Tipos Abstractos de Datos y Algoritmos

Emely Arraiz
Edelmira Pasarella
Cristina Zoltan
Dpto de Computación y Tecnología
de la Información
Universidad Simón Bolívar
Caracas, Venezuela
e-mail: arraiz,pasarella,zoltan@ldc.usb.ve

16 de enero de 2002

Índice General

¶Indice de Figuras

¶ndice de Tablas

Prefacio

Alcance del libro

El tema central de este libro es algoritmia. El conocimiento de los algoritmos cl¶sicos que se utilizan para resolver problemas en forma mecanizada constituye el coraz¶n de la disciplina, mitad arte y mitad tecnolog¶a, llamada **Inform¶tica**. La algoritmia es un tema que si bien est¶ normalmente asociado a los computadores, puede verse simplemente como una disciplina que se ocupa de conocer y poder comunicar en forma precisa y sin ambigÄedades mecanismos de soluci¶n de problemas.

Si bien el lenguaje de programaci¶n es el mecanismo de comunicaci¶n entre el hombre y la m¶quina, existen principios y modelos que han perdurado en el proceso de describir al computador lo que debe ejecutar para resolver un problema dado.

La evoluci¶n de los lenguajes de programaci¶n iniciaron su existencia apegados al computador mismo (lenguajes ensambladores) y fueron evolucionando hasta llegar a la categor¶a de lenguajes portables, independientes del computador que ejecutar¶ el algoritmo a tal punto que podremos asegurar tener los mismos resultados independientemente del hardware utilizado.

En la primera etapa, se debi¶ traducir el problema a resolver a la estructura del computador que lo resolver¶a. En la actualidad, se modelan los problemas, se modelan las soluciones y es posible en base a esos modelos expresar soluciones en el lenguaje de programaci¶n. En este proceso de expresar las soluciones se ha sentido la necesidad de disponer de bibliotecas y el elemento principal de las bibliotecas es tener mecanismos para asistir al lector a encontrar lo que busca en la biblioteca. La expresi¶n de la soluci¶n no es ¶nica, podemos hablar de familias de soluciones de un mismo problema, donde los diferentes miembros de la familia poseen caracter¶sticas que los hacen mas o menos aptos para la soluci¶n del problema particular que se esta tratando de mecanizar, aptitud que depender¶a de consideraciones ambientales. Es en este punto donde se evidencia la necesidad de conocer en forma precisa los elementos (componentes de software), y con ese conocimiento poder elegir correctamente a ¯n de realizar la implementaci¶on de la soluci¶on al problema que nos ocupa.

Este libro se ocupa de presentar los elementos b¶asicos de la descripci¶on de algoritmos, los **T**ipos **A**bastractos de **D**atos (**TAD**), las propiedades en t¶erminos de su e¯ciencia para ser usados como la materia prima en la soluci¶on de problemas.

Uso del libro

Este libro est¶a destinado principalmente a estudiantes de Computaci¶on que posean algun conocimiento instrumental de un lenguaje de programaci¶on. Un curso usando este libro como texto, debe estar acompa˜nado de un taller en que los estudiantes realicen tareas con implementaciones de los tipos abstractos de¯nidos (TAD)[1].

El libro presenta un cat¶alogo de tipos abstractos, relacionandolos entre ellos [2]. Los TAD se presentan de manera precisa haciendo uso de axiomas que describen sus propiedades. Este libro cubre solamente parte de los modelos importantes dejando el modelo de grafos como material para un siguiente curso.

Las recetas de cocina han sido usadas innumerables veces como contra ejemplo de algoritmo. No es que lo sean en absoluto, solamente que en la literatura culinaria se parte de un cierto conocimiento b¶asico com¶un. Todas las personas que ingresan a la cocina por primera vez, provistas de un libro de cocina, con la certeza de poder realizar un plato suculento id¶entico a aquel que estan observando en la fotograf¶ia, suelen fracasar. En general los errores se deben a la interpretaci¶on de frases tales como Agregar un pizca de sal, o derrita 250 gramos de chocolate que son sumamente ambigÄuas ya que ni la sal ni el chocolate son id¶enticos en todos los lugares del mundo, sin mencionar la interpretaci¶on de pizca. El cocinero profesional interpreta mas acertadamente las frases de una receta, sin embargo la prueba varias veces antes de ofrecerla en una comida.

[1] Gu¶ia de taller de algoritmos y estructuras II, Reporte CI-1995-003, Pasarella et al. Dpto Computaci¶on y tecnolog¶ia de la informaci¶on Universidad Sim¶on Bol¶ivar, Caracas

[2] En el Ap¶endice B se da un esquema de esas relaciones

Los ingredientes del computista son sus tipos (abstractos o concretos), objetos con los que modela la realidad, y el algoritmo es el equivalente de la receta del cocinero, pero una receta de índole industrial: debe ser totalmente libre de ambigüedades que permita asegurar que el producto final tenga exactamente las propiedades que espera el consumidor. En la elaboración de un producto alimenticio industrial no hay lugar a frases del tipo Hoy resultó un poco salado.

Los tipos constituyen los bloque básicos, que deben conocerse a cabalidad, de manera de poder hacer uso de ellos con exacto conocimiento de sus fortalezas y debilidades.

El libro inicia con el Capítulo 1 donde se presenta una introducción básica a los modelos de análisis de algoritmos que ayudarán a hacer un estudio sistemático de los algoritmos presentados en el resto del libro. Cuando se realiza un desarrollo se debe conocer el comportamiento de cada uno de los componentes del mismo. Este conocimiento permitirá hacer optimizaciones en los puntos que resulten críticos.

En el Capítulo 2 se revisan los Tipos Concretos de Datos, que corresponden a los tipos de datos (objetos básicos) que suelen tener los lenguajes de programación, correspondiendo a los bloques mas elementales de los tipos de datos. Estos tipos concretos, varían entre los lenguajes de programación, sin embargo están presentes de una u otra forma en todos los lenguajes.

Los tipos mas elaborados, suelen ser definidos para la aplicación particular. En los lenguajes actuales, están representados por las librerías, que suelen ser librerías orientadas al tipo de problema que se está resolviendo como librerías de finanzas, librerías de juegos instruccionales, librerías de presentaciones.

En el capítulo 3 se introducen la forma en que serán presentados los tipos abstractos en el libro usando como ejemplo el tipo conjunto y multiconjunto como una variante del primero haciendo el enlace a las implementaciones de los TAD usando tipos concretos.

