDato): lista
      Si posición < 0 ó posición > l.fin:
            Devolver l

      Para i desde l.fin – 1 hasta posición:
            l.arreglo[i + 1] = l.arreglo[i]

      l.arreglo[posición] = elemento
      l.fin = l.fin + 1
      Devolver l

BUSCAR(l: lista, elemento: tipoDato): entero
      Para i desde 0 hasta l.fin – 1:
            Si l.arreglo[i] == elemento:
                  Devolver i

      Devolver –1

OBTENER(l: lista, posición: entero): tipoDato
      Si posición < 0 ó posición >= l.fin:
            Devolver NULO

      Devolver l.arreglo[posición]

BORRAR(l: lista, posición: entero): lista
      Si posición < 0 ó posición >= l.fin:
            Devolver l

      Para i desde posición hasta l.fin – 1:
            l.arreglo[i] = l.arreglo[i + 1]

      l.fin = l.fin – 1
      Devolver l

ANTERIOR(l: lista, posición: entero): entero
      Si posición > l.fin – 1 ó posición <= 0:
            Devolver NULO

      Devolver posición – 1

SIGUIENTE(l: lista, posición: entero): entero
      Si posición >= l.fin – 1 ó posición < 0:
            Devolver NULO

      Devolver posición + 1

ANULAR(l: lista): lista
      l.fin = 0
      Devolver l
```

ALGORITMO 4.2 (continuación): Implementación del TDA lista mediante arreglos.

```
PRIMERO(l: lista): entero
      Si l.fin == 0:
            Devolver NULO

      Devolver 0

MOSTRAR(l: lista)
      Para i desde 0 hasta l.fin - 1:
            imprimir(l.arreglo[i])
```

4.3.2 IMPLEMENTACIÓN MEDIANTE PUNTEROS

Para esta representación, que suele recibir el nombre de lista enlazada, se requiere de una estructura que permita almacenar el dato correspondiente a un elemento de la lista y un puntero al siguiente elemento. Cada instancia de esta estructura se denomina nodo. Así, conceptualmente, se puede representar una lista enlazada como una secuencia de nodos, tal como se muestra en la figura 4.1. En este tipo de implementación el primer nodo suele llamarse cabeza de la lista.

Cabe destacar que el campo que apunta al siguiente elemento contiene la dirección de memoria donde este último comienza. En el caso del último nodo de la lista, dicho puntero debe ser nulo.

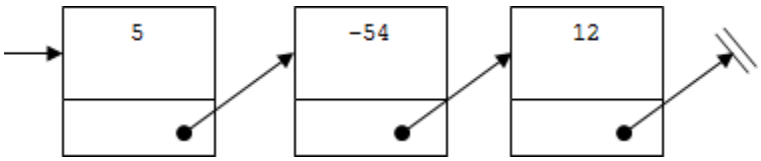


FIGURA 4.1: Representación de una lista enlazada con tres nodos.

Se debe pedir memoria para cada nodo de la lista, y dichos nodos no necesariamente estarán en posiciones contiguas de memoria. Así, se debe explicitar la posición de memoria a la que apunte cada puntero. Así, para agregar un nuevo nodo, se tiene el esquema representado en la figura 4.2. El puntero del nuevo nodo apunta al nodo que ocupaba su posición en la lista, y el nodo anterior apunta ahora al nuevo nodo en lugar de a su sucesor anterior (con flecha segmentada). Este alcance hace necesario liberar la memoria asociada a cada nodo de manera independiente.

Si bien la implementación de una lista enlazada es más compleja que la implementación con arreglos, es más eficiente para la inserción y eliminación de datos, aunque pierde eficiencia en cuanto al acceso a los elementos.

El algoritmo 4.3 muestra el pseudocódigo para implementar el TDA lista como lista enlazada. Cabe señalar que se agregan tres operaciones adicionales: `CREAR_LISTA()`, que inicializa la estructura de datos para su operación, y `CREAR_NODO()`, que realiza la misma tarea para un nodo en particular, y `LARGO()`, que entrega la cantidad de elementos de la lista. Se puede notar que la definición de las operaciones del TDA lista, sus parámetros y sus tipos de retorno son exactamente iguales que en la implementación

mediante arreglos presentada en el algoritmo 4.1. Otra consideración es que se hace una redefinición del tipo `nodo` como tipo `lista`, a fin de mantener la nomenclatura. No obstante, conceptualmente la variable `l` no es más que un puntero al primer nodo de la lista.

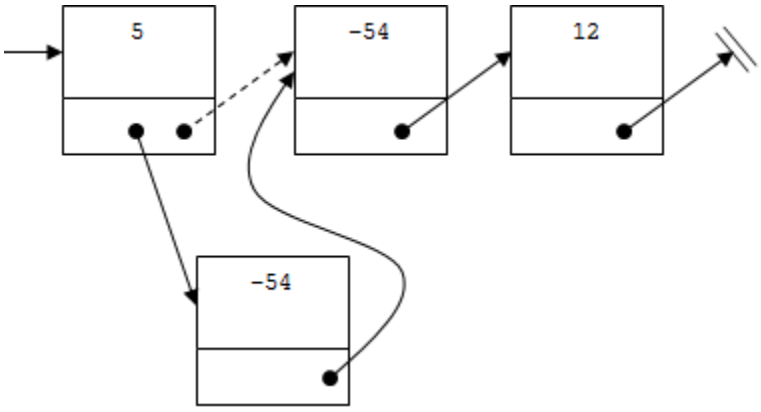


FIGURA 4.2: Insertar un nodo en una lista enlazada.

ALGORITMO 4.3: Implementación del TDA lista mediante punteros.

```
tipo nodo:
      dato: tipoDato
      siguiente: nodo

tipo lista = tipo nodo

CREAR_LISTA():
    lista l
    l = NULO
    Devolver l

CREAR_NODO(elemento: tipoDato): nodo
      nodo nuevo
      nuevo.dato = elemento
      nuevo.siguiente = NULO
      Devolver nuevo

LARGO(l: lista): entero
      i = 0
      nodo índice = l

      Mientras índice != NULO:
            índice = índice.siguiente
            i = i + 1

    Devolver i
```

ALGORITMO 4.3 (continuación): Implementación del TDA lista mediante punteros.

```
INSERTAR(l: lista, posición: entero, elemento: tipoDato): lista
      Si posición < 0:
            Devolver l

      Si posición == 0:
            nodo nuevo = CREAR_NODO(elemento)
            nuevo.siguiente = l
            Devolver nuevo

      i = 0
      nodo índice = l

      Mientras índice != NULO y i < posición – 1:
            índice = índice.siguiente
            i = i + 1

      Si índice != NULO:
            nodo nuevo = CREAR_NODO(elemento)
            nuevo.siguiente = índice.siguiente;
            índice.siguiente = nuevo

      Devolver l

OBTENER(l: lista, posición: entero): tipoDato
      Si posición < 0:
            Devolver NULO

      i = 0
      nodo índice = l

      Mientras índice != NULO y i < posición:
            Si índice.dato == elemento:
                  Devolver i

            índice = índice.siguiente
            i = i + 1

      Si índice == NULO:
            Devolver NULO

      Devolver índice.dato
```

ALGORITMO 4.3 (continuación): Implementación del TDA lista mediante punteros.

```
BORRAR(l: lista, posición: entero): lista
      Si posición < 0 ó l == NULO:
            Devolver l

      Si posición == 0:
            Nodo auxiliar = l.siguiente
            eliminar(l)
            Devolver auxiliar

      i = 0
      nodo índice = l

      Mientras índice != NULO y i < posición - 1:
            índice = índice.siguiente
            i = i + 1

      Si índice == NULO:
            Devolver l

      Nodo auxiliar = índice.siguiente

      Si auxiliar != NULO:
            índice.siguiente = auxiliar.siguiente
            eliminar(auxiliar)

      Devolver l

BUSCAR(l: lista, elemento: tipoDato): entero
      nodo índice = l
      i = 0

      Mientras índice != NULO:
            Si índice.dato == elemento:
                  Devolver i

            índice = índice.siguiente
            i = i + 1

      Devolver -1
```

ALGORITMO 4.3 (continuación): Implementación del TDA lista mediante punteros.

```
ANTERIOR(l: lista, posición: entero): entero
      Si posición <= 0 ó l == NULO:
            Devolver NULO

      nodo índice = l
      i = 0

      Mientras índice != NULO y i < posición - 1:
            índice = índice.siguiente
            i = i + 1

      Si índice == NULO:
            Devolver NULO

      Devolver posición - 1

SIGUIENTE(l: lista, posición: entero): entero
      Si posición < 0 ó l == NULO:
            Devolver NULO

      nodo índice = l
      i = 0

      Mientras índice != NULO y i < posición - 1:
            índice = índice.siguiente
            i = i + 1

      Si índice == NULO ó índice.siguiente == NULO:
            Devolver NULO

      Devolver posición + 1

ANULAR(l: lista): lista
      Mientras l != NULO:
            l = BORRAR(l, 0)

      Devolver l
```

ALGORITMO 4.3 (continuación): Implementación del TDA lista mediante punteros.

```
PRIMERO(l: lista): entero
      Si l == NULO:
            Devolver NULO

      Devolver 0

MOSTRAR(l: lista)
      nodo índice = l

      Mientras índice != NULO:
            imprimir(índice.dato)
            índice = índice.siguiente
```

En ocasiones se hace uso de una variante de la lista enlazada denominada lista doblemente enlazada, que considera los nodos con dos punteros: uno al elemento anterior y otro al elemento siguiente. Esta modificación hace que la operación con la lista sea más eficiente. No obstante, hace que la lista sea más costosa en cuanto a espacio de almacenamiento. Desde luego, la incorporación de este nuevo campo en la estructura de datos hace necesario redefinir algunas operaciones para tener en consideración que el recorrido puede realizarse también en sentido inverso y para tratar adecuadamente el nuevo puntero.

4.3.3 IMPLEMENTACIÓN BASADA EN CURSORES

Muchos lenguajes de programación no cuentan con punteros. En ellos, es posible simular el uso de éstos mediante cursores, es decir, por medio de valores enteros que indiquen la posición dentro de un arreglo. Más claramente, para almacenar la información de un elemento de la lista se crea una estructura (o registro) que contenga un campo para el valor a almacenar y otro para un valor entero que indique la posición del siguiente elemento dentro del arreglo. Adicionalmente, suele ser importante mantener almacenada en una variable la posición de la cabeza de la lista. La figura 4.3 muestra la representación gráfica de una lista de enteros cuya cabeza está en la posición 4, representada por medio de cursores, donde el cursor con valor -1 corresponde al valor nulo.

(0)	(1)	(2)	(3)	(4)	(5)
548	267	4087	23	-7825	452
5	3	-1	0	1	2

FIGURA 4.3: Lista de enteros implementada por cursores.

4.4 VARIANTES DEL TDA LISTA

En la sección anterior se definen las operaciones generales para el TDA lista. No obstante, existen algunos TDA que corresponden a variantes de la definición dada anteriormente:

4.4.1 LISTAS CIRCULARES

Una lista circular es una lista lineal en la que el primer elemento es precedido por el último y el último elemento es seguido a su vez por el primero. Esto permite evitar excepciones en las operaciones sobre listas, pues no existen casos especiales: cada elemento siempre tiene un elemento anterior y un elemento siguiente. Esta variante resulta adecuada cuando se quieren realizar tareas por turnos.

Algunos usos de listas circulares son:
- Llevar un registro de quién es el turno en juegos multijugador.
- Método Round Robin de asignación de procesos (asigna a cada proceso una cantidad de tiempo equitativa y ordenada).

Para trabajar con listas circulares solo se requiere modificar las operaciones `ANTERIOR(l, posición)` y `SIGUIENTE(l, posición)`, como muestran los algoritmos 4.4 (con arreglos) y 4.5 (con listas enlazadas).

ALGORITMO 4.4: Listas circulares implementadas mediante arreglos.

```
ANTERIOR(l: lista, posición: entero): entero
      Si posición > l.fin – 1 ó posición < 0:
            Devolver NULO

      Si posición == 0:
            Devolver devolver l.fin – 1

      Devolver posición – 1

SIGUIENTE(l: lista, posición: entero): entero
      Si posición >= l.fin ó posición < 0:
            Devolver NULO

      Si posición == l.fin:
            Devolver 0

      Devolver posición + 1
```

ALGORITMO 4.5: Listas circulares implementadas mediante punteros.

```
ANTERIOR(l: lista, posición: entero): entero
      Si posición < 0 ó l == NULO:
            Devolver NULO

      Si posición == 0:
            Devolver LARGO(l) - 1

      nodo índice = l
      i = 0

      Mientras índice != NULO y i < posición - 1:
            índice = índice.siguiente
            i = i + 1

      Si índice == NULO:
            Devolver NULO

      Devolver posición - 1

SIGUIENTE(l: lista, posición: entero): entero
      Si posición < 0 ó l == NULO:
            Devolver NULO

      nodo índice = l
      i = 0

      Mientras índice != NULO y i < posición - 1:
            índice = índice.siguiente
            i = i + 1

      Si índice == NULO:
            Devolver NULO

      Si índice.siguiente == NULO:
            Devolver 0

      Devolver posición + 1
```

4.4.2 PILAS

Una pila (*stack*) es una variante de las listas donde todas las inserciones y eliminaciones de elementos ocurren en un extremo de la lista llamado tope. Suelen ser denominadas también como listas LIFO (del inglés *last in, first out* o último en entrar, primero en salir). Una noción intuitiva para el modelo de pila es la de una montaña de platos por lavar, donde el plato de más arriba fue el último en ser agregado, pero es el primero en ser lavado.

Entre otras aplicaciones, las pilas sirven para:

- Administrar el flujo de ejecución de programas. Cada nueva llamada a una función o procedimiento se almacena en una pila de control, y siempre es la función que está al tope de la pila la que se encuentra

en ejecución. Al terminar, se elimina de la pila y continúa la ejecución de la función que hizo la llamada.
- Al compilar un programa, se usa una pila para comprobar, por ejemplo, si los paréntesis están balanceados.
- Llevar un registro de tareas para deshacer (¡como en Word o Excel!).

Las implementaciones del TDA pila suelen incorporar las siguientes operaciones, asumiendo que la estructura de datos de la pila es idéntica a la de una lista:

1. `ANULAR(p: pila): pila.`
   Elimina todos los elementos de la pila. Es idéntica a la operación de la lista común.

2. `TOPE(p: pila): tipoDato.`
   Entrega el elemento al tope de la pila.

3. `PUSH(p: pila, elemento: tipoDato): pila.`
   Inserta el elemento al tope de la pila.

4. `POP(p: pila): pila.`
   Elimina el elemento al tope de la pila. En muchas ocasiones resulta conveniente que esta función retorne el elemento eliminado.

5. `VACÍA(p: pila): booleano.`
   Retorna verdadero si la pila está vacía. En otro caso, retorna falso.

El algoritmo 4.6 muestra la implementación del TDA pila por medio de arreglos.

ALGORITMO 4.6: Implementación de una pila usando arreglos.

```
tipo pila:
      arreglo[n]: tipoDato
      fin: entero

CREAR_PILA(): pila
      pila p
      p.fin = 0
      Devolver p

ANULAR(p: pila): pila
      p.fin = 0
      Devolver p

TOPE(p: pila): tipoDato
      Si p.fin != 0:
        Devolver p.arreglo[p.fin - 1]

    Devolver NULO;
```

ALGORITMO 4.6 (continuación): Implementación de una pila usando arreglos.

```
PUSH(p: pila, int elemento): pila
    Si p.fin == n:
        Devolver p

    p.arreglo[p.fin] = elemento
    p.fin = p.fin + 1
    Devolver p

POP(p: pila): pila
    Si p.fin == 0:
        Devolver p

    p.fin = p.fin -1
    Devolver p

VACÍA(p: pila): booleano
    Si p.fin == 0:
        Devolver verdadero

    Devolver falso
```

Es interesante notar que, para implementar una pila mediante arreglos, resulta más eficiente pensar que el tope está al final del arreglo, pues así es innecesario realizar movimientos de datos al momento de usar las operaciones `PUSH(p, elemento)` y `POP(p)`, con lo que el tiempo de ejecución para ambas es $O(1)$.

El algoritmo 4.7 muestra la implementación del TDA pila usando punteros. En contraste con la implementación anterior, en este caso es más eficiente considerar que el tope de la pila está al comienzo. Así, es innecesario recorrer la pila completa para hacer uso de las operaciones `PUSH(p, elemento)` y `POP(p)`, con lo cual el tiempo de ejecución para cada operación sigue siendo $O(1)$.

ALGORITMO 4.7: Implementación de una pila usando punteros.

```
tipo nodo:
      dato: tipoDato
      siguiente: nodo

tipo pila = tipo nodo
      arreglo[n]: tipoDato
      fin: entero

CREAR_NODO(dato: tipoDato): nodo
      nodo nuevo
      nuevo.dato = dato
      nuevo.siguiente = NULO
      Devolver nuevo

CREAR_PILA(): pila
      pila p
      p = NULO
      Devolver p
```

ALGORITMO 4.7 (continuación): Implementación de una pila usando punteros.

```
ANULAR(p: pila): pila
      Mientras p != NULO:
            p = POP(p)

      Devolver p

TOPE(p: pila): tipoDato
      Si p != NULO:
            Devolver p.dato

      Devolver NULO;

PUSH(p: pila, int elemento): pila
      nodo nuevo = CREAR_NODO(elemento)
      nuevo.siguiente = p
      Devolver nuevo

POP(p: pila): pila
      Si p != NULO:
            nodo auxiliar = p.siguiente
            eliminar(p)
            Devolver auxiliar

      Devolver p

VACÍA(p: pila): booleano
      Si p == NULO:
            Devolver verdadero

      Devolver falso
```

4.4.3 COLAS

Una cola (*queue*) es un tipo particular de lista donde los elementos se insertan en un extremo (el final) y se eliminan en el otro (el inicio). Se conocen también como listas FIFO (*first in, first out*, es decir, primero en entrar, primero en salir). Las operaciones para colas son análogas a las de pilas, excepto por la inserción. También cambia la nomenclatura utilizada. Como las operaciones `QUEUE(q, elemento)` y `DEQUEUE(q)` operan sobre extremos diferentes, una de ellas tiene complejidad constante y la otra, lineal.

1. `ANULAR(q: cola): cola`.
   Elimina todos los elementos de la cola.

2. `INICIO(q: cola): tipoDato`.
   Entrega el primer elemento de la cola.

3. `QUEUE(q: cola, elemento: tipoDato): cola`.
   También llamada encolar, inserta el elemento al final de la cola.

4. `DEQUEUE(q: cola): cola`.
   También llamada desencolar, elimina el primer elemento de la cola. En muchas ocasiones resulta conveniente que esta función retorne el elemento eliminado.

5. `VACÍA(q: cola): booleano`.
   Retorna verdadero si la cola está vacía. En otro caso, retorna falso.

Las colas se usan, entre otras aplicaciones, para:
- Manejo de impresiones.
- Procesamiento secuencial de tareas.
- Manejo de paquetes de red.
- Cola de eventos (movimiento del mouse, presión de una tecla, clic con algún botón del mouse, etc.).

El algoritmo 4.8 muestra la implementación del TDA cola por medio de arreglos. En este caso, la operación `QUEUE(q, elemento)` tiene complejidad $O(1)$, mientras que `DEQUEUE(q)` tiene complejidad $O(n)$.

Análogamente, el algoritmo 4.9 implementa una cola por haciendo uso de punteros. En este caso, la operación `QUEUE(q, elemento)` tiene complejidad $O(n)$, mientras que `DEQUEUE(q)` tiene complejidad $O(1)$.

ALGORITMO 4.8: Implementación de una cola usando arreglos.

```
tipo cola:
      arreglo[n]: tipoDato
      fin: entero

CREAR_COLA(): cola
      cola q
      q.fin = 0
      Devolver q

ANULAR(q: cola): cola
      q.fin = 0
      Devolver q

INICIO(q: cola): tipoDato
      Si q.fin != 0:
            Devolver q.arreglo[0]

      Devolver NULO

QUEUE(q: cola, elemento: tipoDato): cola
      Si q.fin == n:
            Devolver q

      q.arreglo[q.fin] = elemento
      q.fin++
      Devolver q
```

ALGORITMO 4.8 (continuación): Implementación de una cola usando arreglos.

```
DEQUEUE(q: cola): cola
      Si q.fin == 0:
            Devolver q

      Para i desde 0 hasta q.fin - 2:
            q.arreglo[i] = q.arreglo[i + 1]

      q.fin = q.fin - 1
      Devolver q

VACÍA(q: cola): booleano
      Si q.fin == 0:
            Devolver verdadero

      Devolver falso
```

ALGORITMO 4.9: Implementación del TDA cola por medio de punteros.

```
tipo nodo:
      dato: tipoDato
      siguiente: nodo

tipo cola = tipo nodo

CREAR_NODO(dato: tipoDato): nodo
      nodo nuevo
      nuevo.dato = dato
      nuevo.siguiente = NULO
      Devolver nuevo

CREAR_COLA(): cola
      cola q
      q = NULO
      Devolver q

ANULAR(q: cola): cola
      Mientras q:
            q = DEQUEUE(q)

      Devolver q

INICIO(q: cola): tipoDato
      Si q != NULO:
            Devolver q.dato

      Devolver NULO
```

ALGORITMO 4.9 (continuación): Implementación del TDA cola por medio de punteros.

```
QUEUE(q: cola, elemento: tipoDato): cola
      nodo nuevo = CREAR_NODO(elemento)
      nodo indice = q

      Si q == NULO:
            Devolver nuevo

      Mientras indice.siguiente != NULO:
            indice = indice.siguiente

      indice.siguiente = nuevo
      Devolver q

DEQUEUE(q: cola): cola
      Si q != NULO:
            nodo auxiliar = q.siguiente
            eliminar(q)
            Devolver auxiliar