En el capítulo 4 se introducen un refinamiento del TAD multiconjunto, donde se organiza el conjunto asignando puestos a los elementos que lo constituyen.

En los Capítulos 5 y 6 se estudian dos problemas para los cuales el computador ha resultado muy eficiente: la búsqueda y el ordenamieto.

En los Capítulos 7 y 8 se presentan TAD mas elaborados que estan enfocados principalmente a algoritmos mas eficientes de búsqueda.

El apéndice A recoge ejemplos de las implementaciones de algunos tipos, mientras que el ap¶ndice B recoge la descripci¶n de las relaciones que existen entre los tipos abstractos revisados en este libro.

Para cada TAD, revisa una variedad de problemas que pueden resolverse facilmente usando el TAD, analizando los comportamientos de las soluciones, estableciendo cotas de e¯ciencia derivadas de la estrategia de soluci¶n y analizando el comportamiento de cada algoritmo de acuerdo a las implementaciones de los tipos abstractos. Cada cap¶tulo incluye ejercicios y bibliograf¶a.

Agradecimientos

El agradecimiento va principalmente a los estudiantes de Ingenier¶a de Computaci¶n que cursaron Algoritmos y Estructuras II durante varios años en la Universidad Sim¶n Bol¶var usando notas y fotocopias de fotocopias. El agradecimiento va para la Coordinaci¶n de Ingenier¶a de la Computaci¶n y al Departamento de Computaci¶n y Tecnolog¶a de la Informaci¶n de la Universidad Sim¶n Bol¶var que nos di¶ la oportunidad de dictar la asignatura pese a la oposici¶n de parte del profesorado. El tiempo hizo su trabajo.

Finalmente la gracias van para Donald Knuth y Leslie Lamport por haber puesto a LaTeX en manos de la gente. Gracias.

Capítulo 1

Introducción al Análisis de Algoritmos

1.1 Marco Conceptual

Dado que en general un problema que se desea resolver mediante el uso del computador tiene una colección de algoritmos que lo resuelven, es conveniente conocer el costo de cada uno de ellos y su comportamiento frente al conjunto de datos para seleccionar el algoritmo más apropiado al problema que se esté resolviendo. El conocimiento de familias de algoritmos que resuelven un mismo problema nos permite hacer la selección adecuada.

La evaluación del costo de un algoritmo se hace calculando cuánto demora (tiempo de ejecución) y/o cuánto espacio requiere. Según el problema a resolver, el costo dependerá de las características de los datos de entrada.

Muy frecuentemente, cuando se escribe un programa para resolver un problema, se hace con la intención de que éste sea utilizado muchas veces, por lo que resulta conveniente caracterizarlo según su tiempo de ejecución y la calidad de la respuesta. Cuando estudiamos algoritmos es muy importante caracterizar la solución que se obtiene de cada algoritmo a fin de estar seguros que dos algoritmos son equivalentes (para un mismo valor de entrada dan exactamente el mismo resultado) o son similares (pueden dar resultados diferentes, pero desde el punto de vista del problema que estamos resolviendo somos indiferentes a cualquiera de los resultados). Por esta razón uno de los criterios más importantes para seleccionar un programa es evaluar el tiempo que demora en ejecutarse, que de ahora en adelante llamaremos **costo del**

programa. Para analizar el costo de un algoritmo, adoptaremos un **modelo computacional**, que nos dice cual de los recursos usados por la implementación del algoritmo son importantes para su desempeño. La **complejidad** de un algoritmo bajo un modelo computacional es la cantidad de recursos en tiempo o espacio, que la implementación del algoritmo usa para resolver el problema dado. Por ejemplo en el algoritmo para hallar la potencia de un número, usando multiplicaciones, la cantidad de multiplicaciones es el recurso utilizado. El esquema del algoritmo es:

$$x^0 = 1$$

$$x^k = x \cdot x^{(k-1)} \quad ; k > 0$$

Vemos que el número de multiplicaciones que deba realizar el algoritmo está relacionado con k, y dependiendo de la implementación que se haga del mismo hará k o $k-1$ multiplicaciones.

Veamos 2 implementaciones de la función $Y = X^k$

```
If K = 0 then  Y:=1
 else
     begin
        Y:=1
         for i = 1 to K
        Y:=Y * X
     end
 fin
```

```
If K = 0 then  Y:=1
  else
     begin
        Y:=X
         for i = 2 to K
        Y:=Y * X
     end
 fin
```

1.2 Crecimiento asintótico

En general un algoritmo será utilizado repetidas veces y el tiempo que demore dependerá del computador en que se ejecute pero también de una medida del tamaño de los datos. En el ejemplo anterior de la potencia el tamaño que nos interesa es el valor de la potencia y no suele tener importancia el valor del número que estamos elevando a la potencia. Otro ejemplo de tamaño del problema, puede ser el encontrar el titular de una tarjeta de débito. El tiempo requerido puede depender del número total de tarjeta-habientes.

Para analizar un algoritmo y calcular su costo, se consideran los siguientes casos:

Peor caso: Los casos de datos de entrada que hacen demorar m¶s al algoritmo.

Mejor caso: Los casos de datos de entrada que hacer ejecutar m¶s r¶pidamente al algoritmo.

Caso Promedio: En esta medida tomamos en cuenta una posible distribuci¶n de los datos de entrada.

Debe notarse que en el primer (segundo) caso se caracterizan los datos de entrada que hacen demorar m¶s (menos) al algoritmo considerado. El tercer caso se basa en hip¶tesis sobre la distribuci¶n estad¶stica del conjunto de todos los valores de entrada que puedan ser ingresados al algoritmo. En el ejemplo del c¶lculo de la potencia de un n¶mero los 3 casos son iguales ya que para valores de entrada $x; k$ los 2 algoritmos hacen respectivamente k y k ¡ 1 multiplicaciones.

1.2.1 Ejemplo 1

Veamos un algoritmo para buscar un cierto valor en un conjunto y haremos el an¶lisis del mismo para los diversos tipos de entrada.

BUSCAR (dado un vector V de N elementos, busca si el elemento X esta o no en el vector)

```
1.esta =`no'
2. I = 1
3. while I <= N and (esta =`no')
 3.1 if V[I] = X
     then  esta    :=`'
     else  I := I + 1
4. write(`valor de', X, esta, ` fue encontrado')
5. fin
```

Observando el algoritmo vemos que el menor trabajo (menor ejecuci¶n de operaciones) lo realizar¶ cuando ingrese el menor n¶mero de veces al while de la l¶nea 3, independientemente del valor de N. Vemos que esto suceder¶ si la primera vez que verifica la condición en la línea 3.1 esta resulta verdadera, con lo cual el mejor caso para este algoritmo es que el valor buscado se encuentre en la primera posición del vector. Para este caso el algoritmo realiza una cantidad constante de operaciones (no depende de N).