      Devolver q

VACÍA(q: cola): booleano
      Si q == NULO:
            Devolver verdadero

      Devolver falso
```

5 GRAFOS

Para las ciencias de la computación y la matemática, un grafo es una representación gráfica de diversos puntos que se conocen como nodos o vértices, los cuales se encuentran unidos a través de líneas o flechas que reciben el nombre de aristas. Al analizar los grafos, los expertos logran conocer cómo se desarrollan las relaciones recíprocas entre aquellas unidades que mantienen algún tipo de interacción.

5.1 TIPOS DE GRAFOS

Un grafo no dirigido $G(V, A)$ está conformado por un conjunto finito no vacío de vértices V y de un conjunto de aristas A. Se denota $|V| = n$ y $|A| = m$ al número de vértices y de aristas existentes en el grafo, respectivamente. Si (u, v) es una arista no dirigida, entonces $(u, v) = (v, u)$. Se considera a G sin lazos, es decir, $(u, u) \notin A$ y sin aristas múltiples, por lo tanto (u, v) aparece una vez en A. En A hay pares no ordenados de elementos distintos de V. Los vértices se representan gráficamente mediante círculos, mientras que las aristas se simbolizan mediante líneas que conectan dos vértices. En la figura 5.1 se muestra la arista $a = (u, v)$, donde u y v son sus extremos. Se dice que u es adyacente a v y que v es adyacente a u.

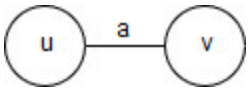


FIGURA 5.1: Grafo no dirigido con dos vértices y una arista.

Los grafos no dirigidos se emplean para modelar relaciones simétricas entre objetos, donde éstos son representados por los vértices del grafo. Dos objetos están conectados por una arista si están relacionados entre sí (se puede asociar algún valor (peso) a la arista para proporcionar información adicional).

EJEMPLO 5.1:

El grafo de la figura 5.2 muestra algunas ciudades de Chile, donde a cada arista (u, v) se le asocia la distancia, en kilómetros que separa a ambas ciudades.

Un grafo dirigido (grafo direccionado, grafo orientado o digrafo) $\vec{G}$ consiste en un conjunto de vértices V y un conjunto de arcos $\vec{A}$. Los vértices, representados gráficamente por círculos, se denominan también nodos o puntos; los arcos pueden llamarse arcos dirigidos o aristas y se representan mediante flechas que conectan pares de círculos, donde u es la cola y v es la cabeza. La arista (u, v) se expresa a menudo como $u \rightarrow v$ o $\overrightarrow{uv}$. Al igual que en los grafos no dirigidos, las aristas de un grafo dirigido pueden tener asociado un peso o valor, y se denota $|V| = n$ y $\left|\vec{A}\right| = m$ a las cantidades de vértices y de aristas, respectivamente.

La figura 5.3 muestra la arista $a = (u, v)$ para un grafo dirigido. Es importante destacar que, en este tipo de grafos, $(u, v) \neq (v, u)$. Obsérvese que la punta de la flecha está en el vértice llamado cabeza y el extremo opuesto, en el vértice llamado cola. Se dice que el arco $a = (u, v)$ va de u a v y que v es adyacente a u. Se dice que a es divergente de u y convergente a v.

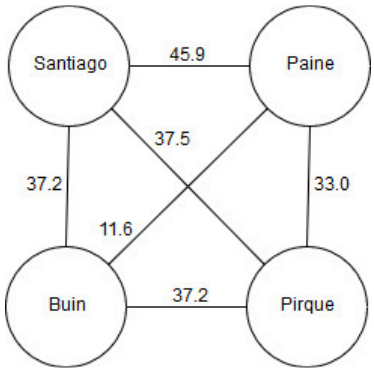


FIGURA 5.2: Modelo de la distancia entre ciudades usando un grafo.

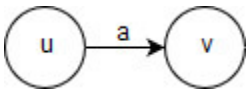


FIGURA 5.3: Grafo direccionado.

5.2 DEFINICIONES ELEMENTALES

5.2.1 GRADO Y ADYACENCIA

En un grafo no dirigido, los vértices u y v son adyacentes si (u, v) es una arista. Se dice que la arista (u, v) es incidente sobre los vértices u y v. A partir de esta idea, se define el conjunto de adyacencia de un vértice v, denotado por $Ady(v)$, como el conjunto de todos los vértices adyacentes a él, es decir:

$$Ady(v) = \{u \in V : (u, v) \in A\}$$

De manera similar, cuando se tienen dos aristas diferentes, $a = (u, v)$ y $b = (u, w)$, con $v \neq w$, se dice que son adyacentes pues ambas inciden sobre el vértice u. Así, el conjunto de adyacencia de una arista (u, v) v, denotado por $Ady\big((u, v)\big)$ es el conjunto de todas las aristas que comparten un extremo con (u, v).

Se denomina grado de un vértice $v \in V$, denotado por $gr(v)$ a la cardinalidad de su conjunto de adyacencia, es decir:

$$gr(v) = |Ady(v)|$$

En particular, un vértice aislado es aquel cuyo grado es 0. Una propiedad importante asociada al concepto de grado es que la sumatoria de los grados de todos los vértices de un grafo es igual a dos veces la cantidad de aristas, es decir:

$$\sum_{v \in V} gr(v) = 2m$$

EJEMPLO 5.2:

Determine los conjuntos de adyacencia para cada vértice y cada arista del grafo no dirigido de la figura 5.4. Obtenga además el grado de cada vértice del grafo.

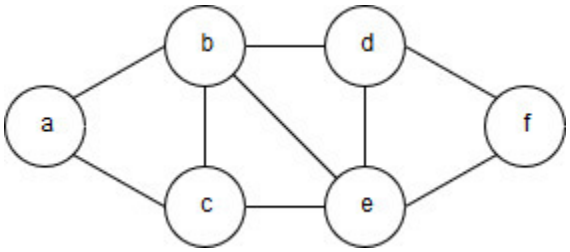


FIGURA 5.4: Grafo no dirigido para el ejemplo 5.2.

$V = \{a, b, c, d, e, f\}$ $\quad n = 6$
$A = \{(a,b), (a,c), (b,c), (b,d), (b,e), (c,e), (d,e), (d,f), (e,f)\}$ $\quad m = 9$

$Ady(a) = \{b, c\}$ $\quad gr(a) = 2$
$Ady(b) = \{a, c, d, e\}$ $\quad gr(b) = 4$
$Ady(c) = \{a, b, e\}$ $\quad gr(c) = 3$
$Ady(d) = \{b, e, f\}$ $\quad gr(d) = 3$
$Ady(e) = \{b, c, d, f\}$ $\quad gr(e) = 4$
$Ady(f) = \{d, e\}$ $\quad gr(f) = 2$

$Ady\big((a,b)\big) = \{(a,c), (b,c), (b,d), (b,e)\}$
$Ady\big((a,c)\big) = \{(a,b), (b,c), (c,e)\}$
$Ady\big((b,c)\big) = \{(a,b), (a,c), (b,d), (b,e), (c,e)\}$
$Ady\big((b,d)\big) = \{(a,b), (b,c), (b,e), (d,e), (d,f)\}$
$Ady\big((b,e)\big) = \{(a,b), (b,c), (b,d), (c,e), (d,e), (e,f)\}$
$Ady\big((c,e)\big) = \{(a,c), (b,c), (b,e), (d,e), (e,f)\}$
$Ady\big((d,e)\big) = \{(b,d), (b,e), (c,e), (d,f), (e,f)\}$
$Ady\big((d,f)\big) = \{(b,d), (d,e), (e,f)\}$
$Ady\big((e,f)\big) = \{(b,e), (c,e), (d,e), (d,f)\}$

Es necesario modificar levemente los conceptos previos para poder aplicarlos a grafos dirigidos. Para un vértice $v \in V$, el conjunto de adyacencia de v se separa en:

$Ady(v) = T^+(v) \cup T^-(v)$

Donde:

- $T^+(v) = \{w \in V : (v,w) \in \vec{A}\}$ es el conjunto de sucesores de v.
- $T^-(v) = \{u \in V : (u,v) \in \vec{A}\}$ es el conjunto de antecesores de v.

Similarmente, se divide el grado de un nodo en:
- Grado de salida: $gr^+(v) = |T^+(v)|$.
- Grado de entrada: $gr^-(v) = |T^-(v)|$.

Así, se tiene la siguiente propiedad: $\forall v \in gr(v) = gr^+(v) + gr^-(v)$.

Se denomina fuente a un vértice f tal que $gr^-(f) = 0$.

Se denomina sumidero a un vértice s tal que $gr^+(s) = 0$.

EJEMPLO 5.3:

Determine los conjuntos de adyacencia para cada vértice y cada arista del grafo dirigido de la figura 5.5. Obtenga además el grado de cada vértice del grafo.

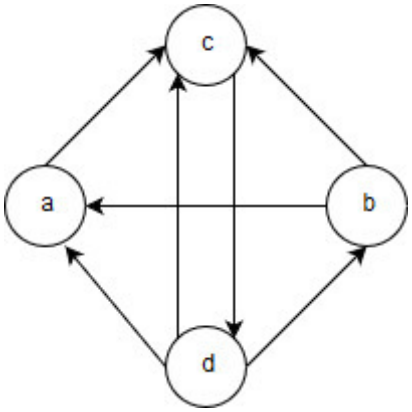


FIGURA 5.5: Grafo dirigido para el ejemplo 5.3.

$V = \{a, b, c, d\}$ $\qquad n = 4$
$A = \{(a,c), (b,a), (b,c), (c,d), (d,a), (d,b), (d,c)\}$ $\qquad m = 7$

$Ady(a) = T^+(a) \cup T^-(a) = \{c\} \cup \{b, d\}$ $\qquad gr(a) = gr^+(a) + gr^-(a) = 1 + 2 = 3$
$Ady(b) = T^+(b) \cup T^-(b) = \{a, c\} \cup \{d\}$ $\qquad gr(b) = gr^+(b) + gr^-(b) = 2 + 1 = 3$
$Ady(c) = T^+(c) \cup T^-(c) = \{d\} \cup \{a, b, d\}$ $\qquad gr(c) = gr^+(c) + gr^-(c) = 1 + 3 = 4$
$Ady(d) = T^+(d) \cup T^-(d) = \{a, b, c\} \cup \{c\}$ $\qquad gr(d) = gr^+(d) + gr^-(d) = 3 + 1 = 4$

$Ady\big((a,c)\big) = \{(b,a), (b,c), (c,d), (d,a), (d,c)\}$
$Ady\big((b,a)\big) = \{(a,c), (b,c), (d,a), (d,b)\}$
$Ady\big((b,c)\big) = \{(a,c), (b,a), (c,d), (d,b), (d,c)\}$
$Ady\big((c,d)\big) = \{(a,c), (b,c), (d,a), (d,b), (d,c)\}$
$Ady\big((d,a)\big) = \{(a,c), (b,a), (c,d), (d,b), (d,c)\}$
$Ady\big((d,b)\big) = \{(b,a), (b,c), (c,d), (d,a), (d,c)\}$
$Ady\big((d,c)\big) = \{(a,c), (b,c), (c,d), (d,a), (d,b)\}$

5.2.2 GRAFO COMPLEMENTO

Dado un grafo $G(V, A)$, se define como grafo complemento de G, denotado por $\bar{G}(V, \bar{A})$, al grafo tal que $\bar{A} = \{(u, v): u, v \in V, (u, v) \notin A\}$. Es decir, $\bar{A}$ solo contiene todas aquellas aristas que no están presentes en el grafo G. Las figuras 5.6 y 5.7 ilustra este concepto para grafos no dirigidos y dirigidos, respectivamente.

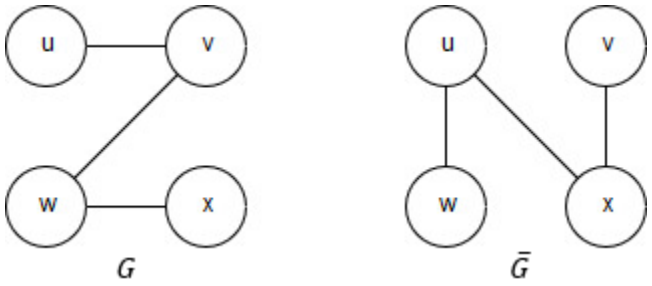


FIGURA 5.6: Un grafo no dirigido y su complemento.

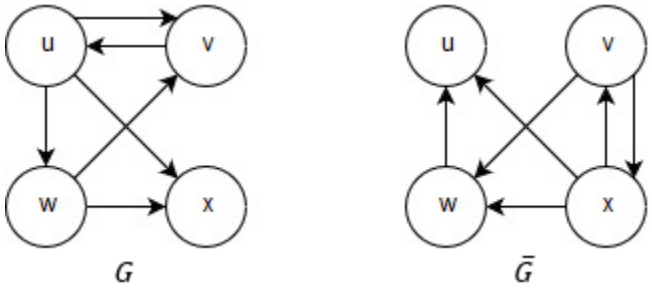


FIGURA 5.7: Un grafo dirigido y su complemento.

5.2.3 GRAFO COMPLETO

Se denomina grafo completo a un grafo en el cual cada vértice está relacionado con todos los demás, es decir, a la unión de un grafo y su complemento. En el caso de grafos no dirigidos, un grafo es completo si $(u, v) \in A \forall u \neq v$. El grafo completo de n vértices se denota por K_n. La figura 5.8 muestra grafos no dirigidos completos para $n = 1$ hasta $n = 4$. Una propiedad importante es que la cantidad de aristas de un grafo no dirigido completo de n vértices siempre es:

$$|A| = \frac{n(n-1)}{2}$$

La noción de grafo completo para un grafo dirigido es análoga a la anterior, pero tienen que aparecer todas las aristas en ambos sentidos, es decir, $(u, v) \in A \; y \; (v, u) \in A \forall u \neq v$. En consecuencia, la cantidad de aristas de un grafo dirigido completo está dada por:

$$\left|\vec{A}\right| = n(n-1)$$

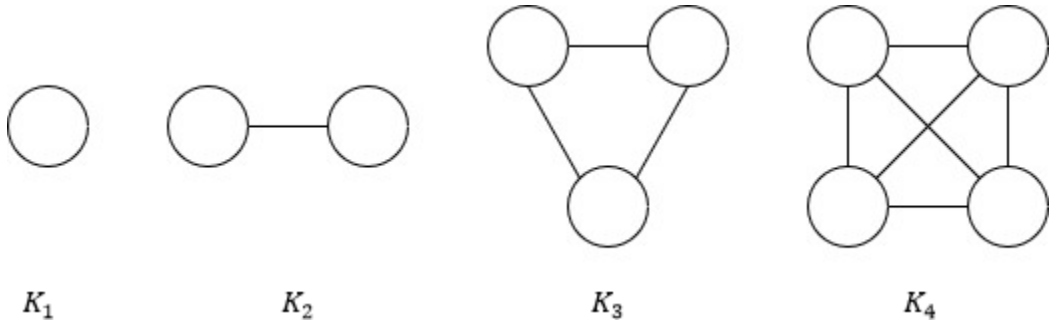


FIGURA 5.8: Grafos no dirigidos completos.

La figura 5.9 muestra grafos dirigidos completos para $n = 1$ hasta $n = 3$.

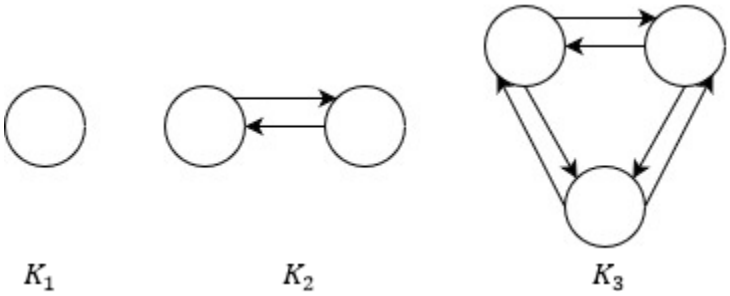


FIGURA 5.9: Grafos dirigidos completos.

5.2.4 GRAFO REGULAR

Un grafo es regular de grado r si todos sus vértices tienen grado r. Como ejemplos, se puede citar a los grafos completos: K_2 es regular de grado 1, K_3 es regular de grado 2, K_4 es regular de grado 3, etc. El grafo 5.10 es regular de grado 2.

5.2.5 SUBGRAFO

Sea $G(V, A)$ un grafo con conjunto de vértices V y conjunto de aristas A. Un subgrafo de G (figura 5.11) es un grafo $H(V', A')$ donde:

1. V' es un subconjunto de V.
2. A' consta de aristas (u, v) de A tales que u y v están en V'.

Dado $T \subset V$, el subgrafo $H(T, A_T)$ donde A_T contiene todas las aristas de G con los dos extremos en T, es un subgrafo inducido por T. Para el grafo G de la figura 5.11, considerando $T = \{a, b, d, e\}$, el subgrafo inducido corresponde al que se muestra en la figura 5.12.

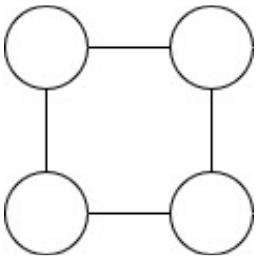


FIGURA 5.10: Un grafo regular de grado 2.

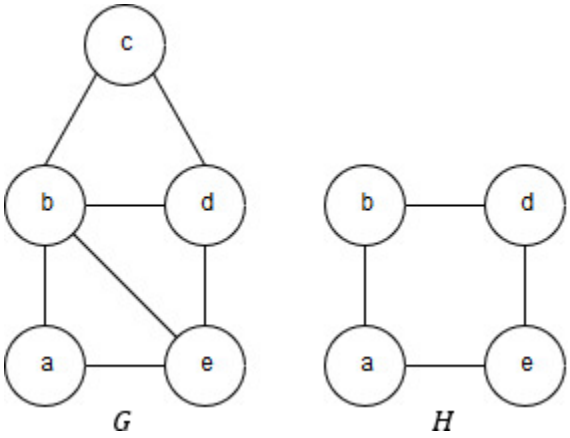


FIGURA 5.11: Grafo G y subgrafo H.

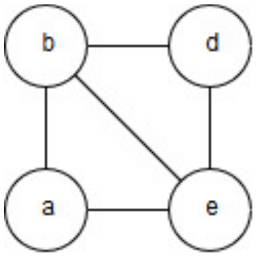


FIGURA 5.12: Subgrafo inducido por T.

5.2.6 CAMINO, CUERDA Y CICLO

Un camino es una secuencia de vértices $v_1, v_2, \ldots, v_n$ tal que (v_i, v_{i+1}) es una arista para $1 \leq i \leq n$. Un camino es simple si todos sus vértices son distintos.
Se define como longitud del camino al número de aristas que lo conforman.

Una cuerda es una arista no perteneciente al camino que une dos vértices de él. En la figura 5.13 (a) se puede ver un camino con cuerda. Además, si un camino no tiene cuerdas se denomina camino inducido, como el de la figura 5.13 (b).

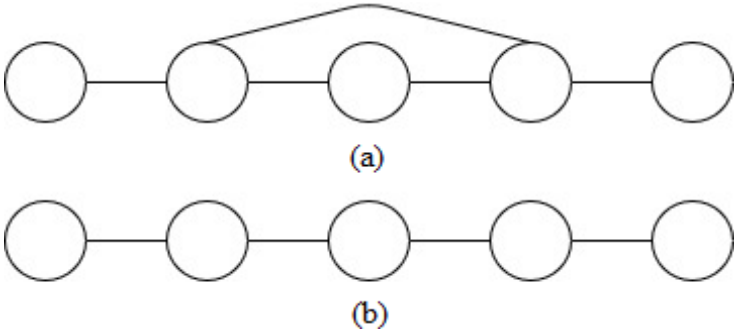


FIGURA 5.13: (a) Camino con cuerda y (b) camino inducido.

Un ciclo de un grafo no dirigido es un camino de longitud mayor o igual a tres, que conecta un vértice consigo mismo. No se consideran ciclos los caminos de la forma u (camino de longitud 0 o lazo), u, u (camino de longitud 1) o u, v, u (camino de longitud 2). Si un ciclo no tiene cuerdas, se le denomina ciclo inducido. En el caso de grafos dirigidos, el ciclo más pequeño posible tiene longitud 2 y es de la forma u, v, u (pues usa dos aristas diferentes).

Un grafo es cíclico si contiene por lo menos un ciclo. En caso contrario, se dice que es acíclico.

EJEMPLO 5.4:

Considere el grafo G de la figura 5.11, cuyo conjunto de vértices es $V = \{a, b, c, d, e\}$ y sus aristas son $A = \{(a, b), (a, e), (b, c), (b, d), (b, e), (c, d), (d, e)\}$. En él se puede distinguir el camino a, b, c, d de longitud 3, donde b, d es una cuerda que une dos componentes del camino. También se puede observar el ciclo a, b, d, e, a.

Se llama ciclo hamiltoniano a un camino $v_1, v_2, \dots, v_n$ de un grafo G que contiene todos los vértices del grafo exactamente una vez. El ciclo $v_1, v_2, \dots, v_n, v_1$ se denomina ciclo hamiltoniano. Se dice que un grafo es hamiltoniano si posee a lo menos un ciclo hamiltoniano. La figura 5.14 (a) es un ejemplo, pues el ciclo $1, 2, 3, 4, 5, 6, 7, 8, 1$ es hamiltoniano.

Un camino euleriano es un camino $a_1, a_2, \dots, a_n$ que pasa por todas las aristas del grafo exactamente una vez. Se llama ciclo euleriano al ciclo $a_1, a_2, \dots, a_n, a_1$. Un grafo es euleriano si posee a lo menos un ciclo euleriano, como el grafo de la figura 5.14 (b). En ella, $10, 1, 8, 5, 2, 7, 6, 3, 4, 9, 10$ es un ciclo euleriano.

En el caso de grafos dirigidos, se define como cadena a una secuencia de vértices tal que cada arista tenga un vértice en común con la arista antecesora (excepto la primera) y uno en común con la sucesora (excepto la última):

$v_1, v_2, \dots, v_k$ tal que $(v_i, v_{i+1}) \in \vec{A}$ o $(v_{i+1}, v_i) \in \vec{A}$, donde $1 \leq i \leq k$.

Se llama camino a una cadena en la cual todas las aristas poseen la misma dirección:

$v_1, v_2, \dots, v_k$ tal que $(v_i, v_{i+1}) \in \vec{A}$ o $(v_{i+1}, v_i) \in \vec{A}$, donde $1 \leq i \leq k$.

En el grafo de la figura 5.5, una cadena es a, b, c, d, mientras que a, c, d es un camino.

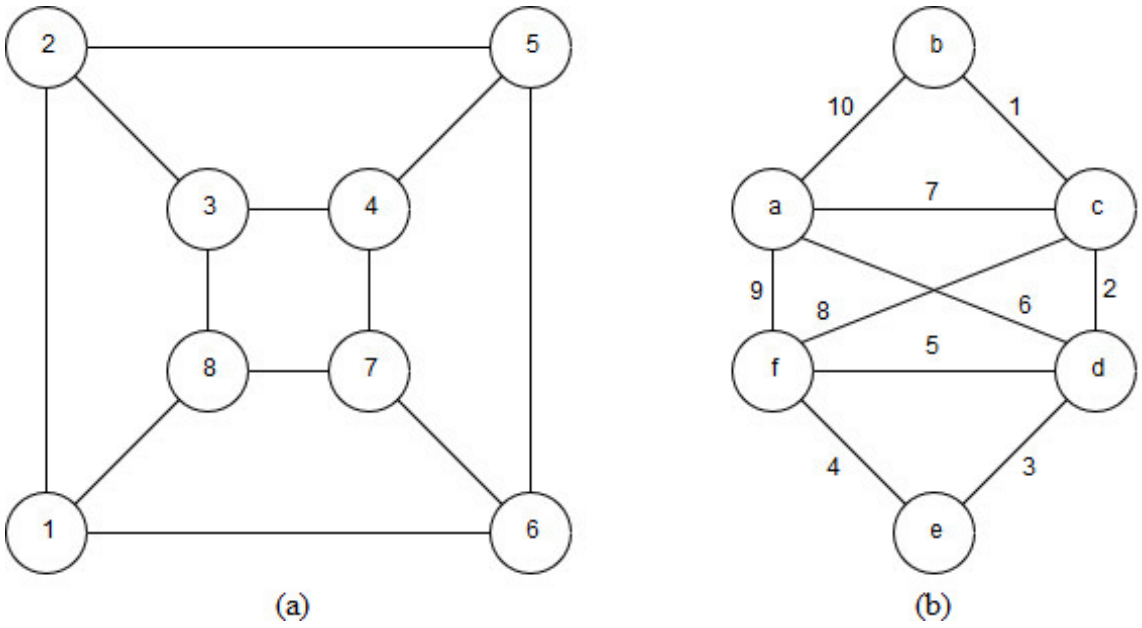


FIGURA 5.14: (a) Grafo hamiltoniano y (b) grafo euleriano.

5.2.7 GRAFOS CONEXOS Y CONECTIVIDAD

Se llama grafo conexo a un grafo no direccionado en el que existe un camino entre todo par de vértices. Los grafos de la figura 5.14 son conexos. En caso contrario, se dice que el grafo es desconexo. En este último caso, se llaman componentes conexas a los subgrafos conexos maximales, es decir, que no estén contenidos en otro subgrafo conexo. La figura 5.15 muestra un grafo desconexo con componentes conexas $\{a, b, c, d\}$, $\{e\}$ y $\{f, g\}$. Un tipo particular de grafo conexo son los grafos conexos acíclicos (como el de la figura 5.16), también llamados árboles.

En un grafo direccionado se distinguen las siguientes definiciones:

- G es fuertemente conexo si para todo v existe un camino direccionado de u a v y otro camino direccionado de v a u (figura 5.17, G_1).
- G es débilmente conexo si el grafo subyacente es conexo. El grafo subyacente es el grafo no dirigido que se obtiene al quitar la orientación de las aristas. El grafo G_2 de la figura 5.17 es débilmente conexo.
- G es unilateralmente conexo si para todo u, v existe un camino direccionado (respetando las direcciones) de u a v o bien de v a u (figura 5.17, G_3).
- G es desconexo si el grafo subyacente es desconexo (figura 5.17, G_4).

Se dice que el vértice v alcanza a w si existe un camino de v a w.

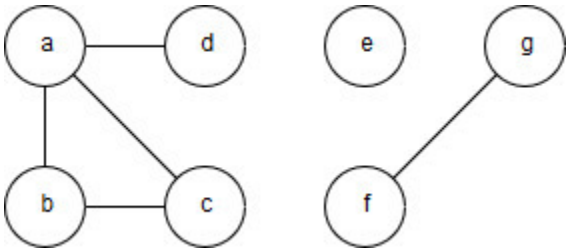


FIGURA 5.15: Grafo desconexo.

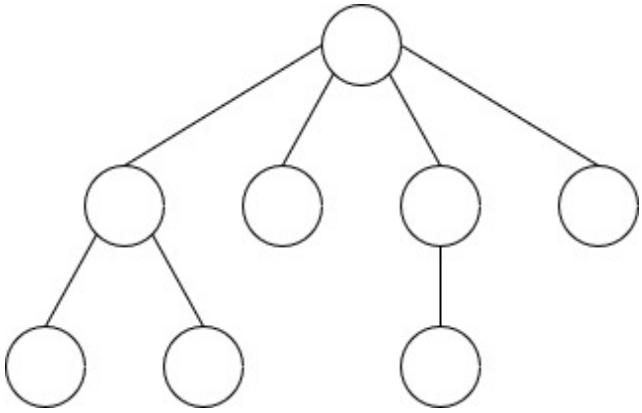


FIGURA 5.16: Grafo conexo acíclico.

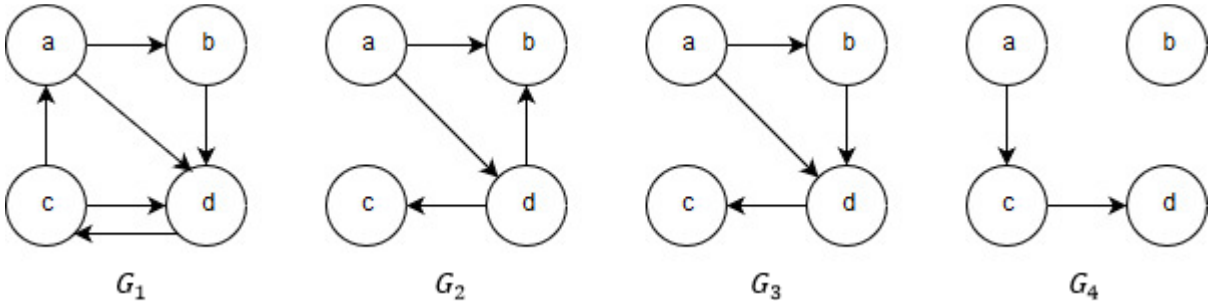


FIGURA 5.17: Grafos dirigidos con distintos niveles de conexión.

Un aspecto importante en el estudio de grafos es el de la conectividad, para el cual se manejan los siguientes conceptos:

- Dado un grafo G conexo, un corte de vértices U de G es un conjunto de vértices $U \subseteq V$, tal que $G(V-U, A)$ es desconexo (en otras palabras, G es desconexo si se le quitan los vértices contenidos en U).
- Dado un grafo conexo G, un corte de aristas de G es $B \subseteq A$ tal que $G(V, A-B)$ es desconexo.
- Se denota C_V a la conectividad de vértices, definida como el menor número de vértices necesarios para desconectar un grafo. En el grafo G_1 de la figura 5.18, $C_V = 1$.
- Se denota C_A a la conectividad de aristas, que corresponde al menor número de aristas necesarias para desconectar el grafo. En el grafo G_2 de la figura 5.18, $C_A = 2$.

- Se llama articulación de G a un vértice $v \in V$ tal que, si es eliminado del grafo G, genera un grafo $G - v$ desconexo. En el grafo G_2 de la figura 5.18, el vértice c es una articulación.
- Se llama puente de G a una arista $a \in A$ tal que, si es eliminada del grafo G, hace que éste sea desconexo. En la figura 5.18, (b, c) es un puente en G_1, a la vez que los vértices b y c son articulaciones. G_2 no tiene puentes.
- Dado un grafo G, se dice que es k-conexo en vértices si $C_V \geq k$:
  - Si $C_V = 0$, entonces el grafo es desconexo.
  - Si $C_V = 1$, entonces el grafo es conexo y el vértice es articulación.
  - Si $C_V \geq 1$, entonces el grafo es conexo.

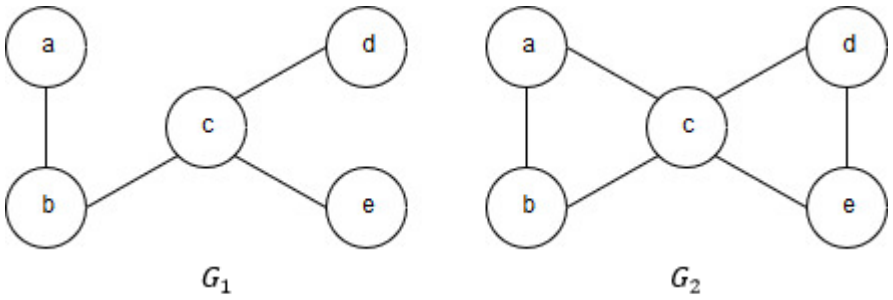


FIGURA 5.18: Grafos G_1 y G_2 para explicar conceptos de conectividad.

EJEMPLO 5.5:

Para el caso del corte de vértices, en el grafo G_1 de la figura 5.18 se puede ver que, si se retiran los vértices b y c, el grafo resultante es desconexo, con:
$U' = \{b, c\}$ corte de vértices.
$U = \{c\}$ corte minimal de vértices.

Si al grafo G_2 de la figura 5.18 se le quita el conjunto de aristas $B = \{(a, c), (b, c)\}$, el grafo resultante es desconexo. Así, B es un corte de aristas.

5.2.8 CLIQUE

Una clique es un subrafo completo con conjunto de aristas K tal que, dados $G(V, A)$ y $K \subseteq V$, $\forall v, w \in K$ se tiene $(v, w) \in A$. Además:

- Una clique es máxima si no existe otra de mayor cardinal.
- Una clique es maximal si no existe otra que la contenga en forma propia.
- Toda clique máxima es maximal.

EJEMPLO 5.6:

En el grafo G_1 de la figura 5.19 es posible distinguir diferentes cliques, entre ellas:

- Clique máxima $\{a, b, d, e\}$, que es también maximal.
- La clique maximal $\{b, c, d\}$, pues no hay otra clique que la contenga.
- La clique no maximal $\{a, b, d\}$.

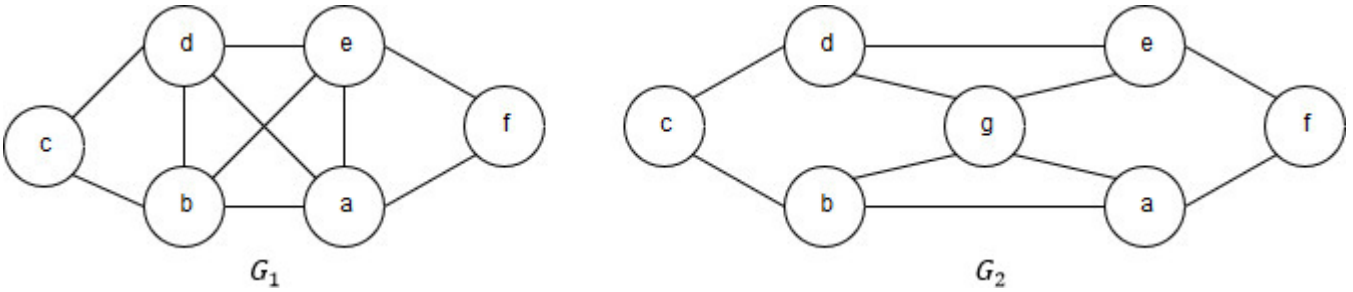


FIGURA 5.19: Grafos G_1 y G_2 para explicar conceptos de clique y conjunto independiente.

5.2.8 CONJUNTO INDEPENDIENTE

Un conjunto independiente o estable I es un subgrafo desconexo tal que, dados $G(V, A)$ y $I \subseteq V$, $\forall v, w \in I$ se tiene $(v, w) \notin A$. Además:

- Un conjunto independiente es máximo si no existe otro de mayor cardinal ($I_{máximo} \Leftrightarrow \forall J$ conjunto independiente de G, $|J| \leq |I|$).
- Un conjunto independiente es maximal si no existe otro que lo contenga en forma propia ($I_{maximal} \Leftrightarrow \exists J$ conjunto independiente de G, $I \subseteq J$).
- Todo conjunto independiente máximo es maximal.

EJEMPLO 5.7:

En el grafo G_2 de la figura 5.19 se distinguen tres conjuntos independientes máximos y maximales, pues no hay otros conjuntos independientes que los contengan:

- $\{c, f, g\}$.
- $\{a, c, e\}$.
- $\{b, d, f\}$.

5.3 EL TDA GRAFO

Se pueden usar muchas estructuras de datos para representar un grafo $G(V, A)$. No obstante, la selección de una estructura adecuada depende del tipo de operaciones que se realicen sobre los vértices y las aristas de G, así como de la cantidad de aristas con respecto a la cantidad de nodos. No obstante, no existe un conjunto básico definido de operaciones a realizar sobre grafos.

5.3.1 MATRIZ DE ADYACENCIA

Una de las representaciones más comunes corresponde a la matriz de adyacencia (cuya estructura de datos se muestra en el algoritmo 5.1). Si G tiene n vértices, su matriz de adyacencia es una matriz booleana A de $n \times n$, donde:

- Si G es dirigido, $A[i, j] = verdadero$ si y solo si existe una arista que va del vértice i al vértice j.

- Si G no es dirigido, $A[i,j] = A[j,i] = verdadero$ si y solo si existe una arista que una los vértices i y j.

ALGORITMO 5.1: Estructura de datos para representar un grafo como matriz de adyacencia.

```
Tipo grafo:
      vértices: entero
      matriz: booleano[vértices][vértices]
```

La figura 5.20 muestra dos grafos y sus respectivas matrices de adyacencia. Cabe destacar que, para grafos no dirigidos, la matriz de adyacencia siempre es simétrica.

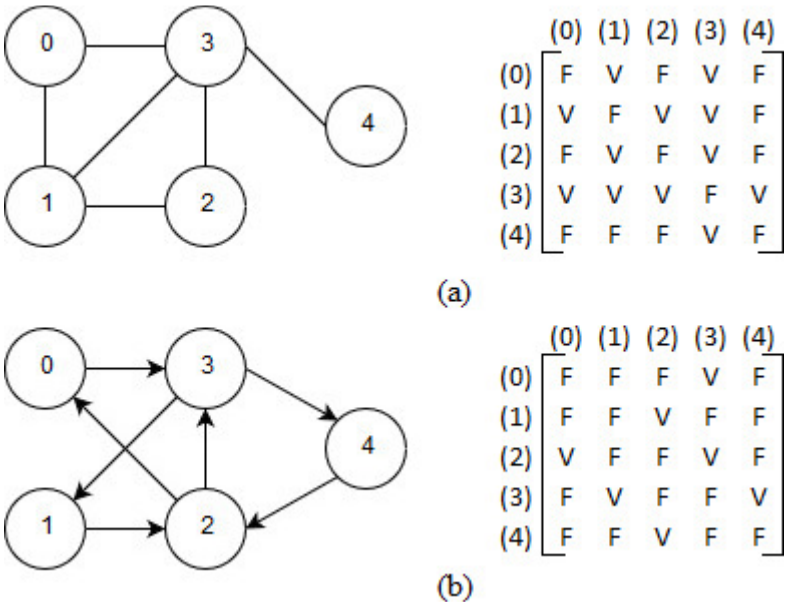


FIGURA 5.20: Matrices de adyacencia para (a) un grafo dirigido y (b) un grafo no dirigido.

Una variante muy relacionada corresponde a la matriz de adyacencia con anotaciones, donde $A[i,j]$ contiene a la anotación de la arista (i,j). Si tal arista no existe, $A[i,j]$ debe contener un valor fuera de rango o no admitido. La figura 5.21 muestra la matriz de adyacencia con anotaciones para un grafo. Se usa el valor 0 como indicador de que la arista está ausente.

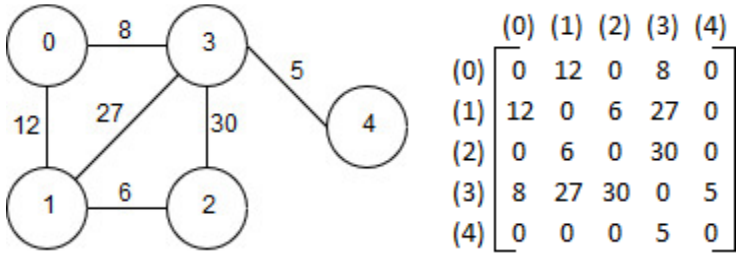


FIGURA 5.21: Matriz de adyacencia con anotaciones para un grafo.

La matriz de adyacencia es una representación adecuada para algoritmos en los que frecuentemente se requiera determinar si una arista está o no presente en G. Sin embargo, la principal desventaja de esta representación es que requiere $O(n^2)$ espacio de almacenamiento aunque la cantidad de aristas sea significativamente menor que n^2, y solo recorrer la matriz tomaría un tiempo $O(n^2)$, lo que perjudicaría a los algoritmos para manipular grafos con $O(n)$ aristas.

5.3.2 LISTA DE ADYACENCIA

Otra representación frecuentemente utilizada para grafos corresponde a la lista de adyacencias. La lista de adyacencias para cada vértice i es una lista que, en algún orden, contiene a todos los vértices adyacentes a i. Una forma de representar esta estructura es mediante un arreglo de cabeceras C, donde $C[i]$ es un puntero a la lista de adyacencias del vértice i. La figura 5.22 muestra las listas de adyacencia para dos grafos.

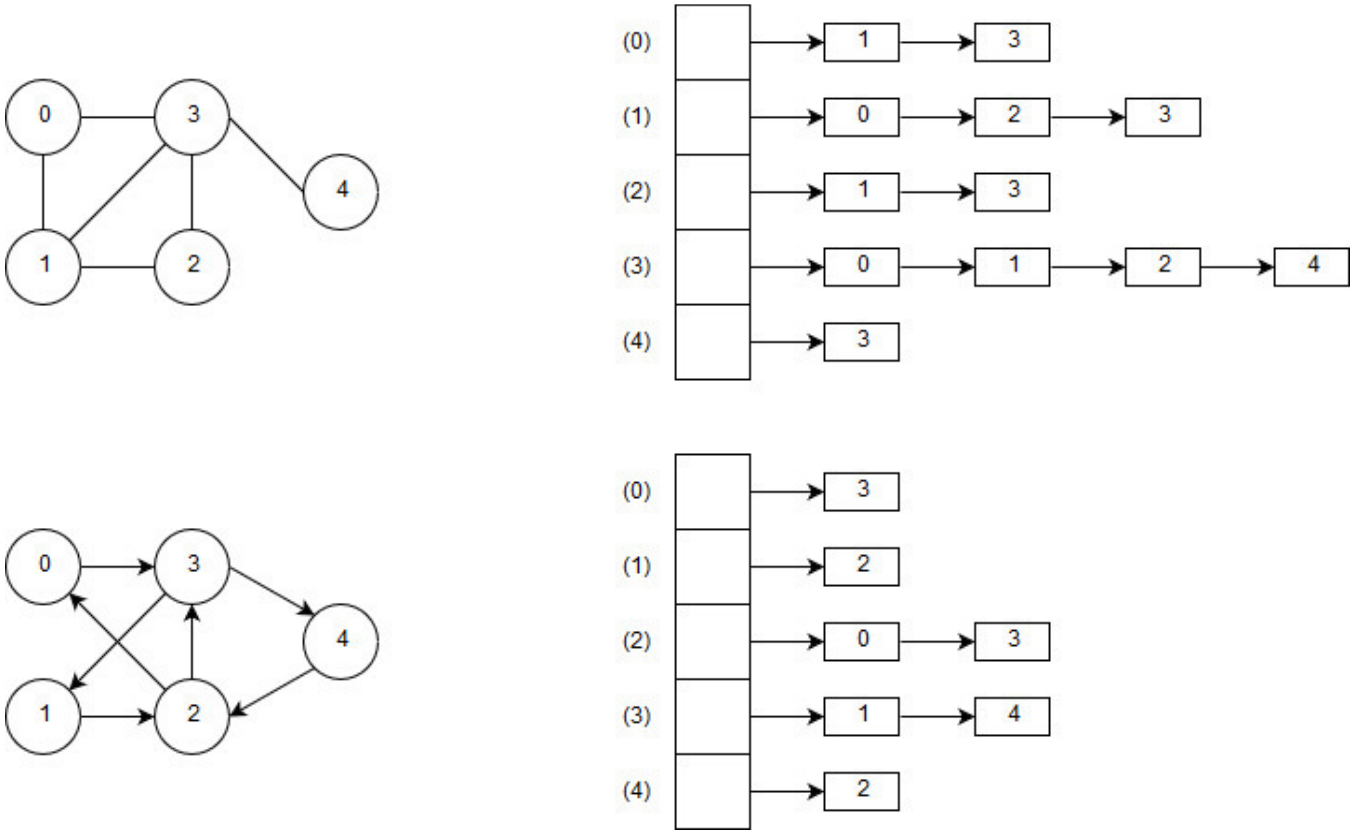


FIGURA 5.22: Listas de adyacencia para (a) un grafo dirigido y (b) un grafo no dirigido.

La representación anterior puede extenderse añadiendo un campo adicional al nodo de la lista cuando el grafo tiene anotaciones, como muestra la figura 5.23.

La representación de lista de adyacencias para un grafo requiere $O(n+m)$ espacio de almacenamiento, donde n es la cantidad de vértices y m, la de aristas. Suele emplearse cuando la cantidad de aristas es muy

inferior a n^2. Sin embargo, una desventaja potencial es que puede tomar tiempo $O(n)$ saber si existe una arista desde el vértice i al j, pues pueden haber n vértices en la lista de adyacencia de i.

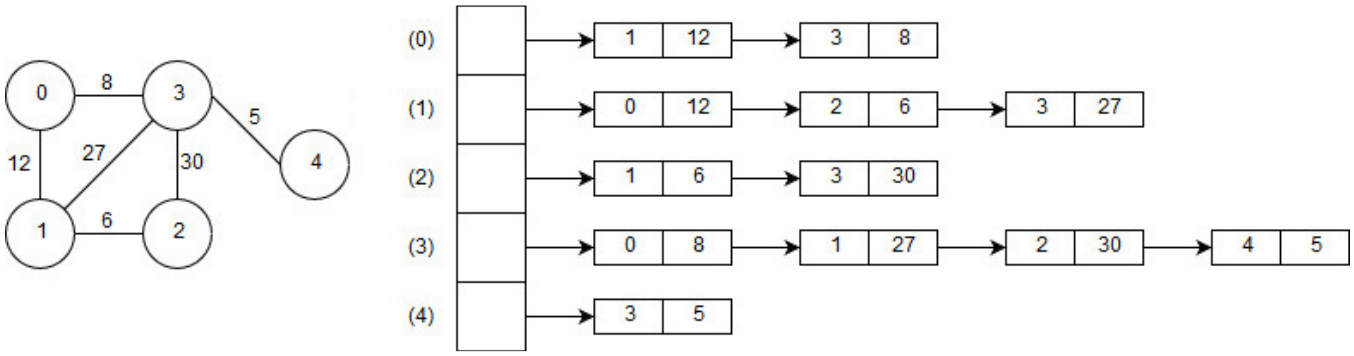


FIGURA 5.23: Lista de adyacencia con anotaciones para un grafo.

5.4 MODELAMIENTO DE PROBLEMAS USANDO GRAFOS

Los grafos son una herramienta muy útil para modelar una gran variedad de problemas, por ejemplo:

- *Redes sociales:* cada vértice corresponde a una persona y se crea una arista entre dos personas si existe una relación entre ellas.
- *Transporte:* cada dirección o ciudad corresponde a un vértice, y se unen mediante aristas que corresponden a rutas entre lugares. Se puede incorporar a la arista un rótulo con la distancia entre ambos lugares.
- *Dependencia:* permite modelar cómo módulos de software, conjuntos de datos y funciones dependen entre ellos.
- *Tecnología:* definir la conectividad de enrutadores, páginas Web, etc.
- *Lenguaje:* definir la relación entre palabras.

Esto resulta sumamente útil al momento de buscar soluciones computacionales para los problemas, puesto que se puede establecer una representación más clara del problema y se puede recurrir a algoritmos ya conocidos para intentar su resolución.

EJEMPLO 5.8:

El problema del vendedor viajero es un problema de gran interés computacional porque tiene aplicación en áreas tan diversas como la planificación, la logística, la fabricación de microchips e incluso el estudio de la secuencia de ADN. Su formulación es muy simple: dada una lista de ciudades y las distancias entre cada par de ellas, ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y regresa a la ciudad origen?

Considere la tabla 5.1, que muestra las distancias existentes entre 4 localidades de la Región Metropolitana. Se puede modelar dicha información usando grafos, resultando el grafo de la figura 5.24.

TABLA 5.1: Distancias entre 4 ciudades de la Región Metropolitana.

	Santiago	Peñaflor	Buin
Peñaflor	35,54	-	-
Buin	35,34	33,52	-
Melipilla	73,1	47,09	67,52

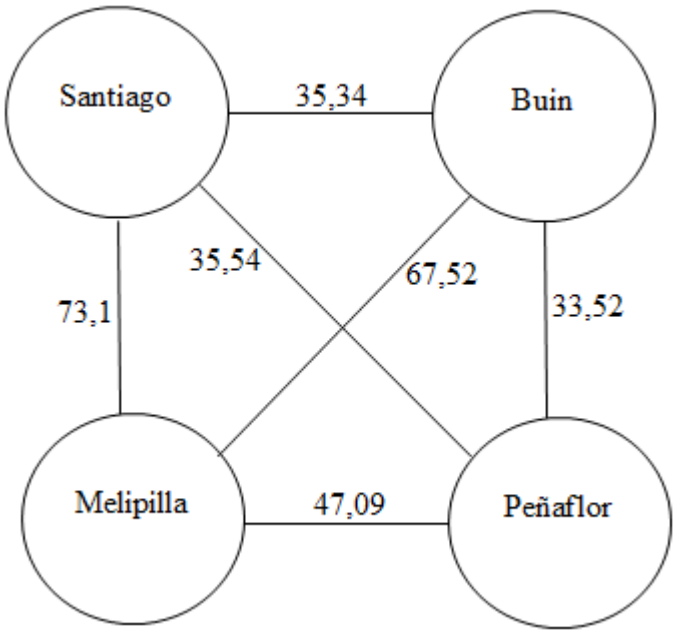


FIGURA 5.24: Grafo que modela la distancia entre ciudades de la tabla 5.1.

5.5 RECORRIDOS DE GRAFOS

Existen dos grandes métodos para recorrer grafos: recorrido en anchura y recorrido en profundidad. Ambos tienen igual complejidad, aunque en la práctica el recorrido en anchura es más eficiente al no ser recursivo. Cualquiera de ellos puede ser fácilmente modificado para poder buscar un vértice en un grafo.

5.5.1 RECORRIDO EN ANCHURA

El criterio utilizado en el recorrido en anchura de un grafo no dirigido es crear una cola de vértices visitados con el vértice inicial. Mientras la cola no esté vacía, se quita el vértice ubicado al comienzo de la cola y se recorre su conjunto de adyacencia. Cuando se encuentre un vértice no visitado, se incorpora al recorrido la arista que permite alcanzarlo y se agrega el vértice al final de la cola.

Se puede comprobar que, en el recorrido construido, se obtiene un grafo con los mismos vértices que el original, pero con un único camino desde el vértice inicial hasta cada nodo (si el grafo es desconexo, solo existen caminos para los vértices situados en la componente conexa del vértice inicial), de donde se desprende que el recorrido es un grafo acíclico (es decir, un árbol).

ALGORITMO 5.2: Recorrido en anchura de un grafo.

```
RECORRIDO_ANCHURA(g: grafo, inicial: vértice): grafo
      grafo recorrido
      cola q
      recorrido.vértices = {inicial}
      recorrido.aristas = {}
      marcados[g.vértices]: booleano[]

      Para i desde 0 hasta g.vértices - 1:
            marcados[i] = falso

      q = QUEUE(q, inicial)
      marcados[inicial] = verdadero

      Mientras VACIA(q) == falso:
            u = INICIO(q)
            q = DEQUEUE(q)

            Para todo v en Ady(u):
                  Si marcados[v] == falso:
                        marcados[v] = verdadero
                        q = QUEUE(q, v)
                        recorrido.vértices = recorrido.vértices U {v}
                        recorrido.aristas = recorrido.aristas U {(u,v)}

      Devolver recorrido
```

EJEMPLO 5.9:

La figura 5.25 muestra un grafo y su recorrido en anchura, comenzando desde el vértice 0. Para obtener este resultado, se considera que los vértices adyacentes se recorren en orden creciente.

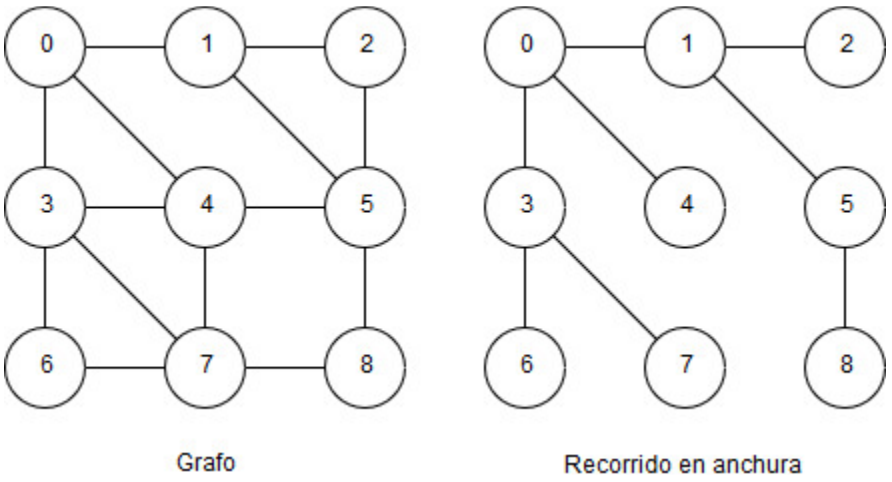


FIGURA 5.25: Un grafo y su recorrido en anchura con vértice inicial 0.

La complejidad del recorrido en anchura está determinada porque visita todos los vértices y todas las aristas del grafo, luego es $O(n+m)$. En el peor de los casos, si el grafo es completo, se tiene que el

número de aristas es $m = n(n-1)/2 = (n^2 - n)/2 \sim O(n^2)$ para un grafo no dirigido, y que para un grafo dirigido, $m = n(n-1) = n^2 - n \sim O(n^2)$. Así, $O(n+m) = O(n+n^2) = O(n^2)$.

5.5.2 RECORRIDO EN PROFUNDIDAD

En cierto modo opuesto al recorrido en anchura, que intenta alcanzar todos los vértices posibles desde el punto de partida antes de avanzar, el recorrido en profundidad (algoritmo 5.3) intenta recorrer siempre el camino más largo posible a partir de un vértice inicial. Opera usando una pila con el vértice inicial. Luego visita los vértices adyacentes al inicial y, cuando encuentra uno no alcanzado, comienza desde allí un nuevo recorrido recursivamente. Se retira el vértice inicial de la pila una vez que se termina de comprobar su conjunto de adyacencia.

Al igual que en el recorrido en anchura, la complejidad de este algoritmo es $O(n+m) \sim O(n^2)$ y el recorrido construido corresponde a un árbol.

ALGORITMO 5.3: Recorrido en profundidad de un grafo.

```
RECORRIDO_PROFUNDIDAD(g: grafo, inicial: vértice): grafo
      grafo recorrido
      pila p
      recorrido.vértices = {inicial}
      recorrido.aristas = {}
      marcados[g.vértices]: booleano[]

      Para i desde 0 hasta g.vértices - 1:
            marcados[i] = falso

      recorrido = PROCEDIMIENTO(g, inicial, p, recorrido, marcados)
      Devolver recorrido

PROCEDIMIENTO(g: grafo, u: vértice, p: pila, recorrido: grafo,
              marcados: booleano[n]): grafo
      p = PUSH(p, u)
      marcados[u] = verdadero

      Para todo v en Ady(u):
            Si marcados[v] == falso:
                  marcados[v] = verdadero
                  recorrido.vértices = recorrido.vértices ∪ {v}
                  recorrido.aristas = recorrido.aristas ∪ {(u,v)}
                  recorrido = PROCEDIMIENTO(g, v, p, recorrido, marcados)

      p = POP(p)
      Devolver recorrido
```

EJEMPLO 5.10:

La figura 5.26 muestra un grafo y su recorrido en anchura, comenzando desde el vértice 0.

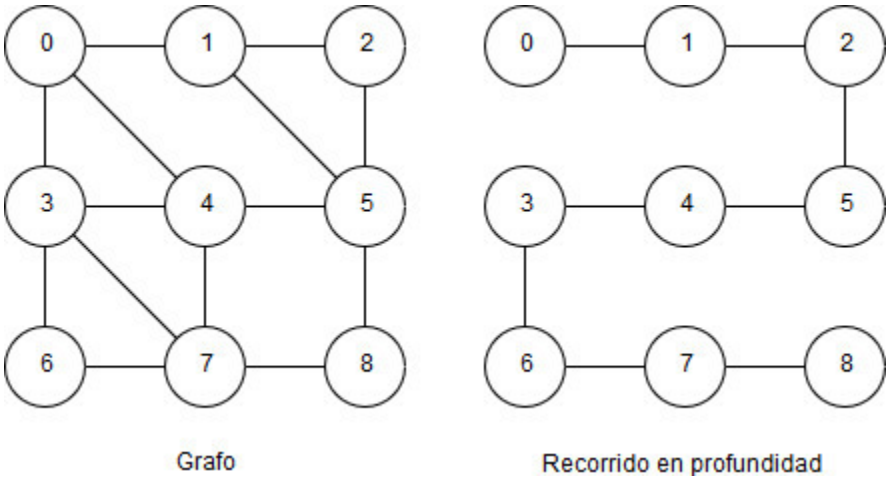


FIGURA 5.26: Un grafo y su recorrido en profundidad con vértice inicial 0.

5.6 GRAFOS DE PROXIMIDAD

5.6.1 VECINO MÁS CERCANO

Sea G un grafo conexo. Se desea encontrar una forma rápida de alcanzar los vértices del grafo desde un vértice inicial dado. El algoritmo del vecino más cercano es una aproximación rápida que, en general, entrega una buena solución en un lapso de tiempo reducido.

La idea de este método (algoritmo 5.4) es sencilla: comenzar desde un vértice dado y escoger como siguiente vértice aquel vértice adyacente al actual que no haya sido visitado que tenga asociada la arista de menor costo.

Este algoritmo, al tomar la mejor decisión inmediata sin considerar la totalidad del problema, por lo general no entrega la mejor solución posible, pudiendo incluso no encontrar solución alguna.

EJEMPLO 5.11:

La figura 5.27 muestra la ejecución del algoritmo del vecino más cercano, comenzando desde el vértice 0.

5.6.2 ÁRBOL DE COBERTURA DE COSTO MÍNIMO

Sea $G(V, A)$ un grafo conexo en que cada arista (u, v) tiene asociado un costo $c(u, v)$. Un árbol de cobertura es un grafo acíclico que conecta todos los vértices en V. El costo de dicho árbol corresponde a la suma de los costos asociados a todas las aristas del árbol.

El árbol de cobertura de costo mínimo, correspondiente al árbol de cobertura de un grafo cuyo costo total es el menor posible, es ampliamente utilizado para el diseño de redes de computadores, donde cada vértice representa una ciudad y una arista corresponde a un posible enlace de comunicaciones.

ALGORITMO 5.4: Algoritmo del vecino más cercano.

```
VECINO_MÁS_CERCANO(g: grafo, inicial: vértice): grafo
      grafo árbol
      alcanzados[n]: booleano[]

      Para i desde 0 hasta n – 1:
            alcanzados[i] = falso

      árbol.vértices = {}
      árbol.aristas = {}

      árbol.vértices = árbol.vértices ∪ {inicial}
      alcanzados[inicial] = verdadero
      cambios = verdadero
      u = inicial

      Mientras cambios:
            cambios = falso
            menorCosto = ∞

            Para todo v en ady(u):
                  Si alcanzados[v] == falso y (u,v).costo < menorCosto:
                        menorCosto = (u,v).costo
                        menorV = v
                        cambios = verdadero

            Si cambios:
                  alcanzados[menorV] = verdadero
                  (u,menorV).costo = menorCosto
                  árbol.vértices = árbol.vértices ∪ {menorV}
                  árbol.aristas = árbol. aristas ∪ {(u,menorV)}
                  u = menorV

      Devolver árbol
```

Existen diversos algoritmos para construir el árbol de cobertura de costo mínimo para un grafo, muchos de ellos basados en la siguiente propiedad: dado $U \subset V$, si $(u, v) \in A$ tiene el mismo costo para las aristas en que $u \in U$ y $v \in V - U$, entonces existe un árbol de cobertura de costo mínimo que incluye a (u, v). A continuación, se explican los principales.

Algoritmo de Prim

Como muestra el algoritmo 5.5, el algoritmo de Prim comienza con un conjunto de vértices alcanzados que solo contiene al vértice inicial y, a partir de ahí, construye un árbol de cobertura incorporando una arista a la vez. En cada iteración, encuentra la arista (u, v) de menor costo que conecte un vértice alcanzado con otro que no haya sido previamente alcanzado. Este proceso se repite hasta que se alcancen todos los vértices. Note que n corresponde a la cantidad de vértices y m, a la de aristas.

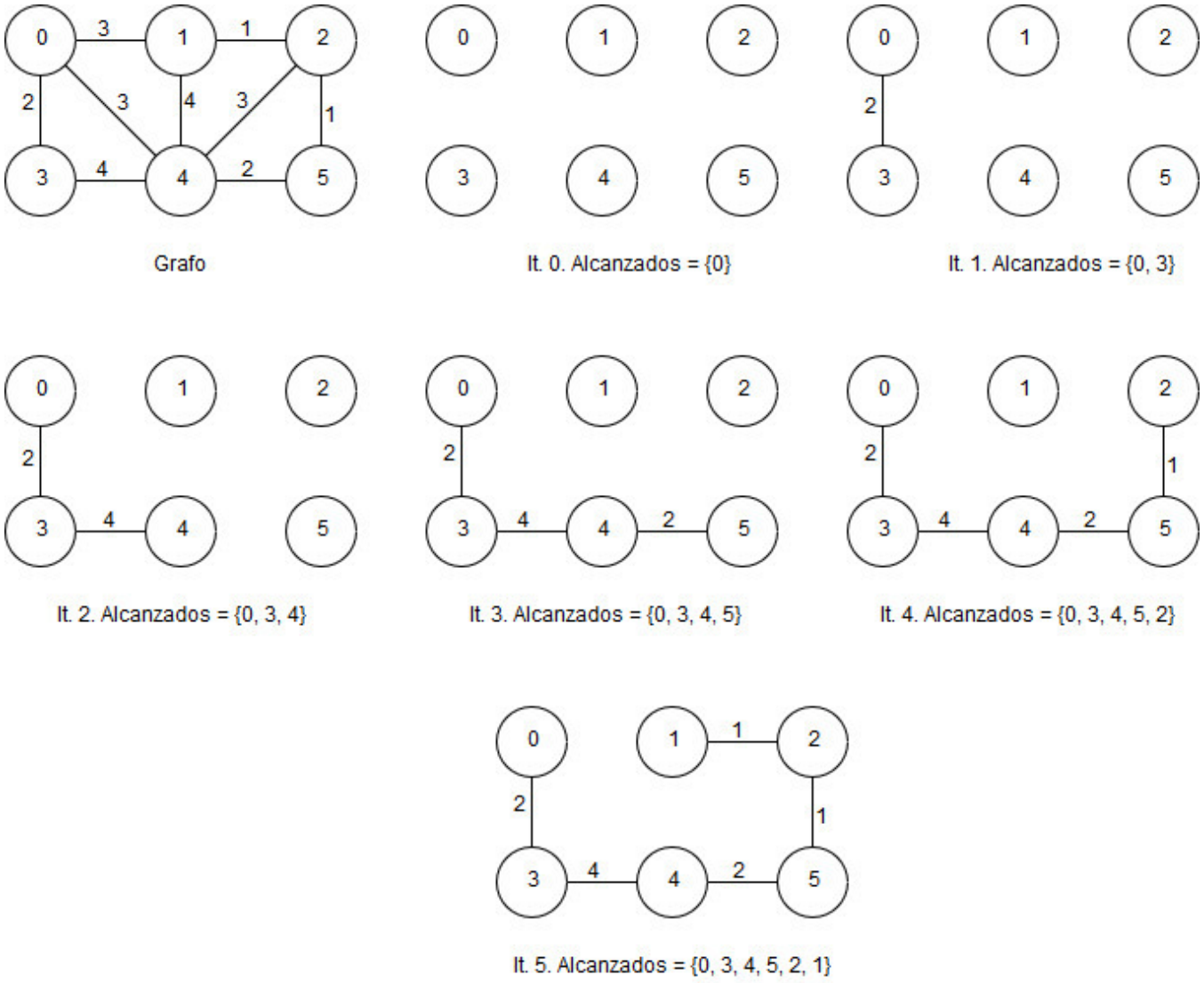


FIGURA 5.27: Funcionamiento del algoritmo del vecino más cercano.

EJEMPLO 5.12:

La figura 5.28 muestra la ejecución del algoritmo de Prim, comenzando desde el vértice 0.

Algoritmo de Kruskal

Sea $G(V, A)$ un grafo conexo en que cada arista (u, v) tiene asociado un costo $c(u, v)$. El algoritmo de Kruskal (algoritmo 5.6) comienza con un árbol que es un grafo desconexo en que cada vértice está en una componente conexa diferente (no tiene ninguna arista). Luego ordena las aristas del grafo original en forma no decreciente según sus costos. Luego revisa cada arista, respetando el orden, para ver si conectan componentes diferentes. Si es así, incorpora dicha arista al árbol. El algoritmo termina cuando todos los vértices están en la misma componente conexa. Cabe señalar que `ORDENAR()` puede ser cualquiera de los

algoritmos de ordenamiento estudiados en el capítulo 3. Además, n corresponde a la cantidad de vértices y m, a la de aristas.

EJEMPLO 5.13:

La figura 5.29 muestra la ejecución del algoritmo de Kruskal.

ALGORITMO 5.5: Algoritmo de Prim.