Por analogía podemos ver que el peor caso es que ingrese N veces a la iteración, y en el último ingreso la condición de la línea 3.1 sea verdadera. Eso significa que los valores que hacen que el algoritmo realice la mayor cantidad de trabajo es un vector en el cual el valor buscado esta en la última posición. Para este caso el algoritmo realiza una cantidad de operaciones proporcional al tamaño del vector (c*N).

Para hacer el análisis del caso promedio usaremos la hipótesis que cualquier vector tiene la misma posibilidad que cualquier otro. Nos preguntaremos cuántos vectores distinguibles podemos tener. En un vector de N posiciones el valor buscado puede estar en cualquiera de ellas o no estar. Por lo tanto debemos considerar N+1 casos. Para los casos en que el elemento buscado está, el trabajo realizado es proporcional a la posición donde se encuentra el elemento, y para el caso en que no se encuentra es prácticamente igual al caso de encontrarse en el último puesto. Por lo tanto sumaremos la función que describe el comportamiento en cada uno de los casos y lo dividiremos por el número total de casos.

$$\left(\sum_{i=1}^{N} c \cdot i + c \cdot N\right)/(N+1) = c \cdot N(N+3)/(2 \cdot (N+1))$$

A través de las medidas tenemos información sobre el comportamiento esperado de un algoritmo, dependiendo de la entrada. Lo que debemos plantearnos cuando queremos desarrollar una aplicación particular, es identificar un algoritmo eficiente en tiempo de ejecución, por lo que requerimos conocer el perfil de los datos de entrada. Este proceso de selección también implica conocer una variedad de algoritmos que resuelvan el problema dado.

Hasta ahora hemos hablado de conjuntos de algoritmos que resuelven un mismo problema. Ahora usaremos otro criterio de organizar conjunto de algoritmos: pertenecer a una misma clase de complejidad.

Definición: f es $O(g)$. Sean f, g dos funciones con dominio en los naturales (correspondiente al tamaño de la entrada) diremos que "f es $O(g)$" (que se lee f es de orden g) o "$f = O(g)$" (f es orden g) si existen constantes positivas c y n_0 tal que $f(n) \leq c \cdot g(n)$ para todo $n \geq n_0$.

Definición: f es $\Theta(g)$ Definimos el conjunto de funciones $\Theta(g)$ (conjunto de funciones del mismo orden o de igual complejidad) como todas aquellas funciones f tales que $f = O(g)$ y $g = O(f)$.

Definición alterna: Dos funciones f y g son del mismo orden si

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

para $c > 0$.

1.2.2 Ejemplo 2

En el ejemplo del cálculo de la potencia, los dos algoritmos son de igual complejidad. Sea f la función de complejidad para uno de ellos y g la función de complejidad del otro. Del análisis resulta que:

$$f(k) = k$$

$$g(k) = k - 1$$

por lo que $\lim_{k \to \infty} (f(k)/g(k)) = \lim_{k \to \infty} \frac{k}{k-1} = 1$ siendo $c = 1$ y $c > 0$.

1.2.3 Ejemplo 3

Para el caso de la búsqueda de un elemento en un vector, vimos que el caso promedio resultó la expresión $c \cdot N(N+3)/(2 \cdot (N+1))$, que es de orden N. Si considemos

$$f(N) = c \cdot N(N+3)/(2 \cdot (N+1))$$

y

$$g(N) = N$$

tendremos que

$$\lim_{k \to \infty} ((c \cdot N(N+3)/(2 \cdot (N+1)))/N) = \lim_{k \to \infty} \frac{c}{2} \cdot \frac{N+3}{N+1} = \frac{c}{2}$$

por lo que tenemos que la búsqueda de un elemento cualquiera en un vector de N elementos es de $O(N)$.

1.3 Algebra de $O(g)$

En general, por la forma de expresión de los algoritmos, podemos calcular el costo de una parte y luego utilizar el álgebra de órdenes para el cálculo del orden de un algoritmo particular.

Podemos dar algunas reglas para el formalismo de $O(g)$, reglas que se pueden derivar de la definición (y dejamos al lector su demostración) que pueden sernos útiles en el estudio de crecimiento de ciertas funciones.

$$\begin{array}{rcl} f(n) & = & O(f(n)); \\ c \cdot O(f(n)) & = & O(f(n)) \\ O(O(f(n))) & = & O(f(n)) \\ O(f(n)) \cdot O(g(n)) & = & O(f(n) \cdot g(n)) \\ O(f(n) \cdot g(n)) & = & f(n) \cdot O(g(n)) \end{array}$$

Otras ecuaciones útiles son

$$\begin{array}{rcll} n^p & = & O(n^m); & \text{donde } p \leq m \\ O(f(n)) + O(g(n)) & = & O(|f(n)|) + O(|g(n)|) & \end{array}$$

1.4 Complejidad de familias de algoritmos

En lo que concierne a la complejidad, podemos atacar como problema encontrar la complejidad de un algoritmo particular o encontrar cotas para la complejidad de una familia de algoritmos que resuelven un problema dado. En el primer caso, nuestro punto de partida es un algoritmo particular concreto, mientras que en el segundo tomamos un problema y analizamos la familia de algoritmos que lo resuelven, pudiendo llegar a conclusiones tales como:

> Sea un computador de una cierta velocidad de cómputo v, para todo algoritmo que resuelve el problema P, de tamaño n, el tiempo requerido para su solución es mayor que n^2. Veamos un ejemplo:
>
> Es bien conocido el algoritmo de buscar el máximo de una colección de elementos. Pero si nos planteamos buscar el máximo de una colección de elementos que no están estructurados entre ellos, sabemos que debemos revisarlos a todos para estar seguros que el elemento que demos es el máximo de la colección. Entonces, si la colección es de tamaño n,

tendremos que revisarlos a todos antes de dar una respuesta, diremos que la funci¶on de costo $f(n) = O(n)$.

En el ejemplo anterior vemos que pudiera haber algoritmos mejores si estructuramos los datos.