```
PRIM(g: grafo, inicial: vértice): grafo
      grafo árbol
      alcanzados[n]: booleano[]

      Para i desde 0 hasta n - 1:
            alcanzados[i] = falso

      árbol.vértices = {}
      árbol.aristas = {}

      árbol.vértices = árbol.vértices ∪ {inicial}
      alcanzados[inicial] = verdadero
      cambios = verdadero

      Mientras cambios:
            cambios = falso
            menorCosto = ∞

            Para todo u en alcanzados:
                  Para todo v en ady(u):
                        Si alcanzados[v] == falso y (u,v).costo < menorCosto:
                              menorCosto = (u,v).costo
                              menorU = u
                              menorV = v

            Si menorCosto < ∞:
                  cambios = verdadero
                  alcanzados[menorV] = verdadero
                  árbol.vértices = árbol.vértices ∪ {menorV}
                  árbol.aristas = árbol.aristas ∪ {(menorU,menorV)}

      Devolver árbol
```

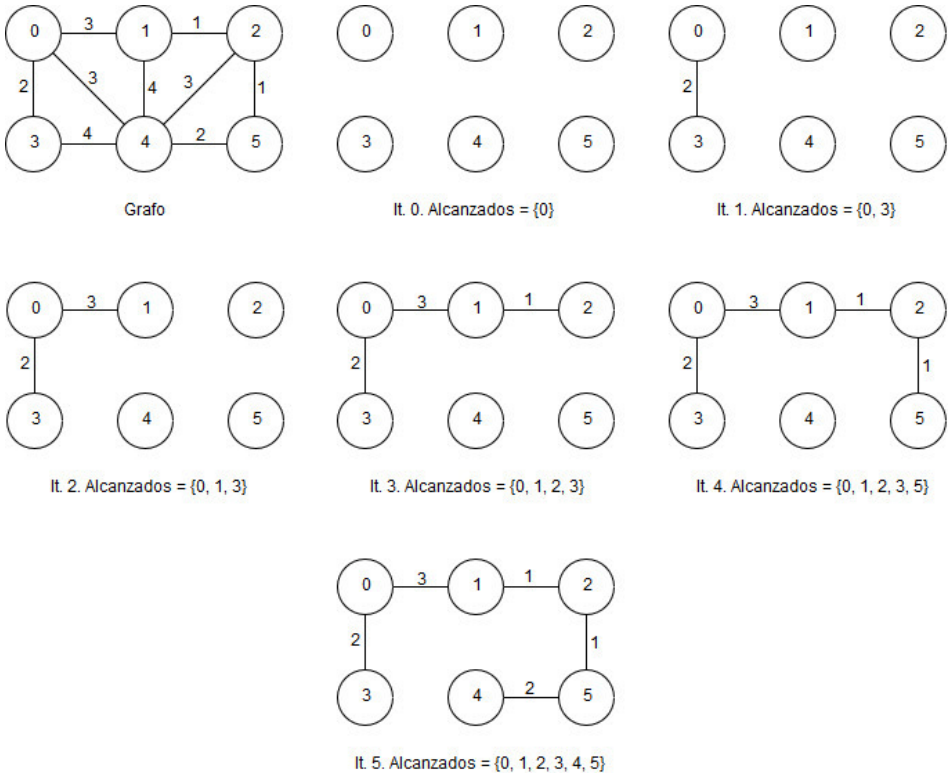


FIGURA 5.28: Funcionamiento del algoritmo de Prim.

ALGORITMO 5.6: Algoritmo de Kruskal.

```
KRUSKAL(g: grafo): grafo
    grafo árbol
    componente[g.vértices]: entero[]

    aristas = ORDENAR(g.aristas)
    árbol.vértices = g.vértices
    árbol.aristas = {}

    Para v desde 0 hasta n - 1:
        componente[v] = v

    Para i desde 0 hasta m - 1:
        u = aristas[i].origen
        v = aristas[i].destino

        Si componente[u] != componente[v]:
            árbol.aristas = árbol.aristas U {(u, v)}
            reemplazar = componente[v]

            Para j desde 0 hasta n - 1:
                Si componente[j] == reemplazar:
                    componente[j] = componente[u]

      Devolver árbol
```

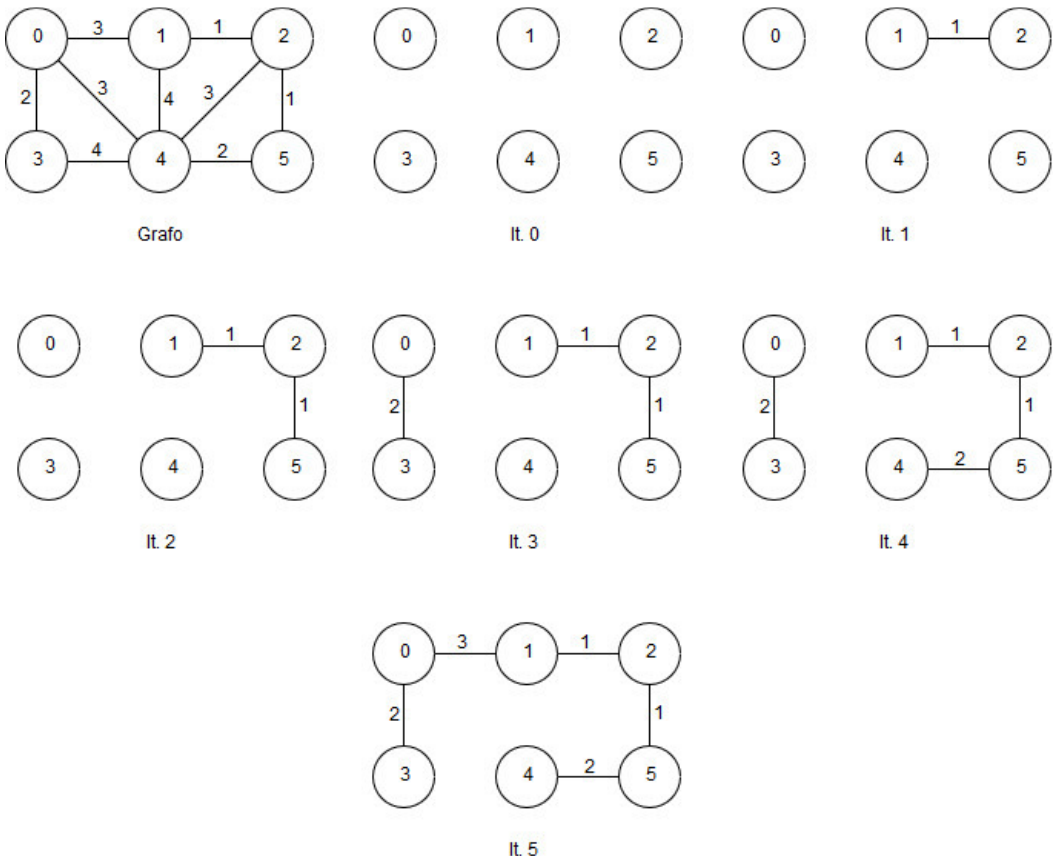


FIGURA 5.29: Funcionamiento del algoritmo de Kruskal.

5.7 CAMINO MÍNIMO

Este algoritmo, denominado también algoritmo de Dijkstra, determina el costo más bajo (la menor distancia) para llegar desde un nodo inicial hasta cada uno de los vértices restantes, almacenando además información que permita identificar la ruta para llegar desde el vértice inicial al destino. El algoritmo opera de la siguiente manera:

1. La distancia mínima para el vértice inicial es 0, mientras que para todos los demás se inicializa con valor infinito.
2. Se marca el vértice inicial como alcanzado.
3. Comenzando desde el vértice inicial, se calcula la distancia para llegar a cada uno de sus vecinos y se registra en la tabla (en este caso, la distancia corresponde simplemente al costo de la arista).
4. Se escoge el vértice no visitado con distancia más baja y se selecciona como vértice actual.
5. Para el vértice actual, se calcula la distancia para llegar a cada uno de sus vecinos y se registra en la tabla (en este caso, la distancia corresponde al costo de alcanzar el vértice sumada al costo de la arista revisada).
6. Se repite desde el paso 4 hasta que no sea posible alcanzar un nuevo vértice.

El algoritmo 5.7 muestra con más detalle la explicación anterior.

ALGORITMO 5.7: Algoritmo de Dijkstra para obtener el camino mínimo.

```
tipo casilla:
      anterior: entero
      costo: entero

Dijkstra(g: grafo, inicial: vértice): casilla[]
      camino[n]: casilla[]
      alcanzados[n]: booleano[]

      Para i = 0 i < g.vértices i++:
            camino[i].anterior = -1
            alcanzados[i] = falso

            Si i != inicial:
                  camino[i].costo = ∞
            Sino:
                  camino[i].costo = 0

      u = inicial
      costoU = 0

      Mientras u != -1 y costoU != ∞:
            alcanzados[u] = verdadero

            Para v en ady(u):
                  Si alcanzados[v] == falso:
                        nuevoCosto = costoU + (u, v).costo

                        Si nuevoCosto < camino[v].costo:
                              camino[v].anterior = u
                              camino[v].costo = nuevoCosto

            costoU = ∞
            u = -1

            Para v en g.vértices:
                  Si alcanzados[v] == falso y camino[v].costo < costoU:
                        costoU = camino[v].costo
                        u = v

      Devolver camino
```

Como resultado de la aplicación del algoritmo, las aristas tendrán asociado un valor correspondiente a la distancia del camino mínimo a recorrer desde el origen O hasta el destino d.

EJEMPLO 5.14:

La figura 5.30 muestra un grafo para ejemplificar el funcionamiento del algoritmo de Dijkstra. Los resultados se muestran en la tabla 5.2.

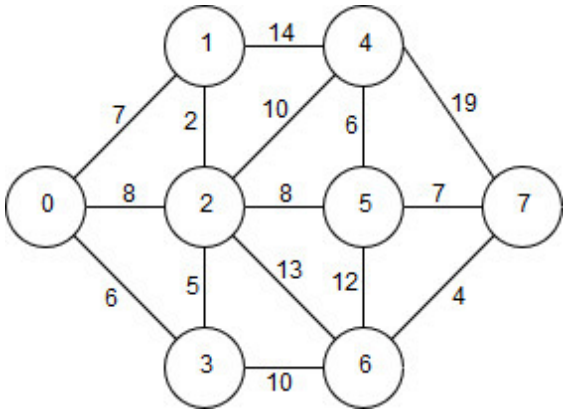


FIGURA 5.30: Grafo para el ejemplo 5.14.

TABLA 5.2: Traza del algoritmo de Dijkstra para el grafo de la figura 5.14 con vértice inicial 0.

	0		1		2		3		4		5		6		7	
u	C	A	C	A	C	A	C	A	C	A	C	A	C	A	C	A
-	0	-1	∞	-1	∞	-1	∞	-1	∞	-1	∞	-1	∞	-1	∞	-1
0			7	0	8	0	6	0	∞	-1	∞	-1	∞	-1	∞	-1
3			7	0	8	0			∞	-1	∞	-1	16	3	∞	-1
1					8	0			21	1	∞	-1	16	3	∞	-1
2									18	2	16	2	16	3	∞	-1
5									18	2			16	3	23	5
6									18	2					20	6
4															20	6
7																

La figura 5.31 destaca el camino mínimo para ir desde el vértice 0 hasta el vértice 7, con un costo total de 20. Para obtener esta información a partir del camino entregado por el algoritmo de Dijkstra, basta con saber que `camino[7].costo` contiene el costo total del camino para llegar al vértice 7 desde el vértice 0 (que es de 20), mientras que `camino[7].anterior` contiene el vértice desde el que se llega a 7 (es decir, 6). A su vez, con `camino[6].anterior` se sabe que el vértice 6 se alcanza desde el vértice 3 y, análogamente, `camino[3].anterior` indica que el vértice 3 es alcanzado desde el vértice 0. Finalmente se tiene que `camino[0].anterior` es -1, lo cual señala que el vértice 0 es el punto de partida.

5.8 FLUJO EN REDES

Considere un grafo dirigido (también llamado red) con dos vértices particulares, fuente y sumidero, y un valor asociado a cada arista correspondiente a su capacidad. Un flujo es una función asigna a cada arista un valor entre 0 y su capacidad, señalando cuánto ha sido ocupado de la capacidad de dicha arista. Los flujos deben respetar la ley de conservación, la cual establece que, para cada vértice excepto la fuente y el

sumidero, el flujo que entra es igual que el que sale. El valor del flujo de una red corresponde a lo que entra al sumidero, y debe ser igual a lo que sale de la fuente. Lo que se busca es un flujo de valor máximo.

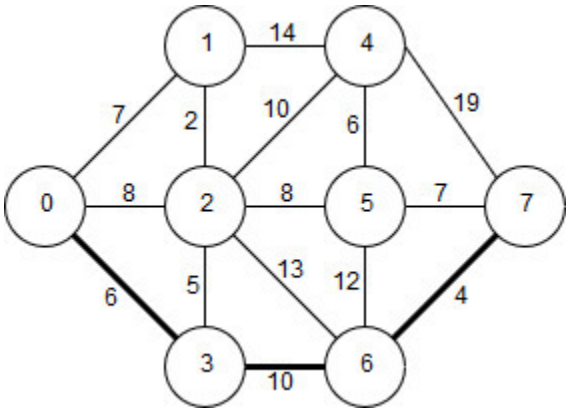


FIGURA 5.31: camino mínimo para ir desde el vértice 0 al vértice 7 en el grafo de la figura 5.30.

El estudio de algoritmos para resolver este problema teórico es de suma importancia, ya que sirve para modelar una gran variedad de problemas de la vida real, entre ellos:

- Transporte de mercadería (logística).
- Flujo de gases y líquidos por tuberías.
- Flujo de componentes o piezas en líneas de montaje.
- Flujo de corriente en redes eléctricas.
- Flujo de paquetes de información en redes de comunicaciones.

Uno de los algoritmos clásicos para resolver el problema de flujo en redes es el de Ford-Fulkerson. Antes de enunciar dicho algoritmo, se requiere definir algunos conceptos previos.

5.8.1 GRAFO RESIDUAL Y CAMINO DE AUMENTO

Se define la capacidad residual de una arista como la capacidad resultante una vez que se hace uso de un flujo. Para ilustrar esta idea, considere una arista (u, v) con capacidad $c(u, v) = 8$ y un $f(u, v) = 6$. La capacidad residual de (u, v) está dada por $cr(u, v) = c(u, v) - f(u, v) = 8 - 6 = 2$.

No obstante, para respetar la ley de conservación de flujo, es necesario tener además un flujo para la arista inversa, con $f(u, v) = -f(v, u)$. Suponga que inicialmente $c(v, u) = 0$. Para conservar el flujo, se tiene que $cr(v, u) = c(v, u) - f(v, u) = c(v, u) + f(u, v) = 0 + 6 = 6$.

Considerando lo anterior, el grafo residual corresponde a un grafo con los mismos vértices que el grafo original y cuyas aristas tienen las capacidades residuales del flujo empleado.

Un camino de aumento es un camino dirigido desde la fuente hasta el sumidero en el grafo residual.

5.8.2 ALGORITMO DE FORD-FULKERSON

Es un algoritmo que determina el flujo máximo que puede existir desde la fuente hasta el sumidero de un grafo dirigido. La idea de operación es sencilla, como muestra el algoritmo 5.8:

1. Hacer que el grafo residual sea igual al grafo original.
2. Inicializar el flujo máximo en 0.
3. Mientras queden caminos desde la fuente al sumidero:
   a. Buscar un camino desde la fuente hasta el sumidero.
   b. Determinar la capacidad del camino, que es igual a la menor capacidad de las aristas en el camino.
   c. Sumar la capacidad del camino al flujo total.
   d. Actualizar los flujos de las aristas del camino con la capacidad del camino.
   e. Actualizar las aristas del grafo residual quitando la capacidad del camino.
4. Devolver el flujo total.

ALGORITMO 5.8: Algoritmo de Ford-Fulkerson para obtener el flujo máximo en una red.

```
FORD_FULKERSON(g: grafo, fuente: vértice, sumidero: vértice): entero
      grafo residual = g
      flujoTotal = 0

      Mientras exista camino de fuente a sumidero en residual:
            flujoCamino = ∞

            Para cada (u,v) en camino:
                  Si (u,v).capacidad < flujoCamino:
                        flujoCamino = (u,v).capacidad

            flujoTotal = flujoTotal + flujoCamino

            Para cada (u,v) en camino:
                  (u,v).capacidad = (u,v).capacidad – flujoCamino
                  (v,u).capacidad = (v,u).capacidad + flujoCamino
                  Reemplazar (u,v) y (v,u) en residual.aristas

      Devolver flujoTotal
```

Un problema del algoritmo 5.8, según fue propuesto originalmente, es que no proporciona un mecanismo para encontrar un camino desde la fuente hasta el sumidero. Existen diferentes mecanismos para esta tarea. Los más convencionales consisten en usar adaptaciones menores del recorrido en anchura o el recorrido en profundidad. El algoritmo 5.9 detalla el método de Ford-Fulkerson buscando los caminos residuales mediante un recorrido en anchura. El camino se entrega como un arreglo de enteros en que cada casilla representa a un vértice v y contiene al vértice u que lo antecede en el camino (al igual que la representación de los caminos empleada por el algoritmo de Dijkstra).

ALGORITMO 5.9: Algoritmo de Ford-Fulkerson adaptado para su implementación.

```
FORD_FULKERSON(g: grafo, fuente: vértice, sumidero: vértice): entero
      grafo residual = g
      flujoTotal = 0
      flujos[n][n]: entero[][]

      Para i desde 0 hasta n:
            Para j desde 0 hasta n:
                  flujos[i][j] = 0

      camino = BUSCAR_CAMINO(residual, fuente, sumidero)

      Mientras camino[sumidero] != -1:
            flujoCamino = ∞

            Para cada (u,v) en camino:
                  Si (u,v).capacidad < flujoCamino:
                        flujoCamino = (u,v).capacidad

            flujoTotal = flujoTotal + flujoCamino

            Para cada (u,v) en camino:
                  (u,v).capacidad = (u,v).capacidad - flujoCamino
                  (v,u).capacidad = (v,u).capacidad + flujoCamino
                  Reemplazar (u,v) y (v,u) en residual.aristas

            camino = BUSCAR_CAMINO(residual, fuente, sumidero)
      Devolver flujoTotal

BUSCAR_CAMINO(g: grafo, fuente: vértice, sumidero: vértice): vértice[]
    cola q
    camino[n]: vértice[]

    Para i desde 0 hasta n:
        alcanzados[i] = falso
        camino[i] = NULO

    q = QUEUE(q, fuente)
    camino[fuente] = fuente

    Mientras VACIA_COLA(q) == falso:
        u = INICIO(q)
        q = DEQUEUE(q)

        Para v desde 0 hasta n:
            Si camino[v] == NULO y (u,v) en G.aristas:
                q = QUEUE(q, v)
                alcanzados[v] = verdadero

                Si camino[sumidero] != NULO:
                    Q = ANULAR(Q)
                    Devolver camino

    Devolver camino
```

EJEMPLO 5.15:

La figura 5.32 muestra la secuencia de grafos residuales para una ejecución del algoritmo de Ford-Fulkerson. En la iteración 0, el grafo residual corresponde al grafo original.

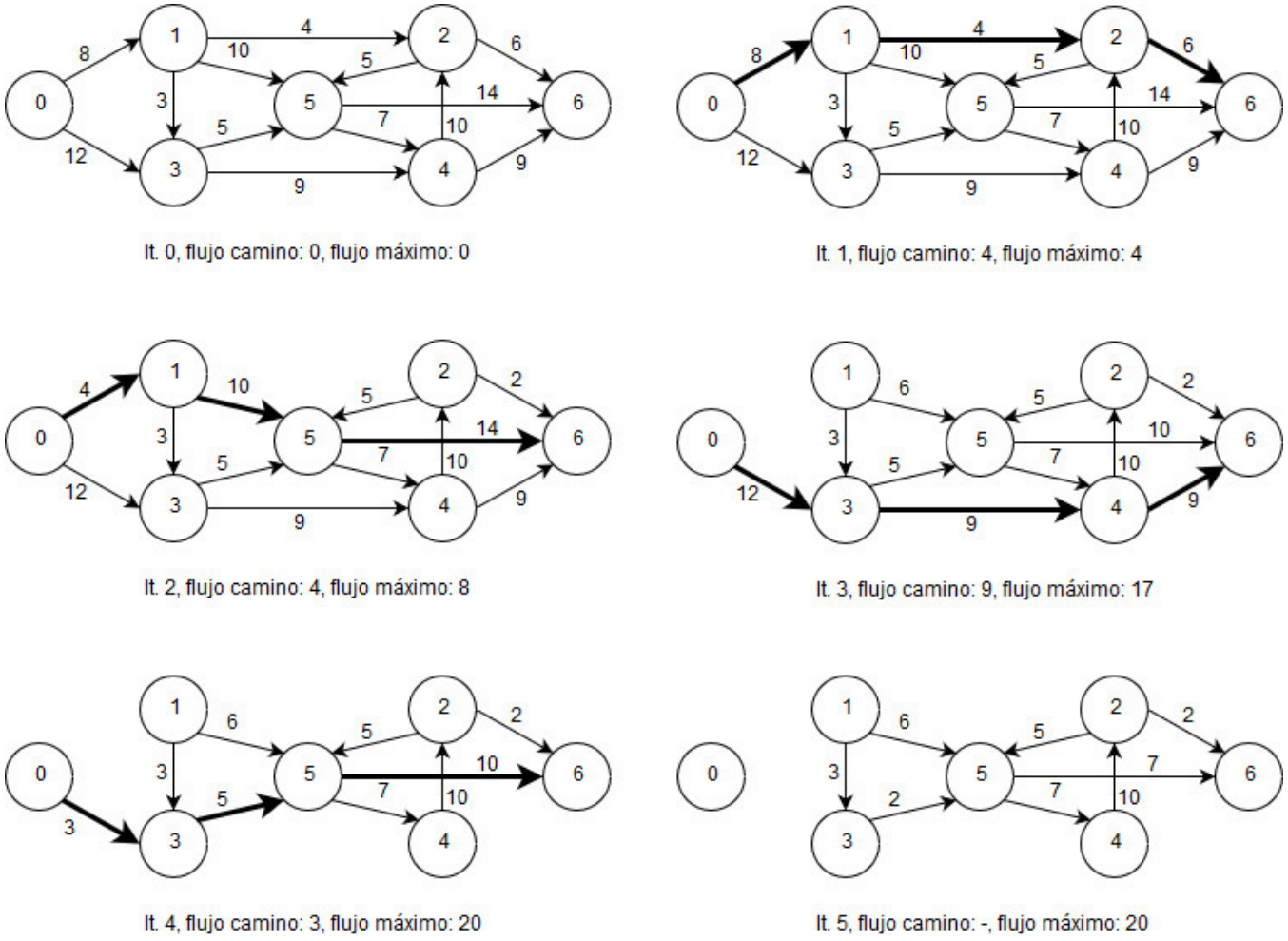


FIGURA 5.32: Funcionamiento del algoritmo de Ford-Fulkerson.

6 ÁRBOLES

Los árboles corresponden a un tipo especial de grafo, denominados grafos acíclicos. Son estructuras jerárquicas para organizar colecciones de elementos. Tienen diversas aplicaciones, por ejemplo: analizar circuitos eléctricos, representar la estructura de fórmulas matemáticas, representar la estructura sintáctica de programas en compiladores, etc.

Existen diferentes variantes para la estructura de árboles. En el presente documento solo se presentan las definiciones básicas y las operaciones más comunes.

6.1 CONCEPTOS BÁSICOS

Un árbol es una colección de elementos llamados nodos, que tiene una estructura jerárquica y donde un nodo se distingue como raíz. Un árbol puede definirse recursivamente como:

1. Un nodo único n es un árbol, donde n es también la raíz.
2. Dado un nodo n y los árboles $T_1, T_2, \dots, T_k$ con raíces $n_1, n_2, \dots, n_k$ respectivamente, se puede construir un nuevo árbol haciendo que n sea el padre de $n_1, n_2, \dots, n_k$. La raíz del árbol resultante es n, y se dice que $T_1, T_2, \dots, T_k$ son subárboles de la raíz. Adicionalmente, los nodos $n_1, n_2, \dots, n_k$ se denominan hijos del nodo n.
3. En ocasiones resulta útil la definición del árbol nulo, es decir, el árbol que no contiene nodos.

Existen algunas definiciones elementales que resultan de gran utilidad en el estudio de los árboles, las cuales se listan a continuación.

Camino: se define como camino desde n_1 hasta n_k a una secuencia de nodos $n_1, n_2, \dots, n_k$ tal que n_i es padre de n_{i+1}, $1 \leq i < k$. Además:

- La longitud del camino está dada por la cantidad de nodos de la secuencia.
- Si existe un camino desde n_i hasta n_j, n_i es ancestro de n_j y n_j es descendiente de n_i.

Hoja: es un nodo sin descendientes.

Subárbol: es un nodo más todos sus descendientes.

Altura de un nodo: corresponde a la longitud del camino más largo desde el nodo hasta una hoja.

Altura de un árbol: está dada por la longitud del camino más largo desde la raíz hasta una hoja.

Profundidad de un nodo: corresponde al largo del camino único desde la raíz del árbol hasta el nodo.

EJEMPLO 6.1:

La figura 6.1 muestra un árbol de raíz A y hojas C, G, H e I. Se puede decir que G es descendiente de D, pues existe el camino $D - F - G$. También se puede decir que B, D e I (junto a sus respectivos descendientes) son subárboles de A. La altura de D es 3, mientras que la altura del árbol es 4. La profundidad de D es 2.

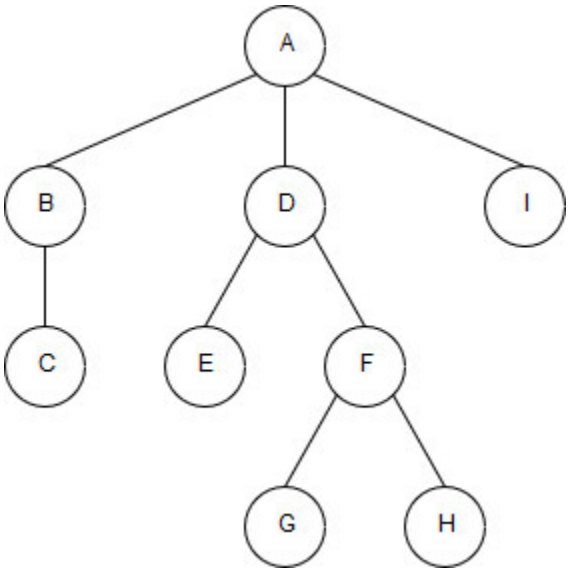


FIGURA 6.1: Un árbol.

6.2 EL TDA ÁRBOL

Existe una gran variedad de operaciones que se pueden realizar sobre árboles. Se presentan a continuación las principales. Cabe destacar que no se incluye una operación de inserción ni de borrado, puesto que el criterio para incorporar (o quitar) un nodo depende del tipo de árbol a construir.

1. `PADRE(T: árbol, n: nodo): nodo.`
   Devuelve el padre del nodo n en el árbol T. Si n es la raíz (que no tiene padre), devuelve nulo.

2. `HIJO_IZQUERDO(T: árbol, n: nodo) : nodo.`
   Devuelve el hijo situado más a la izquierda para el nodo n en el árbol T. Si n es una hoja (no tiene hijos), devuelve nulo.

3. `HERMANO_DERECHO(T: árbol, n: nodo) : nodo.`
   Devuelve el hermano derecho del nodo n en el árbol T, que se define como el nodo m que tiene el mismo padre p que n tal que m está situado inmediatamente a la derecha de n según el ordenamiento de los hijos de p. Devuelve nulo si n no tiene un hermano a su derecha.

4. `CONTENIDO(T: árbol, n: nodo): tipoDato.`
   Entrega el contenido del nodo n en el árbol T.

5. `CREAR_I(dato: tipoDato, T1: árbol, T2: árbol, ..., Ti: árbol): árbol.`
   Es una de una familia infinita de funciones, una para cada valor de $i = 0,1,2,$ .... Esta función crea un nuevo nodo r con contenido v e i hijos, correspondientes a las raíces de los árboles $T1, T2, \ldots, Ti$ ordenadas de izquierda a derecha. Devuelve el árbol con raíz r. Cabe señalar que si $i = 0$, entonces r es la raíz y una hoja a la vez.

6. `RAÍZ(T: árbol): nodo`.
   Entrega el nodo raíz del árbol T. Si T es el árbol nulo, devuelve nulo.

7. `ANULAR(T: árbol): árbol`.
   Hace que T sea el árbol nulo.

6.3 IMPLEMENTACIÓN DEL TDA ÁRBOL

Al igual que ocurre con las listas, existen diferentes maneras de implementar el TDA árbol. A continuación, se introducen las principales.

6.3.1 IMPLEMENTACIÓN MEDIANTE ARREGLOS

Sea T un árbol cuyos nodos están indizados en el rango $[0, n-1]$. Una representación sencilla de T que soporta la operación `PADRE` es un arreglo lineal A donde la estructura correspondiente a cada elemento $A[i]$ contiene un puntero o un cursor al padre del nodo i. Como la raíz no tiene padre, su puntero es nulo. Esta representación hace uso de la propiedad que establece que cada nodo tiene un único padre y permite que el padre de un nodo pueda ser encontrado en tiempo constante. Un camino ascendente, es decir, desde un nodo hacia su padre, hacia el padre de este último, etc., puede ser recorrido en tiempo proporcional a la cantidad de nodos del camino.

La representación mediante un nodo que apunte al padre hace que las operaciones que requieren información de los hijos sean más complejas. Dado un nodo n, la determinación de los hijos de n o de la altura de n resultan ser operaciones costosas en términos de tiempo. Otra dificultad es que la representación en cuestión tampoco especifica el orden en que se encuentran los hijos de un nodo, por lo que se hace necesario imponer un orden artificial para poder implementar operaciones como `HIJO_IZQUIERDO` y `HERMANO_DERECHO`. Una idea es numerar los hijos de un nodo después de numerar al padre, y a su vez asignar números a los hijos en orden creciente de izquierda a derecha.

El algoritmo 6.1 muestra un posible algoritmo para esta implementación, asumiendo que los hijos de un vértice siempre aparecen ordenados de izquierda a derecha. Para la familia de funciones `CREAR_I()`, se muestran a modo de ejemplo `CREAR_0()` y `CREAR_2()`.

ALGORITMO 6.1: Implementación de un árbol mediante arreglos.

```
tipo nodo:
      dato: tipoDato
      padre: entero

tipo árbol:
      tamaño: entero
      arreglo: nodo[]
```

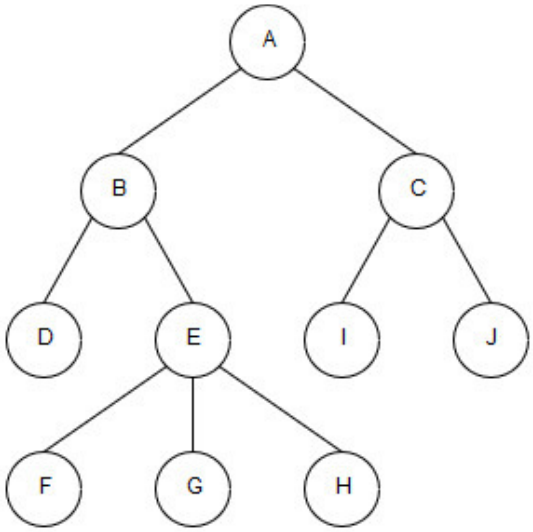
ALGORITMO 6.1 (continuación): Implementación de un árbol mediante arreglos.

```
INICIALIZAR(tamaño: entero): árbol
      árbol T
      T.arreglo = nodo[tamaño]
      T.tamaño = 0
      Devolver T

PADRE(T: árbol, n: entero): entero
      Devolver T.arreglo[n].padre

HIJO_IZQUERDO(árbol T, int n): entero
    Para i desde 0 hasta T.tamaño - 1:
            Si T.arreglo[i].padre == n:
                  Devolver i

    Devolver NULO

HERMANO_DERECHO(T: árbol, n: entero): entero
    Para i desde 0 hasta T.tamaño - 1:
            Si T.arreglo[i].padre == T.arreglo[n].padre:
                  Devolver i

      Devolver NULO

CONTENIDO(T: árbol, n: entero): entero
      Devolver T.arreglo[n].dato

RAÍZ(T: árbol): entero
      Para i = 0 i < T.tamaño i++:
            Si T.arreglo[i].padre == NULO:
                  Devolver i

      Devolver NULO

ANULAR(T: árbol): árbol
      T.tamaño = 0
      eliminar(T.arreglo)
      Devolver T

CREAR_0(dato: tipoDato): árbol
      árbol T = INICIALIZAR(1)
      T.arreglo[0].dato = dato
      T.arreglo[0].padre = NULO
      T.tamaño = 1
      Devolver T
```

ALGORITMO 6.1 (continuación): Implementación de un árbol mediante arreglos.

```
CREAR_2(dato: tipoDato, T1: árbol, T2: árbol): árbol
      árbol T = INICIALIZAR(1 + T1.tamaño + T2.tamaño)
      T.arreglo[0].dato = dato
      T.arreglo[0].padre = NULO
      T.tamaño = 1 + T1.tamaño + T2.tamaño
      j = 1

      Para i desde 0 hasta T1.tamaño - 1:
            T.arreglo[j] = T1.arreglo[i]
            T.arreglo[j].padre = T.arreglo[j].padre + 1
            j = j + 1

      Para i desde 0 hasta T2.tamaño - 1:
            T.arreglo[j] = T2.arreglo[i]

            Si T2.arreglo[i].padre == NULO:
                  T.arreglo[j].padre = 0
            Sino:
                  T.arreglo[j].padre = T.arreglo[j].padre + T1.tamaño + 1

            j = j + 1

      Devolver T
```

La figura 6.2 muestra un árbol implementado mediante un arreglo.



T.tamaño = 10

T.a:

A	B	C	D	E	F	G	H	I	J
-1	0	0	1	1	4	4	4	2	2
(0)	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)

FIGURA 6.2: Árbol implementado mediante un arreglo con cursores.

6.3.2 IMPLEMENTACIÓN MEDIANTE LISTAS DE HIJOS

Una implementación bastante utilizada para los árboles es, para cada nodo, crear una lista con sus hijos. Si bien para estas listas se puede usar cualquiera de las implementaciones estudiadas anteriormente, se suelen usar listas enlazadas debido a que los nodos pueden tener distintas cantidades de hijos. La figura 6.3 muestra una representación del árbol de la figura 6.2 mediante listas de hijos. Se puede observar que existe un arreglo de cabeceras indexado por los nodos. Cada cabecera[i] apunta a una lista de nodos correspondiente a los hijos del nodo i. Un punto denota el puntero nulo.

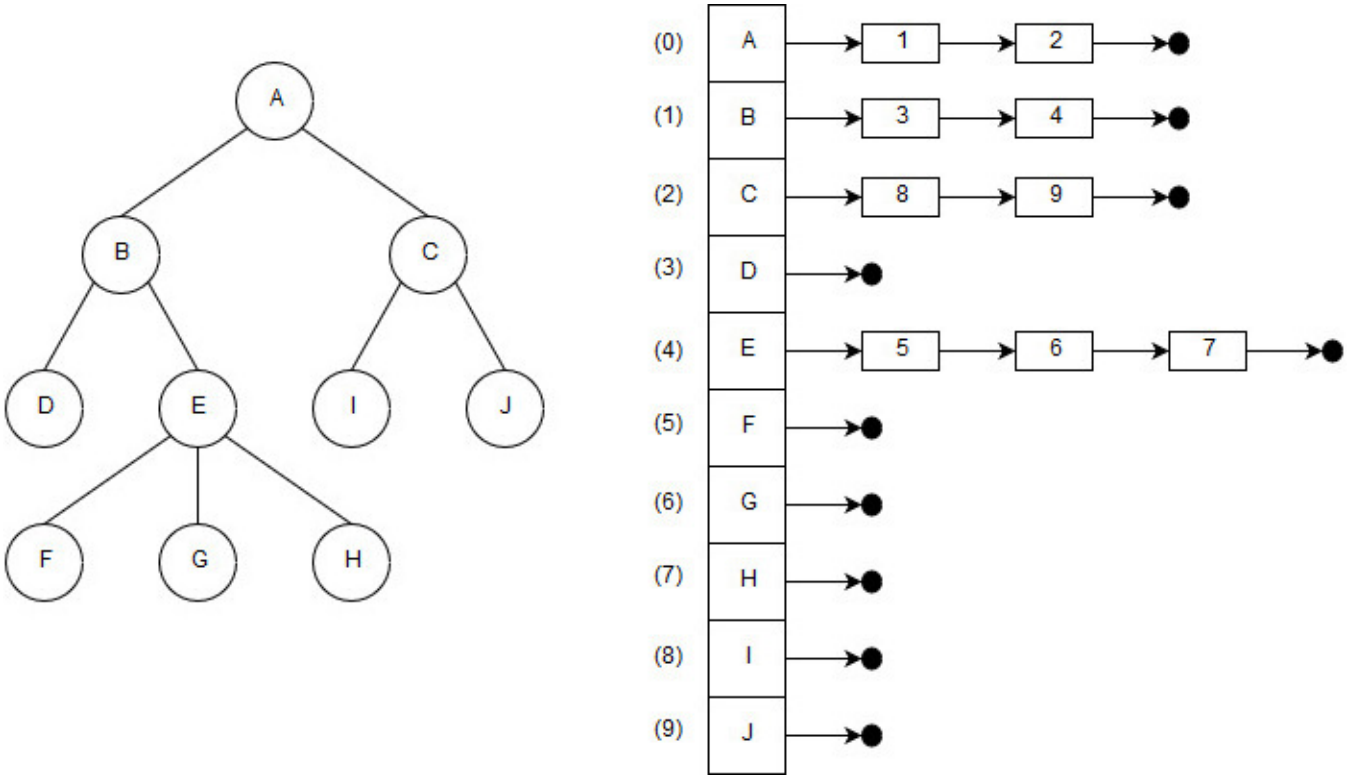


FIGURA 6.3: Árbol implementado mediante listas de hijos.

El algoritmo 6.2 muestra un posible algoritmo para esta implementación, asumiendo que los hijos de un vértice siempre aparecen ordenados de izquierda a derecha y que la raíz del árbol siempre es el primer elemento del arreglo de nodos. Para la familia de funciones `CREAR_I()`, se muestran a modo de ejemplo `CREAR_0()` y `CREAR_2()`.

ALGORITMO 6.2: Implementación de un árbol mediante listas de hijos.

```
tipo nodo:
      dato: tipoDato
      hijos: lista

tipo árbol:
      tamaño: entero
      arreglo: nodo[]
```

ALGORITMO 6.2 (continuación): Implementación de un árbol mediante listas de hijos.

```
CREAR_0(dato: tipoDato): árbol
      árbol T
      T.arreglo = nodo[1]
      T.arreglo[0].dato = dato
      T.arreglo[0].hijos = NULO
      T.tamaño = 1
      Devolver T

CREAR_2(dato: tipoDato, T1: árbol, T2: árbol): árbol
      tamaño = 1 + T1.tamaño + T2.tamaño
      T.arreglo = nodo[tamaño]
      T.tamaño = 0
      T.arreglo[0].dato = dato
      j = 1

      Para i desde 0 hasta T1.tamaño - 1:
            T.arreglo[j].dato = T1.arreglo[i].dato

            Para k desde 0 hasta LARGO(T1.arreglo[i].hijos) - 1:
                  listaHijos = T.arreglo[j].hijos
                  posicion = LARGO(listaHijos)
                  hijo = OBTENER(T1.arreglo[i].hijos, k) + 1
                  listaHijos = INSERTAR(listaHijos, posicion, hijo)

            j = j + 1

      Para i desde 0 hasta T2.tamaño - 1:
            T.arreglo[j].dato = T2.arreglo[i].dato

            Para k desde 0 hasta LARGO(T2.arreglo[i].hijos) - 1:
                  listaHijos = T.arreglo[j].hijos
                  posicion = LARGO(listaHijos)
                  hijo = OBTENER(T2.arreglo[i].hijos, k) + 1 + T1.tamaño
                  listaHijos = INSERTAR(listaHijos, posicion, hijo)

            j = j + 1

      T.tamaño = tamaño
      listaHijos = T.arreglo[0].hijos
      posicion = LARGO(listaHijos)
      hijo = 1
      listaHijos = INSERTAR(listaHijos, posicion, hijo)
      listaHijos = T.arreglo[0].hijos
      posicion = LARGO(listaHijos)
      hijo = 1 + T1.tamaño
      listaHijos = INSERTAR(listaHijos, posicion, hijo)
      Devolver T
```

ALGORITMO 6.2 (continuación): Implementación de un árbol mediante listas de hijos.

```
ANULAR(T: árbol): árbol
      Para i desde 0 hasta T.tamaño - 1:
            T.arreglo[i].hijos = ANULAR(T.arreglo[i].hijos)

      eliminar(T.arreglo)
      T.tamaño = 0
      Devolver T

RAÍZ(T: árbol): entero
      Si T.tamaño == 0:
            Devolver - 1

      Devolver 0

CONTENIDO(T: árbol, n: entero): tipoDato
      Si n <= 0 ó n >= T.tamaño:
            Devolver NULO

      Devolver T.arreglo[n].dato

PADRE(T: árbol, n: entero): entero
      Si n <= 0 ó n >= T.tamaño:
            Devolver NULO

      Para i desde 0 hasta T.tamaño - 1:
            Para j desde 0 hasta LARGO(T.arreglo[i].hijos) - 1:
                  Si OBTENER(T.arreglo[i].hijos, j) == n:
                        Devolver i

    Devolver NULO

HIJO_IZQUERDO(T: árbol, n: entero): entero
      Si n < 0 ó n >= T.tamaño:
            Devolver NULO

      Si LARGO(T.arreglo[n].hijos) == 0:
            Devolver NULO

      Devolver OBTENER(T.arreglo[n].hijos, 0)

HERMANO_DERECHO(T: árbol, n: entero): entero
      Si n <= 0 ó n >= T.tamaño:
            Devolver NULO

      Para i desde 0 hasta T.tamaño - 1:
            Para j desde 0 hasta LARGO(T.arreglo[i].hijos) - 1:
                  Si OBTENER(T.arreglo[i].hijos, j) == n:
                        Devolver OBTENER(T.arreglo[i].hijos, j + 1)

      Devolver NULO
```

6.3.3 IMPLEMENTACIÓN HIJO IZQUIRDO, HERMANO DERECHO

La implementación más utilizada, que también requiere de punteros, pero no considera el uso de ningún tipo de arreglo y, por ende, facilita la creación de nuevos árboles, es la de hijo izquierdo, hermano derecho. En ella, cada nodo es una estructura conformada por un campo para el dato, otro para un puntero al hijo izquierdo y otro con un puntero al hermano derecho. La figura 6.4 muestra un árbol y su representación mediante este esquema.

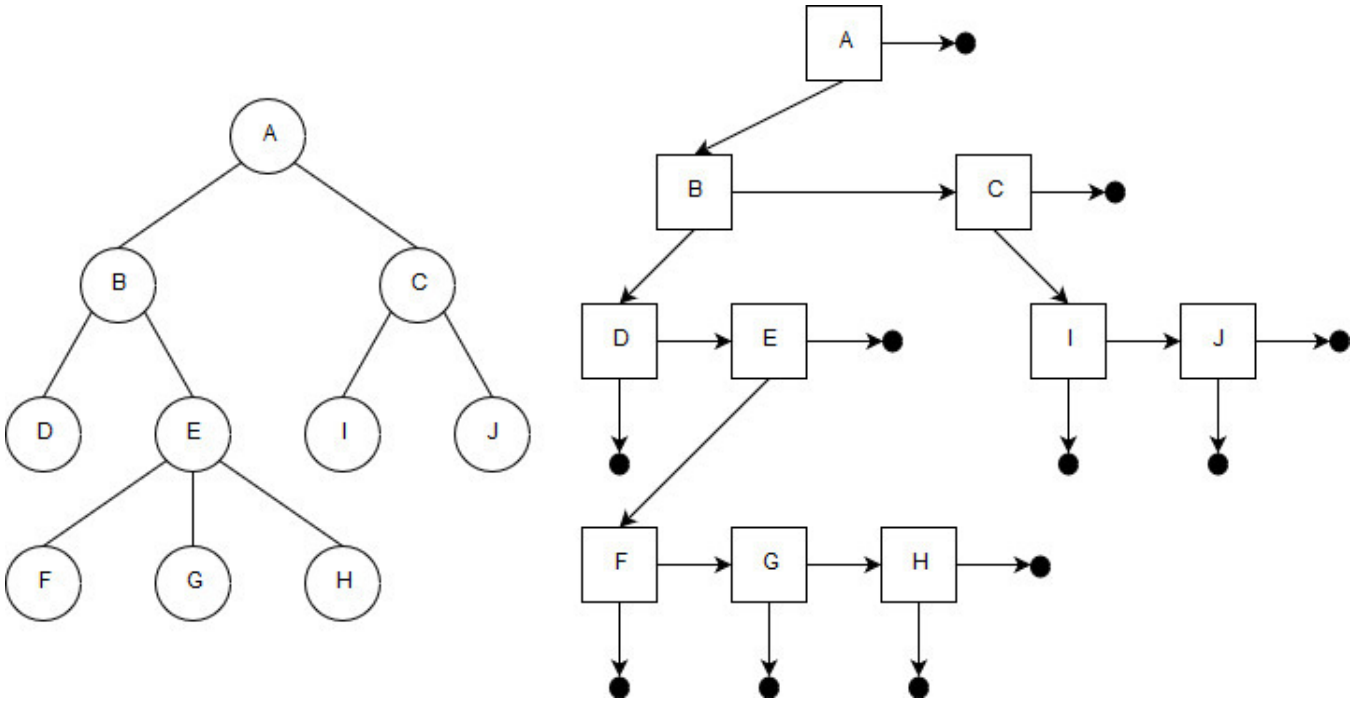


FIGURA 6.4: Árbol según implementación hijo izquierdo, hermano derecho.

El algoritmo 6.3 muestra la definición de las operaciones del TDA árbol con la implementación hijo izquierdo, hermano derecho.

ALGORITMO 6.3: Implementación hijo izquierdo, hermano derecho de un árbol.

```
tipo nodo:
      dato: tipoDato
      hijoIzquierdo: nodo
      hermanoDerecho: nodo

tipo árbol = tipo nodo
```

ALGORITMO 6.3 (continuación): Implementación hijo izquierdo, hermano derecho de un árbol.

```
CREAR_NODO(dato: tipoDato): nodo
      nodo nuevo

      Si nuevo != NULO:
            nuevo.dato = dato
            nuevo.hijoIzquierdo = NULO
            nuevo.hermanoDerecho = NULO

      Devolver nuevo

PADRE(T: árbol, n: nodo): nodo
      Si n == T ó T == NULO:
            Devolver NULO

      indice = T.hijoIzquierdo

      Mientras indice != NULO:
            Si indice == n:
                  Devolver T

            padre = PADRE(indice, n)

            Si padre != NULO:
                  Devolver padre

            indice = indice.hermanoDerecho

      Devolver NULO

HIJO_IZQUERDO(T: árbol, n: nodo): nodo
      Si n == NULO:
            Devolver NULO

      Devolver n.hijoIzquierdo

HERMANO_DERECHO(T: árbol, n: nodo): nodo
      Si n == NULO:
            Devolver NULO

      Devolver n.hermanoDerecho

CONTENIDO(T: árbol, n: nodo): tipoDato
      Si n == NULO:
            Devolver NULO

      Devolver n.dato

CREAR_0(dato: tipoDato): árbol
      nodo nuevo = CREAR_NODO(dato)
      Devolver nuevo
```

ALGORITMO 6.3 (continuación): Implementación hijo izquierdo, hermano derecho de un árbol.

```
CREAR_2(dato: tipoDato, T1: árbol, T2: árbol): árbol
      árbol T = CREAR_NODO_árbol(dato)
      T1.hermanoDerecho = T2
      T.hijoIzquierdo = T1
      Devolver T

RAÍZ(T: árbol): nodo
      Devolver T

ANULAR(T: árbol): árbol
      Si T == NULO:
            Devolver T

      T.hijoIzquierdo = ANULAR(T.hijoIzquierdo)
      T.hermanoDerecho = ANULAR(T.hermanoDerecho)
      eliminar(T)
      Devolver NULO
```

6.4 RECORRIDO Y ORDEN DE UN ÁRBOL

Los árboles tienen asociada la idea de orden. La regla general es: si a y b son hijos de un nodo n y a está a la izquierda de b, entonces todos los descendientes de a están a la izquierda de todos los descendientes de b.

Existen diversas maneras de recorrer ordenadamente todos los nodos de un árbol. Los principales son preorden, inorden y postorden. Estos últimos pueden definirse recursivamente de la siguiente manera (el algoritmo 6.4 los explica de manera más detallada):

1. Si el árbol T es nulo, entonces la lista vacía es el preorden, inorden y postorden de T.

2. Si el árbol T tiene un único nodo n, entonces la lista que solo contiene el elemento n es el preorden, inorden y postorden de T.

3. Dado el árbol T de raíz n con subárboles $T_1, T_2, \ldots T_k$:
   a. *Preorden:* raíz(T) preorden(T_1) preorden(T_2) … preorden(T_k).
   b. *Inorden:* inorden(T_1) raíz(T) inorden(T_2) … inorden(T_k).
   c. *Postorden:* postorden(T_1) postorden(T_2) … postorden(T_k) raíz(T).

Los recorridos anteriores pueden ser útiles para obtener información jerárquica del árbol. Sean las funciones $postorden(n)$, que entrega la posición de un nodo n en una lista en postorden de los nodos del árbol, y $desc(n)$, que entrega la cantidad de descendientes propios del nodo n (descendientes que no incluyen al propio n).

Las posiciones asignadas en la lista postorden a los nodos del árbol cumplen la propiedad de que todos los nodos en un subárbol de raíz n están situados en ubicaciones consecutivas que van desde

$postorden(n) - desc(n)$ hasta $postorden(n)$. Así, para ver si un nodo x desciende del nodo y, solo basta con comprobar si $postorden(y) - desc(y) \leq postorden(x) \leq postorden(y)$.

Existe una propiedad similar para preorden.

ALGORITMO 6.4: Recorridos de un árbol.

```
INORDEN(T: árbol, l: lista): lista
      nodo indice

      Si T != NULO:
            indice = HIJO_IZQUIERDO(T)

            Si indice != NULO:
                  l = INORDEN(indice, l)
                  indice = HERMANO_DERECHO(indice)

            l = INSERTAR(l, LARGO(l), CONTENIDO(T, RAÍZ(T)))

            Mientras indice != NULO:
                  l = INORDEN(indice, l)
                  indice = HERMANO_DERECHO(indice)

      Devolver l

PREORDEN(T: árbol, l: lista): lista
      nodo indice

      Si T != NULO:
            l = INSERTAR(l, LARGO(l), CONTENIDO(T, RAÍZ(T)))
            indice = HIJO_IZQUIERDO(T)

            Mientras indice != NULO:
                  l = INORDEN(indice, l)
                  indice = HERMANO_DERECHO(indice)

      Devolver l

POSTORDEN(T: árbol, l: lista): lista
      nodo indice

      Si T != NULO:
            indice = HIJO_IZQUIERDO(T)

            Mientras indice != NULO:
                  l = INORDEN(indice, l)
                  indice = HERMANO_DERECHO(indice)

            l = INSERTAR(l, LARGO(l), CONTENIDO(T, RAÍZ(T)))

      Devolver l
```

EJEMPLO 6.1:

Considere el árbol de la figura 6.5.

- Recorrido preorden: A B C D E F G H I J K
- Recorrido inorden: C B D A F E G I H J K
- Recorrido postorden: C D B F G E I J K H A

Considerando el recorrido postorden, los valores para $postorden(n)$ y $desc(n)$ se muestran en la tabla 6.1.

TABLA 6.1: Tabla de valores para el ejemplo 6.1.

n	C	D	B	F	G	E	I	J	K	H	A
$postorden(n)$	0	1	2	3	4	5	6	7	8	9	10
$desc(n)$	0	0	2	0	0	2	0	0	0	3	10

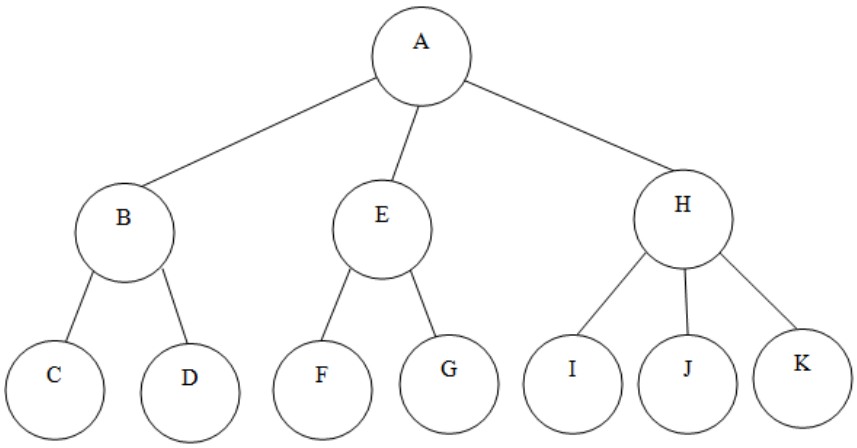


FIGURA 6.5: Árbol para el ejemplo 6.1.

¿J es descendiente de H?
$postorden(H) - desc(H) \leq postorden(J) \leq postorden(H)$
$9 - 3 \leq 7 \leq 9$
$6 \leq 7 \leq 9$ se cumple, por lo que J es descendiente de H.

¿D es descendiente de E?
$postorden(E) - desc(E) \leq postorden(D) \leq postorden(E)$
$5 - 2 \leq 1 \leq 5$
$3 \leq 1 \leq 5$ no se cumple, por lo que D no es descendiente de E.

6.5 EL TDA ÁRBOL BINARIO

Los árboles binarios son árboles cuyos nodos tienen a lo más dos hijos. No obstante, difieren de los árboles generales ya estudiados en que se debe especificar si un hijo de un nodo es el hijo izquierdo o el hijo

derecho. Por ejemplo, los árboles de la figura 6.6 son diferentes porque en el árbol de la izquierda F es el hijo izquierdo de C, mientras que en el de la derecha, F es el hijo derecho de C. Esta variante del TDA árbol es ampliamente utilizada, por lo que merece ser estudiada de manera independiente.

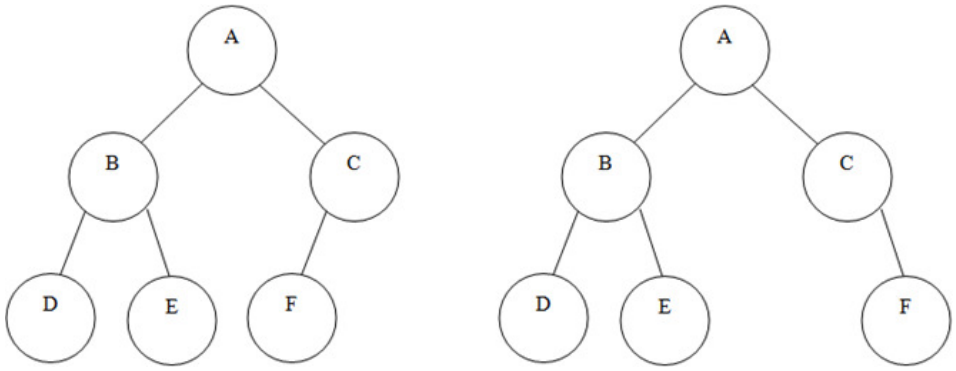


FIGURA 6.6: Dos árboles binarios.

Para el TDA árbol binario se suele trabajar con un conjunto ligeramente diferente de operaciones. Se reemplaza `HERMANO_DERECHO()` por `HIJO_DERECHO()`, y la familia de funciones `CREAR_I()` es reemplazada por una nueva función `CREAR()`. El resto de las operaciones permanece sin cambios.

1. `HIJO_DERECHO(T: árbol, n: árbol): árbol.`
   Devuelve el hijo derecho del nodo n en el árbol T. Si n es una hoja, devuelve nulo.

2. `CREAR(dato: tipoDato, T1: árbol, T2: árbol): árbol.`
   Crea un nuevo nodo r con contenido v, que es la raíz del nuevo árbol, y le asigna como hijo izquierdo la raíz de $T1$ y como hijo derecho, la raíz de $T2$. Para crear una hoja, $T1$ y $T2$ son nulos.

Los recorridos estudiados para árboles generales sufren una leve modificación debido a la modificación de la estructura de datos subyacente, como muestra el algoritmo 6.5.

6.6 IMPLEMENTACIÓN DE ÁRBOLES BINARIOS

6.6.1 IMPLEMENTACIÓN MEDIANTE ARREGLOS

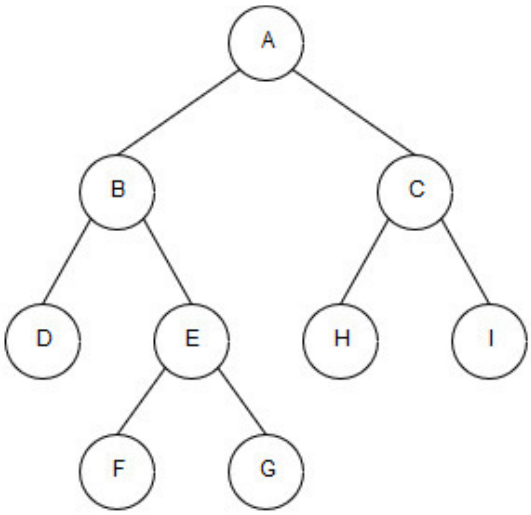
Los árboles binarios pueden ser implementados mediante arreglos haciendo que cada elemento del arreglo sea una estructura con un campo para el dato contenido por el nodo y dos campos enteros para los hijos del nodo, donde los campos de los hijos contienen la posición en el arreglo del hijo izquierdo y el hijo derecho, respectivamente. La figura 6.7 muestra un árbol binario y el arreglo correspondiente. El algoritmo 6.6 nuestra una forma de lograr esta implementación.

ALGORITMO 6.5: Recorridos de un árbol binario.