Veremos ahora el caso de los algoritmos de clasi¯caci¶on que utilizan como modelo de computaci¶on las comparaciones (y los intercambios) para lograr tener una presentaci¶on ordenada de N valores de entrada. En la ¯gura 1.2 se muestra un algoritmo en el que se usan comparaciones entre los elementos x,y,z para decidir cu¶al permutaci¶on de estos elementos da una secuencia ordenada. En cada c¶ırculo se dan los pares de elementos que se comparan y cada salida de las comparaciones est¶a etiquetada por el resultado de la comparaci¶on. Finalmente, en los nodos sin salida se indica la permutaci¶on que corresponde a los elementos ordenados.

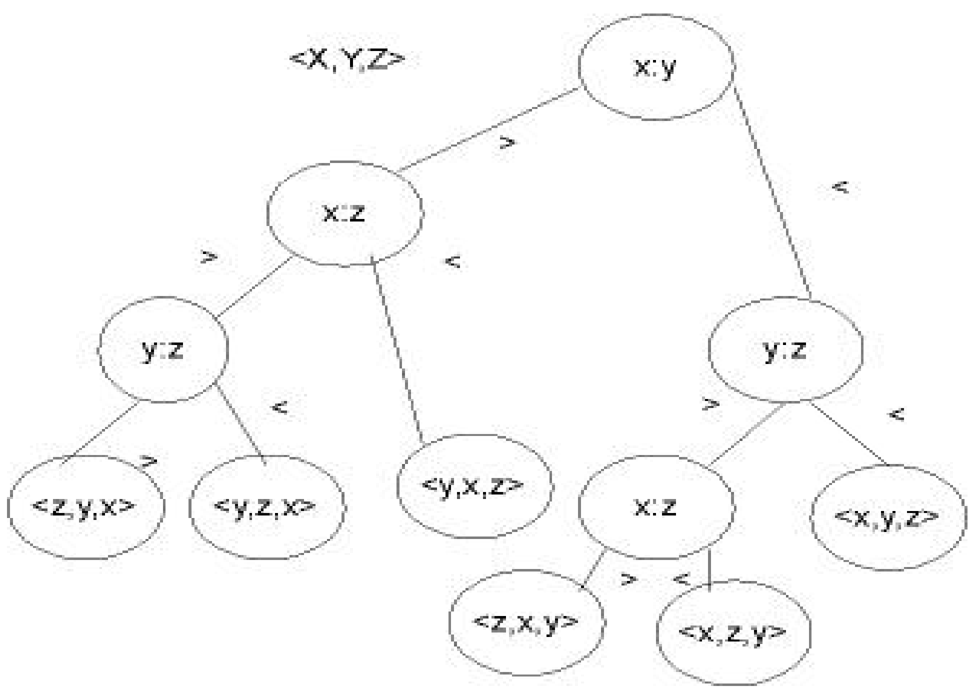


Figura 1.2: Algoritmo de ordenamiento

Supongamos inicialmente que todos los valores de entrada son diferentes. Por lo tanto, la variedad posible de entradas diferentes en cuanto al desorden son todas las permutaciones de N elementos (N!), una sola de las cuales está ordenada. Cada comparación tiene dos resultados posibles, por lo que después de k comparaciones tendremos 2^k resultados posibles. Para que estos resultados sean al menos el número de casos posibles deberá verificarse

$$2^k \geq (N!)$$

$$k \geq \ln N!$$

y por la fórmula de Stirling resulta que

$$k \geq C * (N \ln N)$$

.

Como vemos en el ejemplo anterior, cualquier algoritmo de clasificación que use comparaciones para lograr su objetivo, necesariamente requerirá hacer un número de comparaciones mayor o igual que $N \ln N$ El alcance de este libro se limita al estudio de problemas cuyos algoritmos tienen una complejidad dada por funciones del tipo N^k, llamadas funciones polinomiales. Un tratamiento mas extenso puede encontrarse en la referencia bibliográfica 7.

La tabla 1.1 se muestra el número de operaciones requeridas según la función de complejidad del algoritmo, indicada en las filas y el tamaño de la entrada, indicada en las columnas.

En particular existe una gran variedad de problemas para los que no se conoce algoritmos eficientes. Para ampliar la idea de la importancia de la eficiencia de los algoritmos elegidos para la solución de un problema consideraremos una aplicación desarrollada para un establecimiento pequeño, supongamos que manejan 10 artículos. Esta aplicactón incluye un algoritmo de $O(n^3)$ para un procesamiento de los 10 artículos. El negocio crece, aumentando el número de artículos, y aspirando poder manejar 1.000.000 de artículos. Suponiendo que el computador en que se realiza el procesamiento requiere 1 nanosegundo (un millardésimo de segundo), para cada operación, el algoritmo de procesamiento de artículos tardaría aproximadamente 32 años.

	10	50	100	300	1000
5N	50	250	500	1500	5000
$N \times \log_2 N$	33	282	665	2469	9966
N^2	100	2500	10000	90000	1 millón (7 dígitos)
N^3	1000	125000	1 millón (7 dígitos)	27 millones (8 dígitos)	1 millardo (10 dígitos)
2^N	1024	un número de 16 dígitos	un número de 31 dígitos	un número de 91 dígitos	un número de 302 dígitos
N!	3.6 millones de 7 dígitos	un número de 65 dígitos	un número de 161 dígitos	un número de 623 dígitos	muy grande
N^N	10 millardos de 11 dígitos	un número de 85 dígitos	un número de 201dígitos	un número de 744 dígitos	muy grande

Tabla 1.1: Algunas funciones y sus valores

1.5 Resumen del capítulo 1

En este capítulo hemos revisado los elementos que nos permiten evaluar el costo de un cierto algoritmo. Es muy importante tener claro el costo de las operaciones que usamos para resolver mecánicamente un problema. Cuando se desarrolla una aplicación, los programadores parecen dividirse en dos actitudes extremas:

- "No debo preocuparme por la eficiencia de mi solución, ya que bastará que se disponga de un computador suficiente potente para soportar esta aplicación".

- "Cada línea de código debe ser revisada para optimizar el tiempo de la aplicación".

Infortunadamente ninguno de los dos extremos es saludable ya que las mayoría de las aplicaciones suelen obedecer a la regla del 90-10, que dice que el 90% del tiempo de ejecución se consume en el 10% del código. Siendo extremos, si optimizamos con el fin de reducir a la mitad el tiempo de ejecución, sobre la parte del código que corresponde al 10% de la ejecución, habremos tenido una ganancia neta del 5%, lo cual no parece efectivo dado el costo invertido en la optimización.

Si embargo el material de este cap¶tulo nos permite saber cu¶les procesos de complejidad alta deben tratar de excluirse de ese c¶digo que se utiliza 90% del tiempo de ejecuci¶n de la aplicaci¶n.