```
INORDEN(T: árbol, l: lista): lista
      Si T != NULO:
            l = INORDEN(HIJO_IZQUIERDO(T), l)
            l = INSERTAR(l, LARGO(l), CONTENIDO(T, RAÍZ(T)))
            l = INORDEN(HIJO_DERECHO(T), l)

      Devolver l

PREORDEN(T: árbol, l: lista): lista
      Si T != NULO:
            l = INSERTAR(l, LARGO(l), CONTENIDO(T, RAÍZ(T)))
            l = PREORDEN(HIJO_IZQUIERDO(T), l)
            l = PREORDEN(HIJO_DERECHO(T), l)

      Devolver l

POSTORDEN(T: árbol, l: lista): lista
      Si T != NULO:
            l = POSTORDEN(HIJO_IZQUIERDO(T), l)
            l = POSTORDEN(HIJO_DERECHO(T), l)
            l = INSERTAR(l, LARGO(l), CONTENIDO(T, RAÍZ(T)))

      Devolver l
```

ALGORITMO 6.6: Implementación mediante arreglos de un árbol binario.

```
tipo nodo:
      dato: entero
      hijoIzquierdo: entero
      hijoDerecho: entero

tipo árbol:
      nodo arreglo[]
      entero tamaño

CONTENIDO(T: árbol, n: entero): tipoDato
      Si n < 0 ó n >= T.tamaño:
            Devolver NULO

      Devolver T.arreglo[n].dato

HIJO_IZQUERDO(T: árbol, n: entero): entero
      Si n < 0 ó n >= T.tamaño:
            Devolver NULO

      Devolver T.arreglo[n].hijoIzquierdo

HIJO_DERECHO(T: árbol, n: entero): entero
      Si n < 0 ó n >= T.tamaño:
            Devolver NULO

      Devolver T.arreglo[n].hijoDerecho
```



Tipo nodo:

dato	
h. izq.	h. der.

Arreglo:

(0)		(1)		(2)		(3)		(4)		(5)		(6)		(7)		(8)	
A		B		C		D		E		F		G		H		I	
1	2	3	4	7	8	NULO	NULO	5	6	NULO	NULO	NULO	NULO	NULO	NULO	NULO	NULO

FIGURA 6.7: Un arreglo que implementa un árbol binario.

ALGORITMO 6.6 (continuación): Implementación mediante arreglos de un árbol binario.

```
PADRE(T: árbol, n: entero): entero
      Para i desde 0 hasta T.tamaño - 1:
            Si HIJO_IZQUERDO(T, i) == n ó HIJO_DERECHO(T, i) == n:
                  Devolver i

      Devolver NULO

RAÍZ(T: árbol): entero
      Si T.tamaño == 0:
            Devolver NULO

      Devolver 0

ANULAR(T: árbol): árbol
      T.tamaño = 0
      Devolver T
```

ALGORITMO 6.6 (continuación): Implementación mediante arreglos de un árbol binario.

```
CREAR(dato: tipoDato, T1: árbol, T2: árbol): árbol
      árbol T
      entero tamaño = 1 + T1.tamaño + T2.tamaño

      T.arreglo = nodo[tamaño]
      T.tamaño = tamaño
      j = 1

      Para i desde 0 hasta T1.tamaño – 1:
            T.arreglo[j].dato = T1.arreglo[i].dato
            T.arreglo[j].hijoIzquierdo = HIJO_IZQUERDO(T1, i) + 1
            T.arreglo[j].hijoDerecho = HIJO_DERECHO(T1, i) + 1
            j = j + 1

      Para i desde 0 hasta T2.tamaño – 1:
            T.arreglo[j].dato = T2.arreglo[i].dato
            T.arreglo[j].hijoIzquierdo = HIJO_IZQUERDO(T2, i) + 1 + T1.tamaño
            T.arreglo[j].hijoDerecho = HIJO_DERECHO(T2, i) + 1 + T1.tamaño
            j = j + 1

      T.arreglo[0].dato = dato

      Si T1 == NULO:
            T.arreglo[0].hijoIzquierdo = NULO
      Sino:
            T.arreglo[0].hijoIzquierdo = RAÍZ(T1) + 1

      Si T2 == NULO:
            T.arreglo[0].hijoDerecho = NULO
      Sino:
            T.arreglo[0].hijoDerecho = RAÍZ(T2) + 1 + T1.tamaño

      Devolver T
```

6.6.2 IMPLEMENTACIÓN MEDIANTE PUNTEROS

La implementación mediante punteros de un árbol binario es conceptualmente similar a la implementación mediante arreglos, pero ahora se requieren punteros hacia los hijos izquierdo y derecho, respectivamente, como muestra el algoritmo 6.7.

ALGORITMO 6.7: Implementación mediante punteros de un árbol binario.

```
tipo árbol:
      dato: tipoDato
      hijoIzquierdo: árbol
      hijoDerecho: árbol
```

ALGORITMO 6.7 (continuación): Implementación mediante punteros de un árbol binario.

```
CONTENIDO(T: árbol): tipoDato
      Si T == NULO:
            Devolver NULO

      Devolver T.dato

HIJO_IZQUERDO(T: árbol): árbol
      Si T == NULO:
            Devolver NULO

      Devolver T.hijoIzquierdo

HIJO_DERECHO(T: árbol): árbol
      Si T == NULO:
            Devolver NULO

      Devolver T.hijoDerecho

PADRE(T: árbol, n: árbol): árbol
      árbol padre

      Si T == NULO:
            Devolver NULO

      Si HIJO_IZQUERDO(T, T) == n ó HIJO_DERECHO(T, T) == n:
            Devolver T

      padre = PADRE(HIJO_IZQUERDO(T, T), n)

      Si padre != NULO:
            Devolver padre

      Devolver PADRE(HIJO_DERECHO(T, T), n)

CREAR(dato: tipoDato, T1: árbol, T2: árbol): árbol
      árbol T
      T.dato = dato
      T.hijoIzquierdo = T1
      T.hijoDerecho = T2
      Devolver T

RAÍZ(T: árbol): árbol
      Devolver T

ANULAR(T: árbol): árbol
      Si T == NULO:
            Devolver T

      T.hijoIzquierdo = ANULAR(HIJO_IZQUERDO(T, T))
      T.hijoDerecho = ANULAR(HIJO_DERECHO(T, T))
      eliminar(T)
      T = NULO
      Devolver T
```

6.7 ÁRBOLES BINARIOS ORDENADOS (ABO)

También llamados árboles de búsqueda binaria, son aquellos árboles binarios en que los nodos son almacenados ordenadamente de forma tal que cumplen con la siguiente propiedad: sean x e y dos nodos de un árbol binario. Si y está en el subárbol izquierdo de x, entonces $y \leq x$. Si y está en el subárbol derecho de x, entonces $y \geq x$. Esta característica además permite mostrar los nodos de manera ordenada mediante el recorrido inorden.

La noción de orden presentada anteriormente permite incorporar a los árboles de búsqueda binaria algunas operaciones adicionales a las de un árbol binario cualquiera, relacionadas con la búsqueda de un nodo, para las cuales existen algoritmos con complejidad $O(h)$, donde h corresponde a la altura del árbol. También establece criterios para la inserción y la eliminación de nodos. Las operaciones propias de un ABO, detalladas en el algoritmo 6.8 para una implementación mediante punteros, son:

1. `BUSCAR(T: árbol, dato: tipoDato): árbol`.
   Entrega la posición de `dato` si éste se encuentra en el árbol y nulo en caso contrario. Esta operación puede efectuarse tanto iterativa como recursivamente. En general, la versión iterativa es más eficiente.

2. `MÍNIMO(T: árbol): árbol`.
   Entrega la posición del menor elemento de un ABO no vacío. Devuelve nulo si el árbol está vacío. Esta operación resulta bastante sencilla, porque los elementos más pequeños están situados siempre a la izquierda.

3. `MÁXIMO(T: árbol): árbol`.
   Entrega el mayor elemento de un ABO no vacío. Devuelve nulo si el árbol está vacío. Resulta igual de sencilla que la operación anterior, puesto que los elementos más grandes están situados siempre a la derecha.

4. `SIGUIENTE(árbol: nodo, dato: tipoDato): árbol`.
   Entrega el menor elemento que sea mayor o igual que `dato` si éste existe, y nulo en caso contrario. Si el subárbol derecho no es nulo, entonces el siguiente de `dato` es el elemento más pequeño en el subárbol derecho de `dato`. Por otra parte, si el subárbol derecho está vacío, pero existe un siguiente de `dato`, éste corresponde al menor ancestro de `dato` cuyo hijo izquierdo sea también ancestro de `dato`.

5. `ANTERIOR(árbol: nodo, dato: tipoDato): árbol`.
   Entrega el mayor elemento que sea menor o igual que `dato` si éste existe, y nulo en caso contrario. Si el subárbol izquierdo no es nulo, entonces el anterior de `dato` es el elemento más grande en el subárbol izquierdo de `dato`. Por otra parte, si el subárbol izquierdo está vacío, pero existe un anterior de `dato`, éste corresponde al mayor ancestro de `dato` cuyo hijo derecho sea también ancestro de `dato`.

6. `INSERTAR(árbol: nodo, dato: tipoDato): árbol`.
   Toma un nuevo nodo x de forma tal que ambos hijos de x son nulos. Luego modifica el árbol y los hijos de x a fin de que x quede en una posición del árbol en que se mantenga la condición de orden del árbol.

7. `BORRAR(árbol: nodo, dato: tipoDato): árbol.`
   Para quitar un nodo x de un árbol se tienen los siguientes casos básicos (para los otros casos se pueden usar llamadas recursivas):
   a. Si x es la raíz y es nulo, se devuelve la raíz (nulo).
   b. Si x es la raíz y no tiene hijos, se elimina x y se devuelve nulo.
   c. Si x es la raíz y tiene solo un hijo, se borra x y la nueva raíz pasa a ser el hijo de x.
   d. Si x tiene los dos hijos, la nueva raíz pasa a ser el hijo derecho de x. El hijo izquierdo se inserta como hijo izquierdo del menor nodo de la rama derecha del árbol original.

ALGORITMO 6.8: Operaciones de árboles binarios ordenados.

```
BUSCAR(T: árbol, dato: tipoDato): árbol
      árbol indice = T

      Mientras indice != NULO:
            datoActual = CONTENIDO(indice)

            Si datoActual == dato:
                  Devolver indice

            Si dato < datoActual:
                  indice = HIJO_IZQUERDO(indice)
            Sino :
                  indice = HIJO_DERECHO(indice)

      Devolver NULO

MÍNIMO(T: árbol): árbol
      indice = T

      Mientras HIJO_IZQUERDO(indice) != NULO:
            indice = HIJO_IZQUERDO(indice)

      Devolver indice

MÁXIMO(T: árbol): árbol
      indice = T

      Mientras HIJO_DERECHO(indice) != NULO:
            indice = HIJO_DERECHO(indice)

      Devolver indice

SIGUIENTE(T: árbol, actual: árbol): árbol
      Si T == NULO ó actual == NULO:
            Devolver NULO

      Si HIJO_DERECHO(actual) != NULO:
            Devolver MÍNIMO(HIJO_DERECHO(actual))

      padre = PADRE(T, actual)

      Si HIJO_IZQUERDO(padre) == actual:
            Devolver padre
```

```
    Mientras padre != NULO y HIJO_DERECHO(padre) == actual:
            actual = padre
            padre = PADRE(T, actual)

      Devolver padre

ANTERIOR(T: árbol, actual: árbol): árbol
      Si T == NULO ó actual == NULO:
            Devolver NULO

      Si HIJO_IZQUERDO(actual) != NULO:
            Devolver MÁXIMO(HIJO_IZQUERDO(actual))

      padre = PADRE(T, actual)

      Si HIJO_DERECHO(padre) == actual:
            Devolver padre

      Mientras padre != NULO y HIJO_IZQUERDO(padre) == actual:
            actual = padre
            padre = PADRE(T, actual)

      Devolver padre

INSERTAR(T: árbol, dato: tipoDato): árbol
      hoja = CREAR(dato, NULO, NULO)

      Si T == NULO:
            Devolver hoja

      Si HIJO_IZQUERDO(T) == NULO y dato < CONTENIDO(T):
            T.hijoIzquierdo = hoja
            Devolver T

      Si HIJO_DERECHO(T) == NULO y dato >= CONTENIDO(T):
            T.hijoDerecho = hoja
            Devolver T

      Si dato < CONTENIDO(T):
            T.hijoIzquierdo = INSERTAR(HIJO_IZQUERDO(T), dato)
            Devolver T

      T.hijoDerecho = INSERTAR(HIJO_DERECHO(T), dato)
      Devolver T

BORRAR(T: árbol, dato: tipoDato): árbol
      árbol posicion = BUSCAR(T, dato)
      árbol padre = PADRE(T, posicion)

      Si posicion == NULO:
            Devolver T

      Si HIJO_IZQUERDO(posicion) == NULO y HIJO_DERECHO(posicion) == NULO:
            Si RAÍZ(T) == posicion:
                  eliminar(T)
```

```
                Devolver NULO

            Si posicion == HIJO_IZQUERDO(padre):
                padre.hijoIzquierdo = NULO
            Sino :
                padre.hijoDerecho = NULO

            eliminar(posicion)
            posicion = NULO
            Devolver T

    Si HIJO_DERECHO(posicion) == NULO:
            Si posicion == RAÍZ(T):
                T = posicion.hijoIzquierdo
            Sino  Si HIJO_IZQUERDO(padre) == posicion:
                padre.hijoIzquierdo = posicion.hijoIzquierdo
            Sino :
                padre.hijoDerecho = posicion.hijoIzquierdo

            eliminar(posicion)
            posicion = NULO
            Devolver T

    reemplazo = MÍNIMO(HIJO_DERECHO(posicion))
    valorReemplazo = CONTENIDO(reemplazo)
    reemplazo.dato = posicion.dato
    posicion.dato = valorReemplazo
    posicion.hijoDerecho = BORRAR(HIJO_DERECHO(posicion), dato)
    Devolver T
```

6.8 ÁRBOLES AVL

Su nombre proviene de los apellidos de sus creadores, Adelson-Velskii y Landis. Son árboles de búsqueda binaria que cumplen, además, con la propiedad de equilibrio: la diferencia entre las alturas de los subárboles de cada uno de sus nodos es a lo más 1. Más formalmente:

Sea T un árbol:

- Si T es vacío, es un AVL.
- Si T es un árbol no vacío y sus subárboles son T_{izq} y T_{der}, entonces T es AVL si y solo si:
  - T_{izq} es AVL.
  - T_{der} es AVL.
  - $\left|ALTURA\left(T_{izq}\right) - ALTURA(T_{der})\right| \leq 1$.

Cabe recordar que los ABO son árboles binarios en los que para cada nodo se cumple que todos los datos de su subárbol izquierdo son menores que la raíz y todos los datos del subárbol derecho son mayores que ella. Además, el tiempo de ejecución promedio para las operaciones sobre un ABO con n nodos es $O(\log n)$, pero es $O(n)$ para el peor caso.

La propiedad de equilibrio que debe cumplir un árbol para ser AVL asegura que la profundidad del árbol siempre es $O(\log n)$, por lo que la eficiencia de las operaciones de búsqueda sobre este tipo de árboles está garantizada, siendo su tiempo de ejecución para el peor caso en un AVL de n nodos $O(\log n)$.

Es claro que todo AVL es también un árbol de búsqueda binaria, por lo que se tienen las mismas operaciones. No obstante, las operaciones para insertar y borrar un nodo definidas para ABO no están diseñadas para respetar la condición de equilibrio, por lo que tras alguna de ellas puede darse que el árbol deje de estar balanceado. En tal caso, es necesario reequilibrar el árbol para que éste sea un AVL. Se pueden definir dos operaciones, llamadas rotaciones, que sirven para este fin y que deben efectuarse desde abajo hacia la raíz comenzando desde el nodo en que se produce el desequilibrio. La figura 6.8 muestra gráficamente los cambios que realiza cada rotación. En ella, los nodos están representados por círculos, mientras que los triángulos representan subárboles. Se considera que el nodo donde se produce el desequilibrio es la raíz.

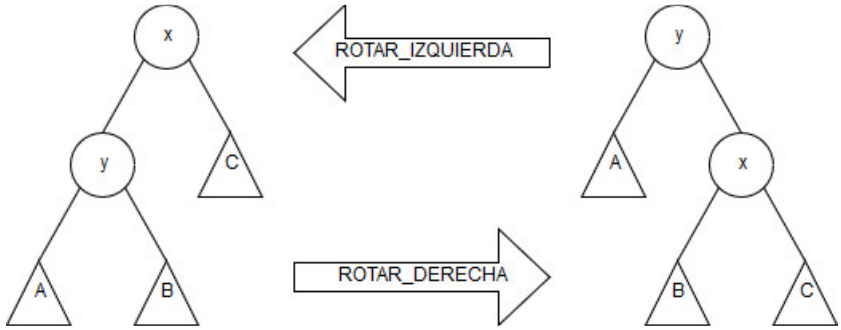


FIGURA 6.8: Rotaciones simples para un AVL.

Con las rotaciones anteriores ya se tienen las operaciones básicas necesarias para poder balancear el AVL tras la inserción o la eliminación de un nodo. Dependiendo del tipo de desequilibrio que se produzca, puede ser necesario realizar una o dos rotaciones, como muestra la figura 6.9.

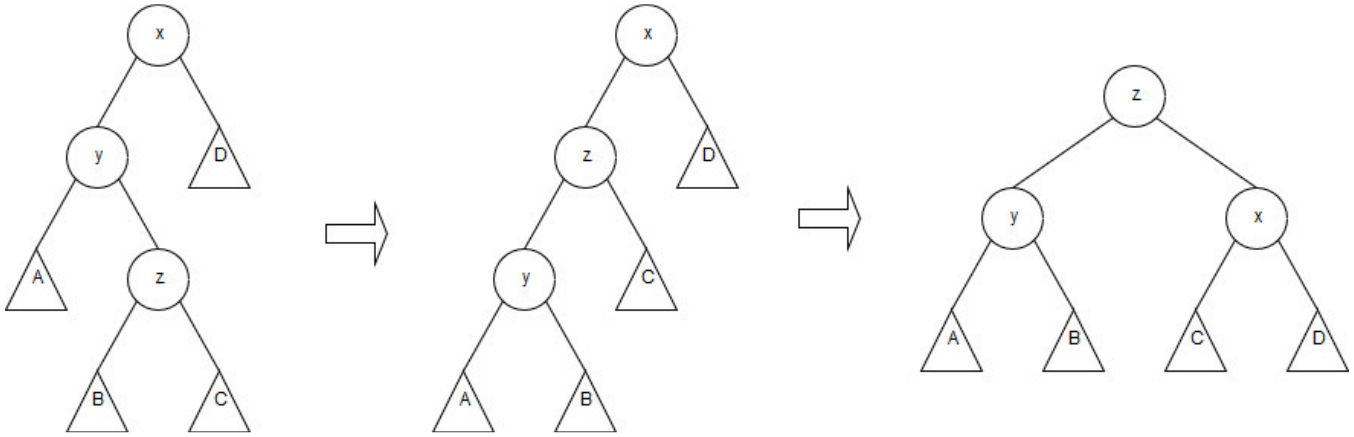


FIGURA 6.9: Rotación doble hacia la derecha para un AVL.

Cuando el desequilibrio es siempre hacia el mismo lado, basta con aplicar una única rotación en el nodo donde éste se produce. Cuando el desequilibrio forma un zig-zag, es necesario rotar al hijo en sentido contrario para luego aplicar la rotación que corresponda en el nodo desequilibrado.

El algoritmo 6.9 muestra las operaciones `insertar()` y `borrar()` para árboles AVL, incluyendo además las operaciones complementarias requeridas para cumplir con la propiedad de equilibrio.

ALGORITMO 6.9: Operaciones de árboles AVL.

```
INSERTAR(T: árbol, dato: tipoDato): árbol
      nuevo = CREAR(dato, NULO, NULO)

      Si T == NULO:
            Devolver nuevo

      Si dato < CONTENIDO(T):
            T.hijoIzquierdo = INSERTAR(HIJO_IZQUERDO(T), dato)
    Sino:
            T.hijoDerecho = INSERTAR(HIJO_DERECHO(T), dato)

    Devolver EQUILIBRAR(T)

BORRAR(T: árbol, dato: tipoDato): árbol
      Si T == NULO:
            Devolver NULO

      Si dato < T.dato:
            T.hijoIzquierdo = BORRAR(HIJO_IZQUERDO(T), dato)
    Sino si dato > T.dato :
            T.hijoDerecho = BORRAR(HIJO_DERECHO(T), dato)
      Sino:
            izquierdo = HIJO_IZQUERDO(T)
            derecho = HIJO_DERECHO(T)

            Si derecho == NULO:
                  eliminar(T)
                  Devolver EQUILIBRAR(izquierdo)

            raíz = MINIMO(derecho)
            PADRE(T, raíz).hijoIzquierdo = NULO
            raíz.hijoIzquierdo = izquierdo
            raíz.hijoDerecho = derecho
            eliminar(T)
            Devolver EQUILIBRAR(raíz)

      Devolver EQUILIBRAR(T)
```

ALGORITMO 6.9 (continuación): Operaciones de árboles AVL.

```
EQUILIBRAR(T: árbol): árbol
      diferencia = DIFERENCIA_ALTURAS(T)

      Si diferencia == 2:
            Si DIFERENCIA_ALTURAS(HIJO_IZQUERDO(T)) < 0:
                  T.hijoIzquierdo = ROTAR_IZQUIERDA(HIJO_IZQUERDO(T))

            Devolver ROTAR_DERECHA(T)

      Si diferencia == -2:
            Si DIFERENCIA_ALTURAS(HIJO_DERECHO(T)) > 0:
                  T.hijoDerecho = ROTAR_DERECHA(HIJO_DERECHO(T))

            Devolver ROTAR_IZQUIERDA(T)

      Devolver T

DIFERENCIA_ALTURAS(T: árbol): entero
      Devolver ALTURA(HIJO_IZQUERDO(T)) - ALTURA(HIJO_DERECHO(T))

ALTURA(T: árbol): entero
      Si T == NULO:
            Devolver NULO

      Si HIJO_IZQUERDO(T) == NULO && HIJO_DERECHO(T) == NULO:
            Devolver 0

      alturaIzquierda = ALTURA(HIJO_IZQUERDO(T))
      alturaDerecha = ALTURA(HIJO_DERECHO(T))

      Si alturaIzquierda > alturaDerecha:
            Devolver 1 + alturaIzquierda

      Devolver 1 + alturaDerecha

ROTAR_DERECHA(T: árbol): árbol
      raíz = HIJO_IZQUERDO(T)
      T.hijoIzquierdo = HIJO_DERECHO(raíz)
      raíz.hijoDerecho = T
      Devolver raíz

ROTAR_IZQUIERDA(T: árbol): árbol
      raíz = HIJO_DERECHO(T)
      T.hijoDerecho = HIJO_IZQUERDO(raíz)
      raíz.hijoIzquierdo = T
      Devolver raíz
```

EJEMPLO 6.2:

La figura 6.10 muestra una serie de inserciones a partir de un árbol nulo, detallando en cada caso las rotaciones realizadas.

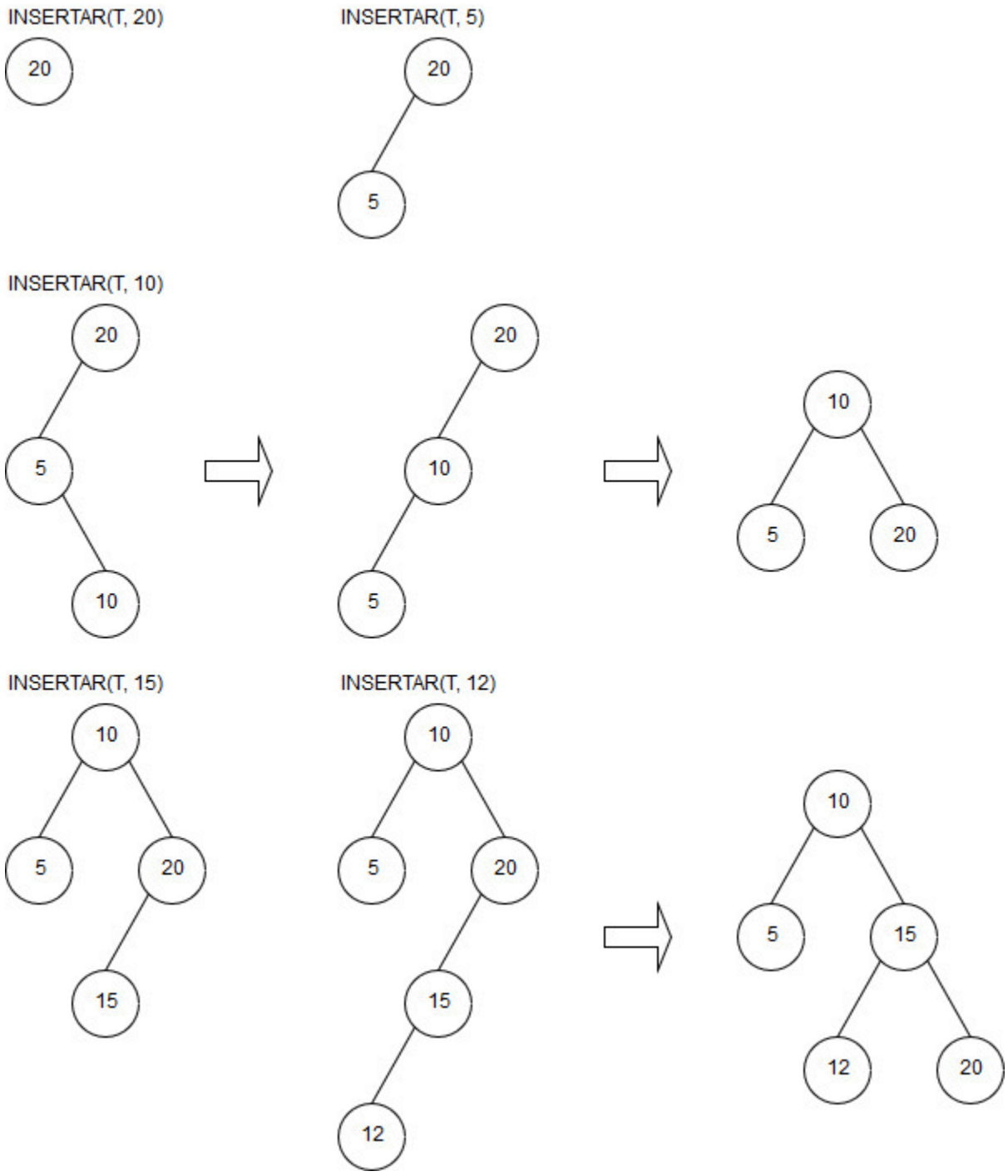


FIGURA 6.10: Rotaciones en árboles AVL.

6.9 ÁRBOLES 2-3

Una clase muy utilizada de árboles son los llamados árboles 2-3. Muchos motores de bases de datos los tienen como base para su funcionamiento, por ser más eficientes que los árboles AVL y por garantizar un tiempo constante de acceso a los datos.

Los árboles 2-3 son árboles no binarios balanceados cuyas características son:

- Cada nodo que no sea una hoja tiene dos o tres hijos.
- Todas las hojas tienen la misma profundidad.
- La información solo se almacena en las hojas, mientras que los nodos restantes sirven para fines de organización.
- Las hojas del árbol están ordenadas de izquierda a derecha.
- Además de los punteros a los hijos, cada nodo intermedio contiene además la siguiente información:
  - El menor elemento del subárbol central.
  - El menor elemento del subárbol derecho.
- Si un nodo tiene dos hijos, éstos son el hijo izquierdo y el hijo central.

Al igual que en los árboles de búsqueda binaria, puede haber más de un árbol 2-3 que permita almacenar los mismos valores, como puede verse en la figura 6.11.

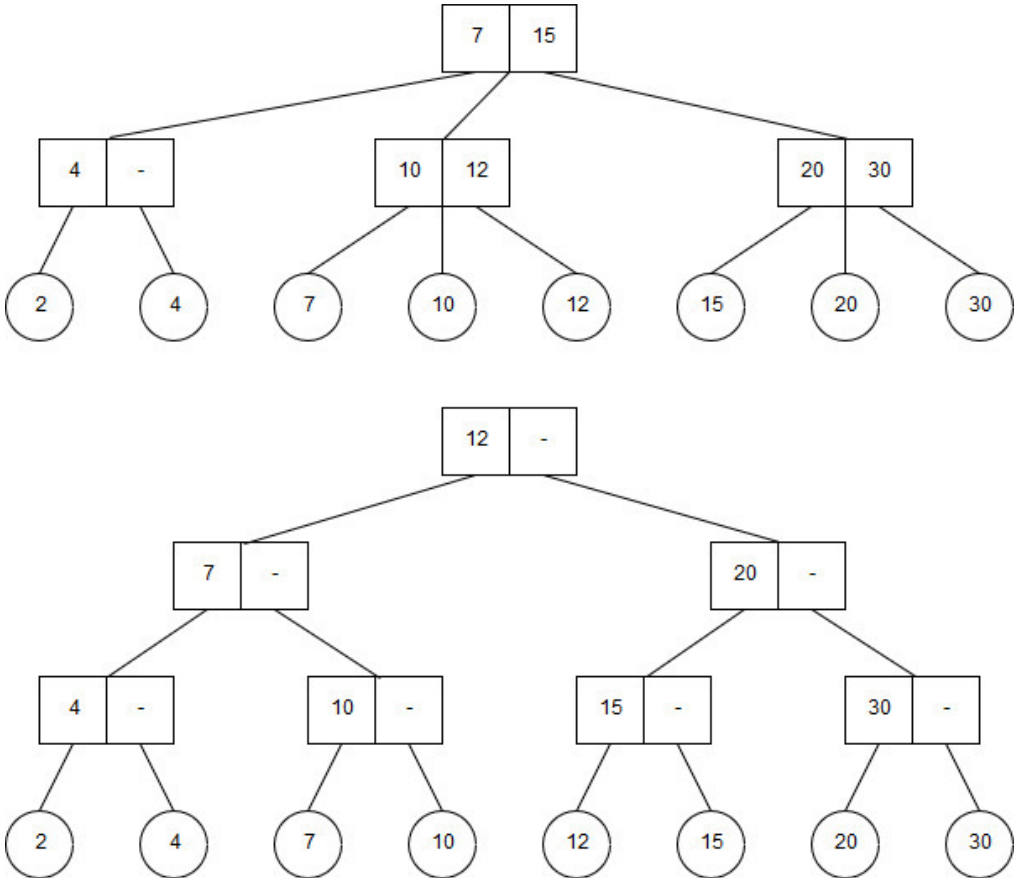


FIGURA 6.11: Tres árboles 2-3 que contienen la misma información.

Las principales operaciones sobre árboles 2-3 son la búsqueda, la inserción y la eliminación, aunque también se requieren muchas de las operaciones generales de árboles y algunas de árboles ordenados.

1. `BUSCAR(T: árbol, dato: tipoDato): árbol.`
   Entrega la posición del elemento `dato` si éste se encuentra en el árbol y nulo en caso contrario.
   El tiempo de búsqueda es proporcional a la altura del árbol, que es $O(\log n)$ con n igual a la cantidad de nodos del árbol. Por otra parte, para la cantidad m de hojas (es decir, la cantidad de datos almacenados) siempre se cumple que $m > n/2$, por lo que el tiempo requerido para la búsqueda es también $O(\log m)$.

2. `INSERTAR(T: árbol, dato: tipoDato): árbol.`
   Inserta una hoja nueva hoja x con el valor `dato` al árbol de forma tal que el árbol siga siendo 2-3. Por la definición de esta clase de árboles, se requiere de casos especiales para árboles vacíos o con una única hoja. Se puede describir la inserción como sigue:
   - Si el árbol está vacío, crear la hoja x y convertirla en la raíz.
   - Si el árbol tiene solo un nodo, crear la hoja x, crear un nuevo nodo interno n con la hoja x y la raíz como hijos izquierdo y central (ordenados de menor a mayor) y con su valor para `mínimoCentral`. El nodo n pasa a ser la raíz.
   - En otro caso:
     - Buscar la hoja h que quedaría inmediatamente a la izquierda de x.
     - Si $a = PADRE(p)$ tiene solo dos hijos, establecer x como hijo central (moviendo el hijo central a la derecha) o hijo derecho. Actualizar los valores de los ancestros según sea necesario.
     - Si a tiene tres hijos:
       - Crear un nuevo nodo interior n.
       - Si h es el hijo derecho de a, hacer que h sea el hijo izquierdo de n y que x sea el hijo central.
       - Si h es el hijo central de a, hacer que el hijo derecho de a pase a ser el hijo central de n y que x sea el hijo izquierdo de n. Actualizar los valores de n.
       - Si h es el hijo izquierdo de a, hacer que los hijos central y derecho de a pasen a ser los hijos izquierdo y central de n, respectivamente. Hacer que x sea el hijo central de n. Actualizar los valores de n.
       - Actualizar los valores de los ancestros según sea necesario.

   Para determinar la complejidad de esta operación se debe tener en cuenta que:
   - Encontrar al padre del nuevo nodo requiere recorrer el camino desde la raíz hasta algún padre de las hojas, operación que ya se vio es de complejidad $O(\log n)$ y también $O(\log m)$.
   - Una vez encontrado el padre, para el peor caso de la inserción se requiere incorporar nuevos nodos y modificar campos de los ancestros hasta la raíz, operación que también cumple con la característica de ser $O(\log n)$ y $O(\log m)$.

   En consecuencia, la complejidad total de la inserción es $O(\log n)$ y $O(\log m)$.

3. `BORRAR(T: árbol, dato: tipoDato): árbol.`
   Es semejante a la inserción, y se puede describir de la siguiente manera:
   - Si la hoja x que contiene a `dato` es el único nodo del árbol, anular la raíz, borrar x y salir.
   - Buscar x y su padre p.

- Si p tiene 3 hijos: borrar x, reacomodar a sus hijos y actualizar sus valores y los de sus ancestros de modo que el árbol siga siendo 2-3.
- Si p tiene solo 2 hijos:
  - Si p es la raíz del árbol: borrar x y hacer que la raíz sea el otro hijo de p.
  - Si p tiene un hermano q con 3 hijos: borrar x, quitar un hijo del hermano y dárselo a p, y actualizar los valores de p, q y sus ancestros de modo que el árbol siga siendo 2-3.
  - Si los hermanos de p tienen solo 2 hijos: borrar x, darle el otro hijo de p a algún hermano q, ajustar los valores de q y eliminar p siguiendo los criterios dados.

El algoritmo 6.10 muestra la definición detallada de la estructura de datos y las operaciones necesarias para implementar un árbol 2-3.

ALGORITMO 6.10: Operaciones de árboles 2-3.

```
tipo hoja:
      dato: tipoDato

tipo nodo:
      mínimoCentral: entero
      mínimoDerecho: entero
      hijoIzquierdo: nodo
      hijoCentral: nodo
      hijoDerecho: nodo

tipo árbol = UNIÓN(tipo nodo, tipo hoja)

CONTENIDO(T: árbol): tipoDato
      Si ES_HOJA(T) == verdadero:
            Devolver h.dato

      Devolver NULO

HIJO_IZQUERDO(T: árbol): árbol
      Si T == NULO:
            Devolver NULO

      Devolver T.hijoIzquierdo

HIJO_CENTRAL(T: árbol): árbol
      Si T == NULO:
            Devolver NULO

      Devolver T.hijoCentral

HIJO_DERECHO(T: árbol): árbol
      Si T == NULO:
            Devolver NULO

      Devolver T.hijoDerecho
```

ALGORITMO 6.10 (continuación): Operaciones de árboles 2-3.

```
CREAR_NODO(hijoIzquierdo: árbol, hijoCentral: árbol, hijoDerecho: árbol): árbol
      nodo nuevo
      nuevo.hijoIzquierdo = hijoIzquierdo
      nuevo.hijoCentral = hijoCentral
      nuevo.hijoDerecho = hijoDerecho
      nuevo.mínimoCentral = MÍNIMO(HIJO_CENTRAL(nuevo))
      nuevo.mínimoDerecho = MÍNIMO(HIJO_DERECHO(nuevo))
      Devolver nuevo

CREAR_HOJA(dato: tipoDato): árbol
      hoja nueva
      nueva.dato = dato
      Devolver nueva

ANULAR(T: árbol): árbol
      Si T == NULO:
            Devolver T

      Si ES_HOJA(T):
            eliminar(T)
            Devolver NULO

      T.hijoIzquierdo = ANULAR(HIJO_IZQUERDO(T))
      T.hijoCentral = ANULAR(HIJO_CENTRAL(T))
      T.hijoDerecho = ANULAR(HIJO_DERECHO(T))
      eliminar(T)
      Devolver NULO

ES_HOJA(T: árbol): booleano
      Si tipo(T) == hoja:
            Devolver verdadero

      Devolver falso

MÍNIMO(T: árbol): entero
      índice = T

      Si T == NULO:
            Devolver NULO

      Mientras ES_HOJA(índice) == falso:
            índice = HIJO_IZQUERDO(índice)

      Devolver CONTENIDO(índice)
```

ALGORITMO 6.10 (continuación): Operaciones de árboles 2-3.

```
PADRE(T: árbol, n: árbol): árbol
      Si n == T:
            Devolver NULO

      Si T != NULO:
            Si HIJO_IZQUERDO(T)==n ó HIJO_CENTRAL(T)==n ó HIJO_DERECHO(T)==n:
                  Devolver T

            Si ES_HOJA(T):
                  Devolver NULO

            padre = PADRE(HIJO_IZQUERDO(T), n)

            Si padre != NULO:
                  Devolver padre

            padre = PADRE(HIJO_CENTRAL(T), n)

            Si padre != NULO:
                  Devolver padre

            padre = PADRE(HIJO_DERECHO(T), n)

      Devolver padre

BUSCAR(T: árbol, dato: tipoDato): árbol
      índice = T

      Mientras ES_HOJA(índice) == falso y índice != NULO:
            Si dato < índice.mínimoCentral:
                  índice = HIJO_IZQUERDO(índice)
            Sino si dato < índice.mínimoDerecho:
                  índice = HIJO_CENTRAL(índice)
            Sino:
                  índice = HIJO_DERECHO(índice)

      Si CONTENIDO(índice) != dato ó índice == NULO:
            Devolver NULO

      Devolver índice
```

ALGORITMO 6.10 (continuación): Operaciones de árboles 2-3.

```
INSERTAR(T: árbol, dato: tipoDato): árbol
      nuevaHoja = CREAR_HOJA(dato)
      padre = T

      Si T == NULO:
            Devolver nuevaHoja

      Si ES_HOJA(T):
            Si dato < CONTENIDO(T):
                  Devolver CREAR_NODO(nuevaHoja, T, NULO)

            Devolver CREAR_NODO(T, nuevaHoja, NULO)

      Mientras ES_HOJA(HIJO_IZQUERDO(padre)) == falso:
            Si dato < padre.mínimoCentral:
                  padre = HIJO_IZQUERDO(padre)
            Sino si dato < padre.mínimoDerecho:
                  padre = HIJO_CENTRAL(padre)
            Sino:
                  padre = HIJO_DERECHO(padre)

      Si HIJO_DERECHO(padre) == NULO:
            Si dato >= padre.mínimoCentral:
                  padre.hijoDerecho = nuevaHoja
            Sino si dato >= CONTENIDO(HIJO_IZQUERDO(padre)):
                  padre.hijoDerecho = HIJO_CENTRAL(padre)
                  padre.hijoCentral = nuevaHoja
            Sino:
                  padre.hijoDerecho = HIJO_CENTRAL(padre)
                  padre.hijoCentral = HIJO_IZQUERDO(padre)
                  padre.hijoIzquierdo = nuevaHoja

            Devolver ACTUALIZAR_ANCESTROS(T, padre)

      Si dato >= CONTENIDO(HIJO_DERECHO(padre)):
            nuevo = CREAR_NODO(HIJO_DERECHO(padre), nuevaHoja, NULO)
      Sino si dato >= padre.mínimoDerecho:
            nuevo = CREAR_NODO(nuevaHoja, HIJO_DERECHO(padre), NULO)
      Sino si dato >= padre.mínimoCentral:
            nuevo = CREAR_NODO(HIJO_CENTRAL(padre), HIJO_DERECHO(padre), NULO)
            padre.hijoCentral = nuevaHoja
      Sino:
            nuevo = CREAR_NODO(HIJO_CENTRAL(padre), HIJO_DERECHO(padre), NULO)
            padre.hijoCentral = HIJO_IZQUERDO(padre)
            padre.hijoIzquierdo = nuevaHoja

      padre.hijoDerecho = NULO
      T = ACTUALIZAR_ANCESTROS(T, padre)
      Devolver INSERTAR_INTERMEDIO(T, PADRE(T, padre), nuevo)
```

ALGORITMO 6.10 (continuación): Operaciones de árboles 2-3.

```
INSERTAR_INTERMEDIO(T: árbol, padre: árbol, intermedio: árbol): árbol
      Si padre == NULO:
            nuevo = CREAR_NODO(T, intermedio, NULO)
            Devolver nuevo

      Si HIJO_DERECHO(padre) == NULO:
            padre.hijoDerecho = intermedio
            Devolver ACTUALIZAR_ANCESTROS(T, padre)

      Si intermedio.mínimoCentral >= padre.mínimoDerecho:
            nuevo = CREAR_NODO(HIJO_DERECHO(padre), intermedio, NULO)
      Sino si intermedio.mínimoCentral >= padre.mínimoDerecho:
            nuevo = CREAR_NODO(intermedio, HIJO_DERECHO(padre), NULO)
      Sino si intermedio.mínimoCentral >= padre.mínimoCentral:
            nuevo = CREAR_NODO(HIJO_CENTRAL(padre), HIJO_DERECHO(padre), NULO)
            padre.hijoCentral = intermedio
      Sino:
            nuevo = CREAR_NODO(HIJO_CENTRAL(padre), HIJO_DERECHO(padre), NULO)
            padre.hijoCentral = HIJO_IZQUERDO(padre)
            padre.hijoIzquierdo = nuevo

      padre.hijoDerecho = NULO
      T = ACTUALIZAR_ANCESTROS(T, padre)
      Devolver INSERTAR_INTERMEDIO(T, PADRE(T, padre), nuevo)

ACTUALIZAR_ANCESTROS(T: árbol, índice: árbol): árbol
      Mientras índice != NULO:
            índice.mínimoCentral = MÍNIMO(HIJO_CENTRAL(índice))
            índice.mínimoDerecho = MÍNIMO(HIJO_DERECHO(índice))
            índice = PADRE(T, índice)

      Devolver T

BORRAR(T: árbol, dato: tipoDato): árbol
      posicionDato = BUSCAR(T, dato)

      Si posicionDato == NULO:
            Devolver T

      Si T == posicionDato:
            eliminar(T)
            Devolver NULO

      padre = PADRE(T, posicionDato)

      Si HIJO_DERECHO(padre) != NULO:
            Si posicionDato == HIJO_DERECHO(padre):
                  eliminar(HIJO_DERECHO(padre))
            Sino si posicionDato == HIJO_CENTRAL(padre):
                  eliminar(HIJO_CENTRAL(padre))
                  padre.hijoCentral = HIJO_DERECHO(padre)
```

ALGORITMO 6.10 (continuación): Operaciones de árboles 2-3.

```
                Sino:
                      eliminar(HIJO_IZQUERDO(padre))
                      padre.hijoIzquierdo = HIJO_CENTRAL(padre)
                      padre.hijoCentral = HIJO_DERECHO(padre)

                padre.hijoDerecho = NULO
                Devolver ACTUALIZAR_ANCESTROS(T, padre)
          Sino:
                Si posicionDato == HIJO_CENTRAL(padre):
                      eliminar(HIJO_CENTRAL(padre))
                Sino:
                      eliminar(HIJO_IZQUERDO(padre))
                      padre.hijoIzquierdo = HIJO_CENTRAL(padre)

                padre.hijoCentral = NULO
                padre.mínimoCentral = NULO
                Devolver BORRAR_INTERMEDIO(T, PADRE(T, padre), padre)

          Devolver T

    BORRAR_INTERMEDIO(T: árbol, padre: árbol, intermedio: árbol): árbol
          Si padre == NULO:
                auxiliar = HIJO_IZQUERDO(intermedio)
                eliminar(padre)
                Devolver auxiliar

          Si intermedio == HIJO_DERECHO(padre):
                hermano = HIJO_CENTRAL(padre)

                Si HIJO_DERECHO(hermano) == NULO:
                      hermano.hijoDerecho = HIJO_IZQUERDO(intermedio)
                      eliminar(intermedio)
                      padre.hijoDerecho = NULO
                      T = ACTUALIZAR_ANCESTROS(T, hermano)
                Sino:
                      intermedio.hijoCentral = HIJO_IZQUERDO(intermedio)
                      intermedio.hijoIzquierdo = HIJO_DERECHO(hermano)
                      hermano.hijoDerecho = NULO
                      T = ACTUALIZAR_ANCESTROS(T, intermedio)
                      T = ACTUALIZAR_ANCESTROS(T, hermano)

          Sino si intermedio == HIJO_CENTRAL(padre):
                hermano = HIJO_IZQUERDO(padre)

                Si HIJO_DERECHO(hermano) == NULO:
                      hermano.hijoDerecho = HIJO_IZQUERDO(intermedio)
                      eliminar(intermedio)
                      padre.hijoCentral = HIJO_DERECHO(padre)
                      padre.hijoDerecho = NULO
                      T = ACTUALIZAR_ANCESTROS(T, hermano)
```

ALGORITMO 6.10 (continuación): Operaciones de árboles 2-3.

```
            Sino:
                  intermedio.hijoCentral = HIJO_IZQUERDO(intermedio)
                  intermedio.hijoIzquierdo = HIJO_DERECHO(hermano)
                  hermano.hijoDerecho = NULO
                  T = ACTUALIZAR_ANCESTROS(T, intermedio)
                  T = ACTUALIZAR_ANCESTROS(T, hermano)
      Sino:
            hermano = HIJO_CENTRAL(padre)

            Si HIJO_DERECHO(hermano) == NULO:
                  intermedio.hijoCentral = HIJO_IZQUERDO(hermano)
                  intermedio.hijoDerecho = HIJO_CENTRAL(hermano)
                  eliminar(hermano)
                  padre.hijoCentral = HIJO_DERECHO(padre)
                  padre.hijoDerecho = NULO
                  T = ACTUALIZAR_ANCESTROS(T, intermedio)
            Sino:
                  intermedio.hijoCentral = HIJO_IZQUERDO(intermedio)
                  hermano.hijoIzquierdo = HIJO_CENTRAL(hermano)
                  hermano.hijoCentral = HIJO_DERECHO(hermano)
                  hermano.hijoDerecho = NULO
                  T = ACTUALIZAR_ANCESTROS(T, intermedio)
                  T = ACTUALIZAR_ANCESTROS(T, hermano)

      Si HIJO_CENTRAL(padre) == NULO:
            Devolver BORRAR_INTERMEDIO(T, PADRE(T, padre), padre)

      Devolver T
```

EJEMPLO 6.3:

La figura 6.12 muestra una serie de inserciones a partir de un árbol nulo.

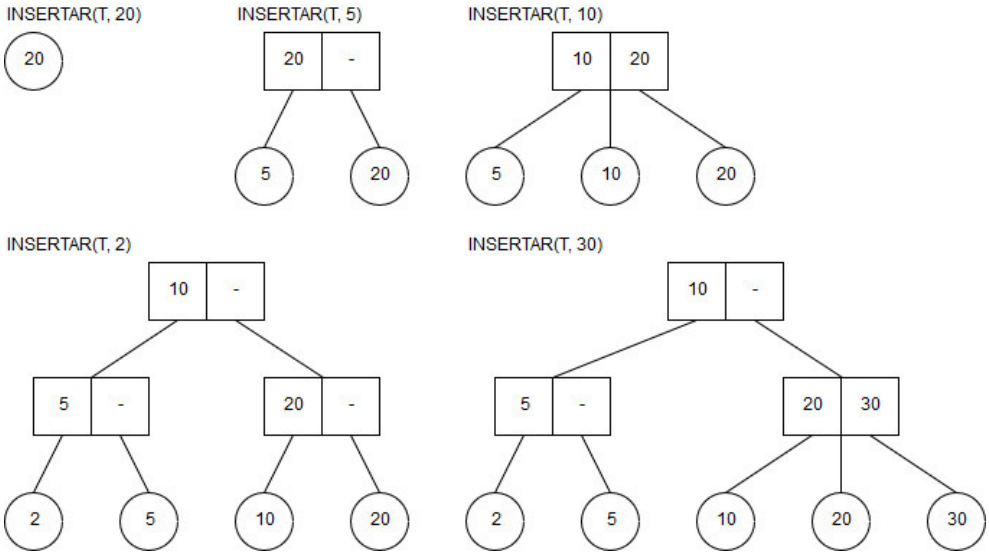


FIGURA 6.12: Inserción en árboles 2-3.

7 TABLAS *HASH*

El *hashing* es una técnica ampliamente utilizada para la implementación de diccionarios (conjuntos de datos para los que solo se requiere contar con funciones para saber si un elemento es miembro del conjunto y para efectuar inserciones y eliminaciones de elementos).

La idea es indexar los elementos a almacenar, que pueden ser un subconjunto k relativamente pequeño de un conjunto universo U potencialmente infinito, en una tabla cuyo tamaño pueda ser significativamente más pequeño que la cantidad de elementos del conjunto universo.

Más concretamente, la idea es reducir el tamaño de la tabla a $|k|$ pero con el beneficio de que la búsqueda de un elemento en la tabla tome un tiempo $O(1)$.

En promedio, el *hashing* requiere de un tiempo constante para efectuar una operación y no necesita que los conjuntos sean subconjuntos de otro conjunto finito mayor llamado universo. En el peor caso, este método necesita un tiempo proporcional al tamaño del conjunto para llevar a cabo cada operación, al igual que las listas. No obstante, si se hace un diseño cuidadoso, es posible hacer que la probabilidad de requerir un tiempo mayor a una constante para una operación *hashing* sea arbitrariamente pequeña.

En general, al emplear esta estructura de datos se distinguen dos formas distintas de *hashing*:
- *Hashing* abierto o externo: permite que el conjunto sea almacenado en un espacio potencialmente infinito, por lo que no establece un límite para el tamaño del conjunto de datos.
- *Hashing* cerrado o interno: usa una cantidad fija de espacio para almacenamiento, lo que en consecuencia establece un límite para el tamaño del conjunto.

7.1 *HASHING* ABIERTO

La estructura de datos básica para el *hashing* abierto se presenta en la figura 7.1. La idea es que la cantidad potencialmente infinita de elementos del conjunto sea particionada en una cantidad finita de clases o categorías. Si se quiere tener b clases, numeradas desde 0 hasta $b-1$, se necesita una función *hash* h tal que, para cada objeto x del tipo de datos correspondiente al conjunto de elementos a almacenar, el resultado de $h(x)$ sea un entero entre 0 y $b-1$. Este valor corresponde a la categoría a la cual pertenece x. Habitualmente, x se denomina llave y $h(x)$ recibe el nombre de valor *hash* de x. Otra nomenclatura frecuente es referirse a las clases como celdas, y se dice que x pertenece a la celda $h(x)$.

Para representar esta estructura de datos se crea un arreglo, denominado tabla de celdas, indexado mediante los números de celda entre 0 y $b-1$. Este arreglo ofrece cabeceras para b listas (u otras estructuras de datos más complejas), donde el contenido de cada una de estas últimas corresponde a los elementos que pertenecen a la celda con esa numeración, es decir, cada lista i contiene a los elementos x tales que $h(x) = i$.

Se espera que cada celda contenga una cantidad similar de elementos, de modo que cada lista sea corta. Si el conjunto total tiene n elementos, entonces la celda promedio tendrá n/b miembros. Cuando es posible estimar n y se escoge b casi igual de grande, la celda promedio contendrá solo uno o dos miembros

y las operaciones del diccionario requerirán, en promedio, una cantidad constante y pequeña de pasos, independientemente de los tamaños de n y b.

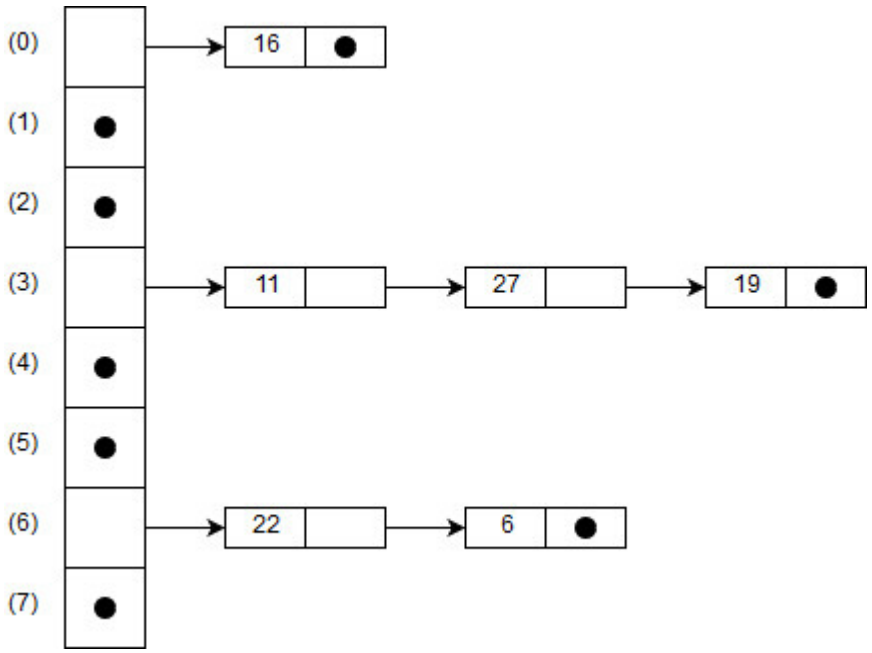


FIGURA 7.1: Organización de datos en *hashing* abierto.

No siempre es claro que sea posible escoger h de modo que un conjunto típico de datos vaya a tener sus elementos distribuidos de manera medianamente homogénea entre las distintas celdas. La idea es escogerla de forma tal que realmente "revuelva" su argumento, es decir, que $h(x)$ sea un valor "aleatorio" que no dependa de x de ninguna manera trivial.

EJEMPLO 7.1:

El algoritmo 7.1 muestra una manera de implementar una tabla hash que almacena cadenas de caracteres, donde cada celda es una lista. Se considera una alternativa razonablemente buena, aunque no perfecta, de función hash cuya idea es tratar los caracteres como enteros (esto es directo en C), usando el código de la máquina para el símbolo con objeto de definir la correspondencia. Así, si x es una llave y el tipo definido para las llaves es un arreglo de caracteres de tamaño 10, se podría definir una función hash h que sume los valores numéricos de los caracteres y luego divida el resultado por b, entregando como respuesta el resto de dicha división (que es un entero entre 0 y $b - 1$).

Para facilitar la lectura, las operaciones de listas se distinguen de las de la tabla hash porque se les añadió el sufijo `_L`.

La figura 7.2 muestra una tabla hash de 5 celdas tras la inserción de algunos datos considerando la implementación del algoritmo 7.1.

ALGORITMO 7.1: Implementación de una tabla *hash* abierta para almacenar strings.