1.6 Ejercicios

1. Haga un algoritmo que encuentre al individuo m¶s alto del mundo y calcule la complejidad del algoritmo obtenido.

2. Realice una implementaci¶n del problema de las torres de Hanoi, que reciba como par¶metro el n¶mero de discos y que calcule el tiempo desde el inicio hasta el ¯n de la ejecuci¶n del algoritmo. Usando la implementaci¶n realizada, haga una tabla de la funci¶n n¶mero de discos, tiempo de ejecuci¶n.

3. Obtenga el orden de los tiempos de ejecuci¶n para el peor caso de los siguientes procedimientos:

   (a)
   ```
   procedure prod-mat(n:integer);
   var i,j,k: integer;
   begin
       for i:=1 to n do
           for j:=1 to n do begin
               c[i,j]:=0;
               for k:=1 to n do
                   c[i,j]:= c[i,j]+a[i,k]*b[k,j]
           end
   end;
   ```

   (b)
   ```
   procedure misterio(n:integer);
   var i,j,k:integer;
   begin
       for i:=1 to n-1 d
           for j:= i+1 to n do
               for k:=1 to j do
                   p; /* p es O(1) */
   end;
   ```

4. Ordene las siguientes funciones de acuerdo a su velocidad de crecimiento:

   (a) n
   (b) $\sqrt{n}$
   (c) $\log n$
   (d) $\log\log n$
   (e) $n \log n$
   (f) $(3/2)^2$
   (g) n^n

5. Agrupe la siguientes funciones de acuerdo a su complejidad:

   (a) $n^2 + \log(n)$
   (b) $n!$
   (c) $\ln(n)$
   (d) $\log(n)$
   (e) n^2

6. Muestre que las siguientes afirmaciones son verdaderas:

   (a) 17 es $O(1)$
   (b) $n(n - 1)/2$ es $O(n^2)$
   (c) $\max(n^3, 10n^2)$ es $O(n^3)$
   (d)
   $$\sum_{i=1} i^k$$
   es $O(n^{k+1})$ para k entero.
   (e) Si $P(x)$ es cualquier polinomio de grado k donde el coeficiente del término de mayor grado es positivo, entonces la complejidad de calcular el polinomio en n es $O(n^k)$

7. Para los siguientes pares de funciones de complejidad, correspondientes a los algoritmos A y B, respectivamente:

A	B
N^2	$100 \times N \times \log_2 N$
$N/2$	$4 \times \log_2 N$

   (a) Encontrar el mínimo entero positivo para el cual el algoritmo B es mejor que el algoritmo A. Justifique.

   (b) Indique el orden para cada función de complejidad dada. Justifique.

1.7 Bibliografía

1. AHO, Alfred, HOPCROFT, John & ULLMAN, Jeffrey. “Data Structures and Algorithms". Addison Wesley, 1985.

2. BAASE, Sara. “Computer Algorithms. Introduction to Design and Analysis". Addison Wesley, 1978.

3. BINSTOCK, Andrew & REX, John. “Practical algoritms for Programmers".Addison Wesley, 1995.

4. GRAHAM Ronald, KNUTH, Donald & PATASHNIK, Oren. “Concrete Mathematics". Addison Wesley, 1989.

5. TREMBLAY, Jean-Paul & SORENSON, Paul. “ An Introduction to Data Structures with Applications". 2^{nd}. MacGraw Hill.

6. KALDEWAIJ, Anne. “Programming: The Derivation of Algorithms". Prentice Hall International Series in Computerr Science 1990.

7. BORODIN,A.B., MUNRO, I “Programming: The Derivation of Algorithms".American Elsevier, New York 1985.

Cap¶tulo 2

Tipos Concretos de Datos

2.1 Marco Conceptual

Los algoritmos generalmente manipulan datos, que poseen propiedades que son explotadas por esos algoritmos. Llamaremos tipo de datos a una clase de la cu¶l conocemos sus caracter¶sticas y la forma de manipularla. En general, un tipo de datos consiste de un rango de valores y las operaciones que pueden realizarse sobre ellos. Clasi¯caremos los tipos en Concretos y Abstractos.

Un tipo concreto es un tipo provisto por el lenguaje de programaci¶n con que se implementa el algoritmo y de ellos hablaremos en la pr¶ximas secciones. Estos tipos, seg¶n los requerimientos de memoria para representar los valores, se dividen en simples y estructurados. Los tipos abstractos de datos, los de¯niremos en el siguiente cap¶tulo.

2.2 La Noci¶n de Variable

Cuando resolvemos un problema algoritmicamente y pretendemos implementar las soluci¶n en un computador, nos encontramos con la necesidad de almacenar los datos que utiliza el algoritmo, por ejemplo, los datos de entrada, de salida e intermedios (privados). Es por esto que surge la noci¶n de **variable** en computaci¶n como una caja donde se pueden almacenar datos. F¶sicamente una variable corresponde a una o varias localidades de la memoria del computador. Para referirnos a las variables en un algoritmo o programa, podemos identi¯carlas con un nombre llamado **identi¯cador** o estar en capacidad de calcular el identi¯cador correspondiente. Durante la ejecución de un programa, los datos almacenados en las variables corresponden a los valores de las mismas.

Generalmente, para acceder al valor de una variable basta con nombrarla por su identificador. Este método de acceso se llama acceso inmediato o directo. Existen otros métodos de acceso que mencionaremos en secciones posteriores. El contenido (valor) de una variable puede cambiar potencialmente en cada paso del algoritmo.

2.2.1 Asignación

La asignación es un operador binario que permite cambiar el valor de una variable y frecuentemente [1] se denota como sigue:

```
v := Expresión
```

La semántica de la asignación es almacenar en la variable v el resultado de evaluar `Expresión`. Nótese que el argumento de la izquierda debe ser una variable. Algunos ejemplos de asignación son los siguientes:

1. `v := 2`
2. `v := x + y`
3. `v := sqr(a) - log(b)`
4. `v := v + 1`

En una asignación las variables son interpretadas posicionalmente. El valor de la variable que ocurre a la izquierda del operador `:=' puede ser diferente antes y después de ejecutarse la asignación mientras que los valores de las que ocurren a la derecha se mantienen iguales, sólo son leidos y usados para calcular el valor que debe almacenarse en la variable de la izquierda.