```
tipo tablaHash:
      tabla: lista[]
      celdas: entero

tipo ubicación:
      llave: entero
      posicion: entero

CREAR(celdas: entero): tablaHash
      tablaHash nueva

      nueva.celdas = celdas
      nueva.tabla = lista[celdas]

      Para i desde 0 hasta celdas - 1:
            nueva.tabla[i] = NULO

      Devolver nueva

FUNCIÓN_HASH(dato: string, celdas: entero): entero
      suma = 0

      Para i desde 0 hasta min(10, LARGO(dato)) -1:
            suma = suma + dato[i]

      Devolver suma % celdas

INSERTAR(T: tablaHash, dato: string): tablaHash
      llave = FUNCIÓN_HASH(dato, T.celdas)

      T.tabla[llave] = INSERTAR_L(T.tabla[llave], 0, dato)
      Devolver T

BUSCAR(T: tablaHash, dato: string): ubicación
      llave = FUNCIÓN_HASH(dato, T.celdas)
      ubicación loc

      loc.llave = llave
      loc.posicion = BUSCAR_L(T.tabla[llave], dato)
      Devolver loc

OBTENER(T: tablaHash, loc: ubicación): string
      Devolver OBTENER_L(T.tabla[loc.llave], loc.posicion)

BORRAR(T: tablaHash, dato: string): tablaHash
      ubicación loc = BUSCAR(T, dato)

      T.tabla[loc.llave] = BORRAR_L(T.tabla[loc.llave], loc.posicion)
      Devolver T
```

ALGORITMO 7.1 (continuación): Implementación de una tabla *hash* abierta para almacenar strings.

```
ANULAR(T: tablaHash): tablaHash
      Para i desde 0 hasta celdas - 1:
            T.tabla[i] = ANULAR_L(T.tabla[i])

      T.celdas = 0
      Devolver T
```

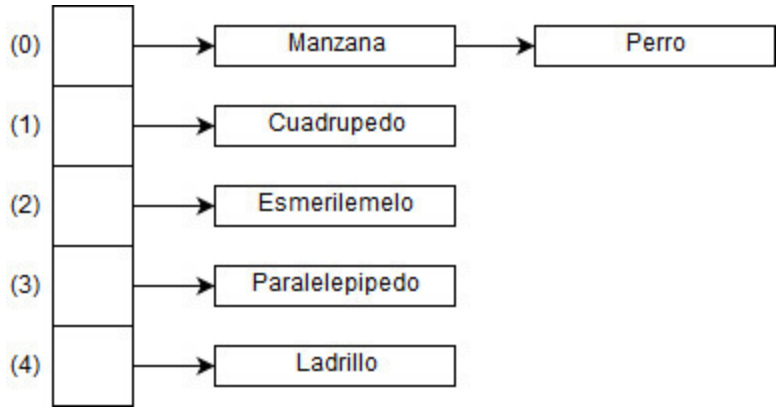


FIGURA 7.2: Tabla hash abierta de 5 celdas implementada de acuerdo al algoritmo 7.1.

7.2 *HASHING* CERRADO

Una tabla *hash* cerrada almacena los elementos del conjunto en la tabla de celdas misma en lugar de usar la tabla para almacenar una lista de cabeceras. En consecuencia, solo es posible almacenar un elemento en cada celda. No obstante, para evitar problemas al tratar de almacenar un segundo elemento en un celda (es decir, cuando se produce una colisión) es posible utilizar una estrategia de *rehash*, la cual escoge una serie de posiciones alternativas $h_1(x)$, $h_2(x)$, ... para el elemento x dentro de la tabla de celdas. Se prueba cada una de estas ubicaciones hasta encontrar una que esté vacía. Si ninguna lo está, entonces la tabla está llena y no se puede insertar el elemento x.

La prueba de pertenencia para un elemento x requiere que se examinen $h(x)$, $h_1(x)$, $h_2(x)$, etc. hasta encontrar el elemento x o bien una celda vacía. La búsqueda puede detenerse al encontrar una celda vacía porque, si no se permite eliminar elementos y, por ejemplo, $h_3(x)$ es la primera celda vacía de la secuencia, no es posible que x esté en las celdas $h_4(x)$, $h_5(x)$ o subsecuentes, puesto que solo hubiese sido posible insertar x en ellas si $h_3(x)$ hubiese estado lleno al momento de la inserción.

Si, en cambio, la eliminación de elementos está permitida, el mecanismo anterior no permite, al momento de encontrar una celda vacía, saber si x podría estar en alguna otra parte porque la celda ahora vacío estaba ocupado al momento de insertar x. El enfoque más efectivo para resolver este problema es insertar una constante *eliminado* (distinta a los valores posibles de ser ingresados y de la constante *vacío*) en aquellas celdas de las que se retire el elemento, para así poder eliminar elementos sin necesidad de tener que recorrer toda la taba al momento de efectuar una búsqueda. Al insertar, se puede tratar a las celdas que contengan *borrado* como vacíos, por lo que dicho espacio podría ser reutilizado.

EJEMPLO 7.2:

El algoritmo 7.2 muestra una manera de implementar una tabla *hash* cerrada que almacena enteros. La función *hash* empleada es $h(x) = (3x + 5)mod(b)$, donde b corresponde a la cantidad de celdas. Para el *rehasing*, se emplea la estrategia lineal, es decir, $g(x) = (h(x) + 1)mod(b)$.

La tabla 7.1 muestra una tabla *hash* de 5 celdas tras algunas operaciones de inserción y borrado considerando la implementación del algoritmo 7.2. se han reemplazado los nombres para facilitar la lectura.

ALGORITMO 7.2: Implementación de una tabla *hash* cerrada para almacenar enteros.

```
tipo tablaHash:
      tabla: entero[]
      celdas: entero

CREAR(celdas: entero): tablaHash
      tablaHash nueva
      nueva.celdas = celdas
      nueva.tabla = entero[celdas]

      Para i desde 0 hasta celdas – 1:
            nueva.tabla[i] = VACÍO

      Devolver  nueva

FUNCIÓN_HASH(dato: entero, celdas: entero): entero
      Devolver |3 * dato + 5| % celdas

FUNCIÓN_REHASH(int llave, celdas: entero): entero
      Devolver  (llave + 1) % celdas

INSERTAR(T: tablaHash, dato: entero): tablaHash
      llave = FUNCIÓN_HASH(dato, T.celdas)
      cuenta = 0

      Mientras cuenta < T.celdas:
            Si T.tabla[llave] == VACÍO ó T.tabla[llave] == BORRADO:
                  T.tabla[llave] = dato
                  Devolver  T

            llave = FUNCIÓN_REHASH(llave, T.celdas)
            cuenta = cuenta + 1

      Devolver  T
```

ALGORITMO 7.2 (continuación): Implementación de una tabla *hash* cerrada para almacenar enteros.

```
BUSCAR(T: tablaHash, dato: entero): entero
      llave = FUNCIÓN_HASH(dato, T.celdas)
      cuenta = 0

      Mientras T.tabla[llave] != dato y cuenta < T.celdas:
            Si T.tabla[llave] == VACÍO:
                  Devolver  NULO

            cuenta = cuenta + 1
            llave = FUNCIÓN_REHASH(llave, T.celdas)

      Si T.tabla[llave] == dato:
            Devolver  llave

      Devolver  NULO

OBTENER(T: tablaHash, locación: entero): entero
      Si locación < 0 ó locación >= T.celdas:
            Devolver  NULO

      Devolver  T.tabla[locación]

BORRAR(T: tablaHash, dato: entero): tablaHash
      locación = BUSCAR(T, dato)

      Si locación != NULO:
            T.tabla[locación] = BORRADO

      Devolver  T

ANULAR(T: tablaHash): tablaHash
      T.celdas = 0
      Devolver  T
```

TABLA 7.1: Ejemplo de funcionamiento de *hashing* cerrado.

Operación(T, x)	**h(x)**	**tabla[0]**	**tabla[1]**	**tabla[2]**	**tabla[3]**	**tabla[4]**
INSERTAR(T, 5)	0	5	VACÍO	VACÍO	VACÍO	VACÍO
INSERTAR(T, 12)	1	5	12	VACÍO	VACÍO	VACÍO
INSERTAR(T, 42)	1	5	12	42	VACÍO	VACÍO
INSERTAR(T, 8)	4	5	12	42	VACÍO	8
BORRAR(T, 12)	1	5	BORRADO	42	VACÍO	8
BORRAR(T, 5)	0	BORRADO	BORRADO	42	VACÍO	8
INSERTAR(T, 37)	1	BORRADO	37	42	VACÍO	8
BORRAR(T, 8)	4	BORRADO	37	42	VACÍO	BORRADO
INSERTAR(T, 24)	2	BORRADO	37	42	24	BORRADO
INSERTAR(T, 0)	0	0	37	42	24	BORRADO
INSERTAR(T, -6)	3	0	37	42	24	-6
BORRAR(T, 24)	2	0	37	42	BORRADO	-6

7.3 EFICIENCIA DE LAS FUNCIONES *HASH*

Como ya se ha mencionado, el *hashing* es una manera eficiente de representar diccionarios y otros tipos de datos abstractos basados en conjuntos. En esta sección se examina el tiempo promedio requerido para efectuar una operación de diccionario en una tabla *hash* abierta. Si se cuenta con b celdas y n elementos, entonces cada celda tendrá en promedio n/b miembros, y se espera que una inserción, eliminación o prueba de pertenencia requiera de tiempo $O(1 + n/b) = O(n/b)$. La constante 1 representa el tiempo necesario para encontrar la celda, mientras que n/b corresponde al tiempo necesario para buscar en la celda. Si es posible escoger b de modo que sea cercano a n, este último tiempo se convierte en una constante para cada operación. Así, si se asume que es igualmente probable que un elemento sea asignado a cualquier celda, el tiempo requerido para efectuar una operación es una constante independiente de n.

EJEMPLO 7.3:

Si, por ejemplo, se desea crear una tabla *hash* con todos los identificadores presentes en el código fuente de un programa cualquiera, cada vez que se encuentre la declaración de un nuevo identificador, éste debe ser insertado en la tabla *hash* luego de comprobar que no existía previamente. Asumiendo que es equiprobable que un identificador esté en cualquier celda, es posible construir la tabla *hash* con n elementos en tiempo $O\big(n(1 + n/b)\big) = O(n^2/b)$ que, al hacer $b = n$, se convierte en $O(n)$.

En la siguiente fase, se encuentran identificadores en el cuerpo del programa. Es necesario encontrar los identificadores en la tabla *hash* para devolver información asociada a ellos. El tiempo esperado para encontrar un identificador, si éste puede ser cualquier elemento de la tabla con igual probabilidad, corresponde al tiempo promedio necesario para insertar un elemento, asumiendo que cada nuevo elemento de una celda se inserta la final de la lista. Así, el tiempo esperado de búsqueda es también $O(1 + n/b) = O(n/b)$.

El análisis anterior asume que una función *hash* distribuye los elementos uniformemente entre las celdas, pero ¿existe tal función? La función definida en el ejemplo 7.1 es una típica función *hash*, para la cual se analiza ahora su desempeño.

EJEMPLO 7.4:

Suponga que se usa la función *hash* del ejemplo 7.1 para almacenar 100 llaves correspondientes a los strings A0, A1, ..., A99 en una tabla con 100 celdas. Suponiendo que los códigos numéricos para los símbolos correspondientes a los dígitos del 0 al 9 forman una sucesión aritmética como ocurre en los códigos ASCII y EBCDIC, es fácil comprobar que las llaves son asignadas, a lo sumo, a 29 de las 100 celdas y que la celda con más elementos contendrá 9 de las 100 llaves. Si se calcula la cantidad promedio de pasos necesarios para la inserción, asumiendo que la inserción del i-ésimo elemento reuiere $i + 1$ pasos, se requiere un total de 395 pasos para almacenar las 100 llaves, mientras que la estimación $n(1 + n/b)$ sugiere tan solo 200 pasos.

La sencilla función *hash* del ejemplo 7.1 podría tratar ciertos conjuntos de entrada, tales como los strings consecutivos del ejemplo 7.4, de manera no aleatoria. Existen alternativas que se acercan más a la aleatoriedad. Por ejemplo, si se consideran llaves numéricas conformadas por 5 dígitos, éstas pueden ser elevadas al cuadrado para obtener un número con 9 ó 10 dígitos y luego tomar solo los dígitos centrales

de este resultado. Para un llave n, los dígitos 4 a 7 contando las posiciones desde la derecha tienen valores que dependen de casi todos los dígitos de n. Por ejemplo, el cuarto dígito depende de todos los dígitos de n excepto el de más a la izquierda, y el quinto, de todos. Así, para $b = 100$, se podría escoger tomar como valor de la función *hash* a los dígitos 6 y 5.

Se puede generalizar la idea anterior para situaciones en que b no es una potencia de 10. Suponga que las llaves son enteros en el rango entre 0 y k. Si se busca un entero c tal que $bc^2 \approx k^2$, entonces la función:

$$h(n) = \frac{n^2}{c} \; mod \; b$$

efectivamente toma un valor en base b desde el centro de n^2.

En este caso, si $k = 100$ y $b = 8$, se podría escoger $c = 354$. Así, se tiene:

$$h(456) = \frac{456^2}{354} \; mod \; 8 = \frac{207936}{354} \; mod \; 8 = 587 \; mod \; 8 = 3$$

Para usar la estrategia de elevar al cuadrado y tomar los dígitos centrales cuando las llaves son strings, se pueden agrupar los caracteres de derecha a izquierda en grupos de un tamaño fijo de caracteres, por ejemplo 4, rellenando a la izquierda con espacios vacíos de ser necesario. Luego, tratar cada bloque como un único número entero formado por la concatenación de los códigos de los caracteres. Finalmente, sumar todos los bloques y aplicar la estrategia antes descrita para llaves numéricas.

8 *HEAPS*

Un *heap* (binario) es un objeto de tipo arreglo que puede ser visualizado como un árbol binario casi completo, es decir, cada nodo del árbol corresponde a un elemento del arreglo, y el árbol tiene todos sus niveles llenos excepto el más bajo, el cual está lleno desde la izquierda hasta un punto determinado.

Los heaps son muy útiles para implementar colas de prioridad, en que los elementos se ordenan de acuerdo a su importancia o urgencia. También son de ayuda para implementar algunos algoritmos para grafos de manera más eficiente, así como para implementar algunas funciones estadísticas.

Para representar un *heap* A mediante un arreglo, es necesario conocer dos atributos: su longitud ($A.largo$), correspondiente al tamaño del arreglo, y el tamaño del heap ($A.tamHeap$), que indica la cantidad de elementos almacenados en A. en otras palabras, los elementos en $A[1..A.tamHeap]$ contienen datos, mientras el resto está vacío o con basura. La raíz del árbol está almacenada en $A[1]$ y, dado el índice i de un nodo, se pueden calcular fácilmente los índices de su padre y de sus hijos:

1. `PADRE(H: heap, i: entero): entero`.
   Devuelve la posición del padre del nodo i, dada por $\lfloor i/2 \rfloor$.

2. `HIJO_IZQUIERDO(H: heap, i: entero): entero`.
   Devuelve la posición del hijo izquierdo del nodo i, dada por $2(i+1)-1$.

3. `HIJO_DERECHO(H: heap, i: entero): entero`.
   Devuelve la posición del hijo izquierdo del nodo i, dada por $2(i+1)$.

Viendo un *heap* como un árbol, se define la altura de un nodo en un *heap* como la cantidad de aristas en el camino simple más largo desde el nodo hasta una hoja, y se define la altura del *heap* como la altura de su raíz. Como los *heaps* se basan en un árbol binario completo, se puede demostrar que las operaciones sobre *heaps* tardan, en el peor caso, un tiempo proporcional a su altura, es decir, $O(\log n)$.

Existen dos tipos de *heaps* binarios: los *max-heaps* y los *min-heaps*. En ambos casos, los valores de los nodos deben cumplir una determinada propiedad.

En un *max-heap*, para cada nodo i distinto de la raíz se debe cumplir que:

$$A[PADRE(i)] \geq A[i]$$

La figura 8.1 muestra un *max-heap* visto como árbol binario y como un arreglo. Organizados en forma opuesta a los *max-heaps*, la propiedad que deben cumplir los *min-heaps* es que para cada nodo i distinto de la raíz se debe cumplir que:

$$A[PADRE(i)] \leq A[i]$$

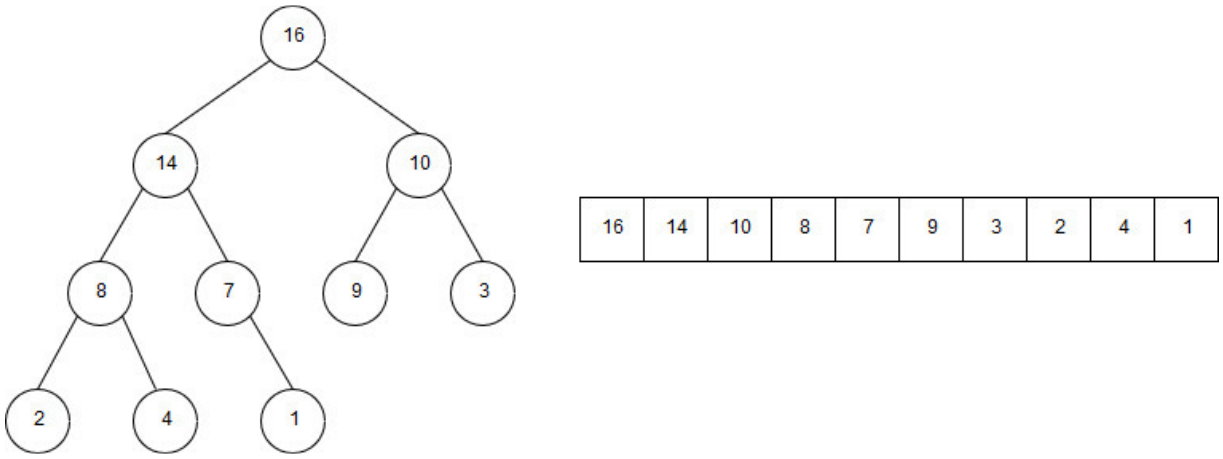


FIGURA 8.1: Un *max-heap* visto como árbol y como arreglo.

8.1 OPERACIONES SOBRE *HEAPS*

Existen diversas operaciones que se pueden aplicar sobre *heaps* (algunas de ellas difieren dependiendo del tipo de *heap*), entre ellas:

1. `INSERTAR(H: heap, dato: tipoDato): heap.`
   Inserta un nuevo dato en el heap *H*. Requiere equilibrar el *heap* resultante.

2. `BORRAR_MÁXIMO(H: heap): heap.`
   Elimina la raíz de un *max-heap*. Para *min-heaps*, la operación equivalente es `BORRAR_MÍNIMO(H)`.

3. `BORRAR(H: heap, dato: tipoDato): heap.`
   Elimina un dato del *heap*. Requiere equilibrar el *heap* resultante.

4. `ES_VACÍO(H: heap): booleano.`
   Devuelve verdadero si el heap no contiene ningún elemento, y falso en caso contrario.

5. `CREAR(): heap.`
   Crea un *heap* vacío.

6. `ANULAR(H: heap): heap.`
   Borra todos los elementos del heap.

7. `CREAR_HEAP(arreglo: tipoDato[]): heap.`
   Crea un *heap* a partir de los elementos contenidos en el arreglo.

8. `PERCOLAR_ARRIBA(H: heap, i: entero): heap.`
   Mueve el nodo *i* hacia arriba en el árbol. Se usa para reestablecer la propiedad de equilibrio.

9. `PERCOLAR_ABAJO(H: heap, i: entero): heap.`
   Mueve el nodo *i* hacia abajo en el árbol. Se usa para reestablecer la propiedad de equilibrio.

10. `A_HEAP(arreglo: tipoDato[], i:entero, tamañoHeap: entero).`
    Ordena los elementos en el arreglo para que cumplan la propiedad de *heap*.

11. `Heap CREAR_HEAP(tipoDato arreglo[]).`
    Crea un *heap* a partir de los elementos contenidos en el arreglo.

El algoritmo 8.1 muestra el detalle de las operaciones definidas para crear un *max-heap*.

ALGORITMO 8.1: Implementación de un *max-heap*.

```
tipo maxHeap:
      arreglo: tipoDato[]
      tamaño: entero
      tamañoArreglo: entero

PADRE(H: maxHeap, i: entero): entero
      Si ES_VACIO(H):
            Devolver NULO

      Devolver i / 2

HIJO_IZQUIERDO(H: maxHeap, i: entero): entero
      hijo = 2 * i + 1

      Si ES_VACIO(H) ó hijo >= H.tamaño:
            Devolver NULO

      Devolver hijo

HIJO_DERECHO(H: maxHeap, i: entero): entero
      hijo = 2 * i + 2

      Si ES_VACIO(H) ó hijo > H.tamaño:
            Devolver NULO

      Devolver hijo

ES_VACIO(H: maxHeap): booleano
      Si H.tamaño == 0:
            Devolver verdadero

      Devolver falso

CREAR(): maxHeap
      maxHeap H
      H.arreglo = tipoDato[MAX_TAMANO]
      H.tamañoArreglo = MAX_TAMANO
      H.tamaño = 0
      Devolver H
```

ALGORITMO 8.1 (continuación): Implementación de un *max-heap*.

```
ANULAR(H: maxHeap): maxHeap
      H.tamaño = 0
      Devolver H

INSERTAR(H: maxHeap, dato: tipoDato): maxHeap
      Si H.tamaño == H.tamañoArreglo:
            Devolver H

      H.arreglo[H.tamaño] = dato
      H.tamaño = H.tamaño + 1
      H = PERCOLAR_ARRIBA(H, H.tamaño - 1)
      Devolver H

BORRAR_MAXIMO(H: maxHeap): maxHeap
      Si ES_VACIO(H) == verdadero:
            Devolver H

      H.arreglo[0] = H.arreglo[H.tamaño -1]
      H.tamaño = H.tamaño - 1
      H = PERCOLAR_ABAJO(H, 0)
      Devolver H

BORRAR(H: maxHeap, dato: tipoDato): maxHeap
      i = 0

      Mientras i < H.tamaño y H.arreglo[i] != dato:
            i = i + 1

      Si H.arreglo[i] == dato:
            H.arreglo[i] = H.arreglo[H.tamaño - 1]
            H.tamaño = H.tamaño - 1
            H = PERCOLAR_ABAJO(H, i)

      Devolver H

PERCOLAR_ARRIBA(H: maxHeap, i: entero): maxHeap
      padre = PADRE(H, i)

      Mientras H.arreglo[i] > H.arreglo[padre]:
            auxiliar = H.arreglo[i]
            H.arreglo[i] = H.arreglo[padre]
            H.arreglo[padre] = auxiliar
            i = padre
            padre = PADRE(H, i)

      Devolver H
```

ALGORITMO 8.1 (continuación): Implementación de un *max-heap*.

```
PERCOLAR_ABAJO(H: maxHeap, i: entero): maxHeap
      cambios = verdadero

      Mientras cambios == verdadero y i < H.tamaño:
            cambios = falso

            hijoIzquierdo = HIJO_IZQUIERDO(H, i)
            hijoDerecho = HIJO_DERECHO(H, i)

            Si hijoIzquierdo != NULO y H.arreglo[i] < H.arreglo[hijoIzquierdo]:
                  cambios = verdadero
                  auxiliar = H.arreglo[i]
                  H.arreglo[i] = H.arreglo[hijoIzquierdo]
                  H.arreglo[hijoIzquierdo] = auxiliar
                  i = hijoIzquierdo
        Sino si hijoDerecho != NULO y H.arreglo[i] < H.arreglo[hijoDerecho]:
                  cambios = verdadero
                  auxiliar = H.arreglo[i]
                  H.arreglo[i] = H.arreglo[hijoDerecho]
                  H.arreglo[hijoDerecho] = auxiliar
                  i = hijoDerecho

      Devolver H

A_HEAP(arreglo: tipoDato[], i: entero, tamañoHeap: entero):
      hijoIzquierdo = 2 * i + 1
      hijoDerecho = 2 * i + 2

      Si hijoIzquierdo <= tamañoHeap y arreglo[hijoIzquierdo] > arreglo[i]:
            mayor = hijoIzquierdo
      Sino:
            mayor = i

      Si hijoDerecho <= tamañoHeap y arreglo[hijoDerecho] > arreglo[mayor]:
            mayor = hijoDerecho

      Si mayor != i:
            auxiliar = arreglo[i]
            arreglo[i] = arreglo[mayor]
            arreglo[mayor] = auxiliar
            A_HEAP(arreglo, mayor, tamañoHeap)

CREAR_HEAP(arreglo[]: tipoDato): maxHeap
      H: maxHeap
      i: entero
      tamañoArreglo = LARGO(arreglo)
      H.tamañoArreglo = tamañoArreglo
      H.tamaño = tamañoArreglo
      H.arreglo = arreglo

      Para i desde tamañoArreglo / 2 hasta = 0:
            A_HEAP(H.arreglo, i, tamañoArreglo)

      Devolver H
```

8.2 EL ALGORITMO HEAPSORT

Este algoritmo de ordenamiento toma un arreglo y lo trabaja como si fuese un *heap*. Como el máximo elemento del arreglo queda en la raíz del árbol, almacenada en la posición 0, puede ser situado en su posición definitiva intercambiándolo con el último elemento.

Si se descarta el último del *heap*, los hijos de la raíz siguen siendo *max-heaps*, pero la nueva raíz puede violar la propiedad de *max-heap*. En consecuencia, se debe restaurar dicha propiedad usando `A_HEAP(arreglo, i, tamañoHeap)`. Se repite el proceso anterior hasta trabajar con un *heap* de tamaño 2.

El algoritmo 8.2 detalla el funcionamiento de heapsort.

ALGORITMO 8.2: heapsort.

```
HEAPSORT(arreglo: tipoDato[], tamañoHeap: entero): tipoDato[]
      Para i desde tamañoHeap hasta 1:
            auxiliar = arreglo[i]
            arreglo[i] = arreglo[0]
            arreglo[0] = auxiliar
            tamañoHeap = tamañoHeap - 1
            A_HEAP(arreglo, 0, tamañoHeap)

      Devolver arreglo
```

BIBLIOGRAFÍA CONSULTADA

AHO, A. V.; HOPCROFT, J. E.; ULLMAN, J. D. (1987). *Data Structures and Algorithms*, Addison-Wesley, United States of America.

AITKEN, P.; JONES, B. (1994). *Aprendiendo C en 21 días*, Prentice-Hall Hispanoamericana, México.

BRASSARD, G.; BRATLEY, P. (1997). *Fundamentos de Algoritmia*, Prentice Hall, Madrid, España.

CORMEN, T. H. *et al*. (2009). *Introduction to Algorithms* (3rd ed.), The MIT Press, Cambridge, Massachusetts - London, England.

GUEREQUETA, R.; VALLECILLO, A. (1998). *Técnicas de diseño de algoritmos*, Servicio de Publicaciones de la Universidad de Málaga, España.

JIMÉNEZ, J. A. (2009). *Matemáticas para la computación*, Alfaomega, México.

KOLMAN, B.; BUSBY, R. C. (1986). *Estructuras de Matemáticas Discretas para la Computación*, Prentice-Hall Hispanoamericana S. A., México.

SCHILDT, H. (1999) *C Manual de referencia* (3ª ed.), Osborne McGraw-Hill, Madrid, España.

VILLANUEVA, M. (2010). *Apuntes de la Asignatura Algoritmos Avanzados*, Departamento de Ingeniería Informática, Facultad de Ingeniería, Universidad de Santiago de Chile, Santiago, Chile.