Una variable desde el punto de vista computacional es diferente a una variable en matemáticas donde representa un valor desconocido de un dominio en una expresión. En computación una variable es un objeto que cambia de estado a través de la operación de asignación. Analicemos lo que hace la

[1]Historicamente se usó el operador = , luego a partir de ALGOL 60 := y JAVA usa nuevamente el operador =, pero con un significado totalmente distinto.

asignaci¶on del ¶ultimo ejemplo: lee el valor de v antes de ejecutarse la asignaci¶on, lo incrementa en uno y el resultado lo almacena en v. En matem¶aticas una expresi¶on an¶aloga con la igualdad no tiene sentido.[2]

Dependiendo del lenguaje de implementaci¶on, el dominio de valores que puede tomar una variable se ¯ja antes de la ejecuci¶on (de¯nici¶on est¶atica). Ese dominio de valores se denomina el tipo de la variable y en estos casos las variables se clasi¯can seg¶un sus tipos . Tambi¶en es posible de¯nir el tipo de una variable durante la ejecuci¶on (de¯nici¶on din¶amica) pero estos casos est¶an fuera del alcance de este libro. De ahora en adelante nos referiremos a variables con tipos de¯nidos est¶aticamente.

Para que una asignaci¶on est¶e bien de¯nida es necesario que la evaluaci¶on de la expresi¶on de la derecha d¶e un valor que pertenezca al tipo de la variable de la izquierda (consistencia de tipos).

2.3 Tipos Simples

Los tipos simples son aquellos cuyos valores s¶olo requieren una localidad de memoria.[3] Una variable de un tipo simple corresponde exactamente a una localidad (casilla,direcci¶on) de memoria. Entre los tipos simples m¶as frecuentes tenemos enteros, reales, l¶ogicos y caracteres.

2.3.1 Tipo Entero

El tipo entero de¯ne un subconjunto de Z. Generalmente es un rango [¡ maxent::maxent ¡ 1], donde maxent es una constante tan grande como lo permita los mecanismos de direccioonamiento del computador. Los lenguajes de programaci¶on permiten, en general, manejar diversa variedad de enteros: 2 bytes (16 bits), 4 bytes (32 bits), etc. Los valores correspondientes para maxent se muestran en la tabla 2.1.[4]

Par este tipo se de¯nen los operadores

+ Suma

[2]En particular esta expresi¶on se considera frecuente y diversos lenguaje han tratado de darle sintaxis mas compactas: en el lenguaje C es v += 1 y en JAVA puede escribirse v++

[3]Cuando nos referimos a localidad queremos expresar que su recuperaci¶on se hace mediante una sola direcci¶on de memoria, independientemente del espacio que ocupe

[4]Los lenguajes de programaci¶on tienden a independidarse de la estructura f¶³sica de los computadores, buscando la portabilidad de las aplicaciones desarrolladas.

Tamaño de almacenamiento	Maxent
8 bits	128
16 bits	32768
32 bits	2147483648
64 bits	9223372036854775808

Tabla 2.1: ENTEROS

- Resta

* Multiplicación

div División Entera

mod Resto de una División Entera

Además en las especificaciones del lenguaje se establece un orden de evaluación entre los operadores para evitar ambigüedades al evaluar expresiones como:

2*3/1+2

cuyo resultado podría ser: 2, 8 o 10.

2.3.2 Tipo Real

Dado a que el tamaño de la memoria de un computador es finito, el conjunto de reales que se puede representar también es finito. Para representar a los reales se utiliza lo que se conoce como representación de punto flotante en la cual cada número real r es representado como un par $< e, m >$ siendo e y m enteros tal que

$$r = m \times B^e \quad -E < e < E \quad -M < m < M$$

[5] donde m es la mantisa, e es el exponente, B es la base de la representación punto flotante (en general se utiliza como base una potencia pequeña de 2) y E y M son constantes características de la representación que dependen

[5] Estas condiciones son aproximadas y dependen de la arquitectura particular del computador

generalmente del hardware. Sin embargo se ha establecido un estandard llamado IEEE 754 para el tipo punto flotante.

El tipo real provee todos los operadores aritméticos mencionados en la sección anterior pero hay que tener en consideración que debido a la representación dejan de cumplirse algunas propiedades aritméticas conocidas. Estos tipos numéricos suelen estar apegados a la arquitectura del computador. Los lenguajes de programación dejaban sus semántica sin definir, por lo que programas escritos en un mismo lenguaje podían tener una comportamiento diferente. [6]

Par este tipo se definen los operadores

+ Suma

- Resta

* Multiplicación

 División

2.3.3 Tipo Lógico o Boolean

La variables del tipo lógico pueden tomar valores de verdad true o false [7]. El tipo provee operaciones lógicas básicas, como por ejemplo:

and ($\wedge$)

or ($\vee$)

not ($\neg$)

2.3.4 Tipo Carácter

Este tipo define un conjunto ordenado de caracteres codificados. Inicialmente los computadores usaron el código ASCII (American Standard Code for Information Interchange). Los caracteres ASCII se dividen en imprimibles y no imprimibles. Originalmente eran $128 = 2^7$ caracteres por lo que sólo

[6] Algol 68 define los tipos por primera vez y luego este detalle es olvidado para ser retomado por JAVA.

[7] Algunos lenguajes permitían una interpretación entera de estos valores. Esto da lugar a errores de programación

se requer¶a 7 bits para almacenar un car¶acter en memoria. Luego se us¶o la tabla ASCII extendida, 256=2^8 caracteres, que requiere 8 bits. En la actualidad, dado que los software deben estar preparados para el manejo de varios idiomas, se utiliza el c¶odigo UNICODE que utiliza 16 bits para almacenar un car¶acter.

Los operadores que provee son los siguientes:

ord(c), que retorna el n¶umero que ocupa el car¶acter c en la tabla UNICODE

char(n), que retorna el car¶acter que ocupa la posici¶on n en la tabla UNICODE

succ(c), pred(c), sucesor y predecesor, respectivamente, del car¶acter c.

La posiciones de los caracteres en la tabla UNICODE establecen un orden entre ellos.

2.4 Referencias

Las referencias son una familia de tipos cuyos valores nos permiten \referirnos" a elementos de un determinado tipo. Es por eso que decimos que son una familia ya que podemos de¯nir referencias para cada uno de los tipos de¯nidos.

Dado que las referencias no poseen una aritm¶etica son pocas las operaciones que pueden realizarse con ellos[8].

Los operadores que provee son los siguientes:

a := b la asignaci¶on de referencias de un mismo tipo

En la siguiente secci¶on se ver¶a otros operadores para las referencias [9]

[8] Algunos lenguajes admiten una aritm¶etica para las referencias, sin embargo es un recurso que puede introducir inestabilidad en los programas

[9] Este tipo de datos se introdujo en ciertos lenguajes argumentando que se pod¶{a lograr programas mas e¯cientes. En general su uso se convirti¶o en fuente de errores. JAVA utiliza este concepto internamente y no es un tipo del lenguaje.

2.5 Tipos Estructurados

Los tipos estructurados son aquellos cuyos valores poseen una estructura interna y requieren más de una casilla de memoria para su representación.

2.5.1 Arreglos

Los arreglos constituyen un tipo estructurado, siendo una característica que todos los elementos que los componen son del mismo tipo. Por lo que al definir un arreglo deberá indicarse el tipo de los elementos que lo componen (tipo base): enteros, reales, referencias a reales, etc y el rango de enteros con los que se identificarán los elementos que lo componen. Cuando el número de elementos es fijo se denominan arreglos estáticos y cuando este número puede variar de denominan arreglos dinámicos. El rango de valores es por lo tanto una n-tuplas (donde n es el numero de elementos del arreglo) de elementos tipo base.

Algunos ejemplos de tuplas son los siguientes:

1. (2,3,9,0,12,32) un arreglo de seis componentes de enteros.

2. (2.7,67.5) un arreglo con dos componentes reales.

Otra característica de este tipo es el acceso directo a cada una de sus componentes, denominadas elementos de un arreglo. La implementación de los arreglos en los lenguajes de programación se realiza en general reservandole un bloque de memoria para su almacenamiento.

Si un arreglo ha sido definido teniendo un rango de valores entre $[i; j]; i \leq j$ entonces $\forall k \; i < k < j$ el k-ésimo elemento del arreglo estará ubicado en memoria entre el elemento $k - 1$ y el elemento $k + 1$.

Los operadores que provee son los siguientes:

`A[i]`, que retorna el valor de la i-esima componente del arreglo, cuando es usado en una expresión. Esto permite usar todo los operadores del tipo base.

`A[i] := expresion` , que asigna el valor de la expresión a la i-ésima componente del arreglo.

Algunos lenguajes de programación asocian los arreglos a vectores y por extensión a matrices de más de una dimensión y proveen las operaciones asociadas a esos tipos matemáticos: suma , multiplicación, producto escalar, producto vectorial, inversa de una matriz, etc.

2.5.2 Registros

Los registros constituyen un tipo estructurado, que a diferencia de los arreglos no requieren que sus componentes sean del mismo tipo. Otro elemento que lo caracteriza es que las componentes no se identi¯can por la posici¶on como en los arreglos, sino que se identi¯can por nombre y por lo tanto pueden no almacenarse en posiciones contiguas.

Un registro posee un nombre y cada componente a su vez tiene nombre. La variedad de valores puede asociarse a tuplas de pares ordenados, donde el primer elemento de cada par es el identi¯cador de la componente y el segundo su valor.

Algunos ejemplos de tuplas son los siguientes:

1. ((nombre,J),(apellido,P), (c¶edula,12867456))
   registro que pudiera llamarse persona de tres componentes: nombre apellido y c¶edula.

2. ((peso,2.7),(altura,67.5),(identificador,e))
   registro con tres componentes, dos de ellas reales (peso y altura) y la tercera de tipo car¶acter denominada identi¯cador.

Los operadores que provee son los siguientes:

persona.nombre, que retorna el valor de nombre del registro cuando es usado en una expresi¶on. Esto permite usar todo los operadores del tipo base.

persona.apellido := expresion , que asigna el valor de la expresi¶on a la componente apellido del registro per sona.

2.5.3 Listas

Las listas son tipos estructurados, pudiendo ser homogeneas (todos sus elementos son del mismo tipo), o no homogeneas. Una caracter¶istica de las listas es que los elementos poseen ref erencias a elementos de la lista como parte del elemento. En contraposici¶on a los arreglos, donde el operador para acceder a un elemento (o al siguiente en un recorrido) es ¯jo y asociado a la estructura de datos, en las listas un elemento posee como parte de s¶i mismo la ref erencia al siguiente. Es de hacer notar que esta noci¶on de siguiente no es ¶unica, puede haber varios siguientes.

En un arreglo A de enteros, de una dimensión, el elemento contiguo del elemento $A[i]$ es el elemento $A[i+1]$, lo que corresponde a una ley constante de acceso al elemento siguiente. En las listas, se levanta esta rigidez haciendo que el elemento de la lista posee como parte de sí mismo la referencia a un siguiente (o a varios siguientes). Siguiendo con la comparación con los arreglos, si un arreglo A ha sido definido teniendo un rango de elementos entre $[i; j]$, sabemos que el primer elemento de la estructura será $A[i]$ y el último $A[j]$. En las listas no hay direccionamiento directo a los elementos de la lista, deberá existir un mecanismo de acceso al primer elemento de la lista (generalmente conocida como la cabeza de la lista) y cada elemento de la lista dirá cual es el siguiente. Deberemos disponer de un valor de la referencia para indicarnos que no existe el siguiente (valor NIL) [10].

2.6 Resumen del capítulo 2

En este capítulo revisamos los tipos concretos que proveen los lenguajes de programación. Se debe tener presente que si bien los lenguajes parecen proveer los tipos básicos que usamos en matemáticas, en realidad de lo que se dispone en los lenguajes es de un modelo finito de los objetos infinitos tales como los enteros y los reales. En particular, propiedades de los objetos originales no se conservan en estos modelos finitos, propiedades tales como la asociatividad o la conmutatividad. Esto impone restricciones en el desarrollo de las aplicaciones y no tenerlo en cuenta suele ser el origen de problemas en la etapa de validación de una aplicación. Una aplicación puede validarse en un ambiente, pero con el tiempo puede haber variaciones que hagan perder la validez de las hipótesis usadas en la validación. Como ejemplos de estos problemas podemos mencionar la aparición de discos duros de gran tamaño, el problema de Y2K, la inflación, etc. El factor común de todos estos problemas es el modelo de datos utilizado.

[10] En particular JAVA no posee referencias y por lo tanto no puede manejar listas como se han presentado en esta sección. Sin embargo si la estructura de lista se requiere para alguna implementación particular siempre se pueden implementar usando arreglos y siendo las referencias posiciones dentro del mismo. Este esquema hace relocalizable los códigos, mientras que con referencias no necesariamente lo son

2.7 Ejercicios

1. Sea § un operador unario cuyo ¶nico operando es un arreglo. La interpretaci¶n de este operador es tal que devuelve la suma de todos los elementos del arreglo. Utilice su lenguaje de programaci¶n favorito para demostrar que diferentes implementaciones de este operador pueden dar resultados distintos al aplicarlo al mismo argumento. Comience con arreglos de reales, sabiendo que la operaci¶n de suma de reales es asociativa y conmutativa. Demuestre con sus implementaciones que estas propiedades no son v¶lidas en la suma de reales representables en el computador.

2. Realice al menos dos implementaciones del mismo operador para arreglos de enteros y demuestre que la suma de enteros en el computador tampoco satisface la propiedad de asociatividad y conmutatividad.

3. Dado como par¶metro la ubicaci¶n del primer elemento de un arreglo de dos dimensiones, diseñe el algoritmo para obtener la ubicaci¶n del elemento con sub¶ndices i, j.

2.8 Bibliograf¶a

1. FLANAGAN, D. \Java in a nutshell". O'Reilly. Segunda edici¶n. pp 23-33. 1997

2. HAREL, D. \Algoritmics, The spirit of Computing". Addison Wesley. S, pp 36-37. 1997

Capítulo 3

Tipos Abstractos de Datos (TAD). Representaciones

3.1 Marco Conceptual

Supongamos que queremos resolver un problema bastante complejo desde el punto de vista tanto de su funcionalidad como de la organización de sus datos. Al tratar de dar una solución nos encontramos con que hay que controlar demasiados detalles y esto hace que la tarea sea realmente ardua. En estos casos, es conveniente reducir la complejidad o los detalles que van a ser considerados a un mismo tiempo. Una manera de lograr esto es a través de un proceso de abstracción: abstracción procedural y abstracción de datos.

La abstracción procedural se refiere a la definición de funciones abstractas (que se implementan como procedimientos) que separan el "qué" hace la función del "cómo" lo hace. Una técnica usualmente usada para hacer abstracción procedural es el refinamiento paso a paso.

La abstracción de datos, a diferencia de la procedural, permite la descripción de objetos abstractos. Es decir, se establecen las clases de objetos del problema, independiente de cómo estos objetos van a ser representados (obviando los detalles). Como producto de la abstracción de datos se especifican los nombres y los dominios de las operaciones asociadas a cada clase de objetos y se definen los significados abstractos de las mismas.

Un **Tipo Abstracto de Datos (TAD)** corresponde a una clase de objetos establecida en el proceso de abstracción de datos y viene dado por su especificación y una implementación que la satisfaga.

Cualquier aplicación que requiera utilizar TADs, puede hacerlo conociendo solamente las especificaciones; es decir, no es necesario esperar hasta que el tipo esté totalmente implementado ni es necesario comprender las implementaciones.

Para implementar un TAD se requiere que el lenguaje de implementación soporte encapsulamiento, es decir, que brinde la posibidad de crear unidades de programas donde se coloquen juntos la representación de los objetos y el código asociado a cada operación. Además, debe soportar ocultamiento de información que se refiere a la posibidad de permitir el uso de los operaciones sin que el usuario pueda ver la representación de los objetos ni el código de las operaciones.

Hay muchos lenguajes de programación que soportan TADs (CLU , ALPHARD, EUCLID, JAVA), sin embargo, es posible utilizar TADs como una técnica de programación cuando se programa con lenguajes como PASCAL que no tienen el soporte adecuado para implementarlos.

Es conveniente hacer la especificación de un TAD utilizando un lenguaje formal. Esto nos libera de las ambigüedades propias del lenguaje natural. El lenguaje utilizado para hacer la descripción de un TAD lo llamaremos formalismo para especificar TADs.

3.2 Especificación de TADs

En una especificación, las diversas partes que la componen contribuyen de forma diferente en la descripción de una clase de objetos que pueden crearse, transformarse o descomponerse.

Un componente de una especificación de un TAD son los operadores, dando en la sección titulada **sintaxis** la forma de escribirlos (la descripción de su sintaxis). Como son operadores, daremos cuál es el tipo de los operandos y del resultado (dominio y rango). Ejm:

$$\text{insertar} : \text{Multiconjunto} \times \text{Elemento} \Rightarrow \text{Multiconjunto}$$

donde insertar es el nombre del operador, y utilizando la notación de funciones, indicamos el producto cartesiano de los dominios de los operandos, seguido de una flecha ($\Rightarrow$) que separa el tipo del resultado que da el operador.

En el ejemplo que nos ocupa estamos indicando que insertar es un operador binario, cuyo primer argumento es un multiconjunto y el segundo un elemento. El resultado de la operación es un multiconjunto. De igual manera daremos la sintaxis de la operación coseno que transforma ángulos en reales en el intervalo [-1,1].

$$\mathrm{coseno} : \mathrm{Angulo} \Rightarrow [-1; 1]$$

Paralelamente a la sintaxis de los operadores se da una definición de los mismos; esto lo haremos en la sección titulada **semántica** que está constituída por un conjunto de proposiciones, en lógica de primer orden, llamadas axiomas.

Usualmente daremos algunas propiedades que nos permitan comprender y describir el significado del operador que estamos definiendo. En matemáticas, en ocasiones damos completamente la definición de una función (función total) y otras veces, se define parcialmente la función (función parcial). Por ejemplo, si definimos la función inversa sobre los racionales Q con la siguiente sintaxis:

$$\mathrm{inversa} : Q \Rightarrow Q$$

podemos expresar la semántica como sigue:

$$\forall q (q \in Q \wedge q \neq 0 \rightarrow \mathrm{inversa}(q) \cdot q = 1)$$

Hemos usado para definir el operador inversa el operador de multiplicación de los racionales y la identidad de los mismos. Vemos que hemos definido la función inversa parcialmente ya que no sabemos lo que pasa si $q = 0$. Debe notarse que esta definición no es constructiva ya que no nos da idea de cómo conseguir el valor que corresponde a $\mathrm{inversa}(q)$. Sin embargo, cualquier implementación debe dar un resultado que verifique el axioma $\mathrm{inversa}(q) \cdot q = 1$.

Para los fines de definir la semántica de de las operaciones de un tipo, podemos dejar indefinido el valor o completar la función con valores arbitrarios o convenientes a la implementación.

En las especificaciones presentadas en este libro veremos que con gran frecuencia que apelamos a definiciones incompletas, indicando que somos indiferentes a la forma en que se completen. No hay que olvidarse que en computación se deben evitar los casos de evaluación de operadores con argumentos para los cuales no están definidos. Esto puede lograrse de varias formas: estableciendo claramente las precondiciones de los programas asociados a cada operador a fin de que las aplicaciones se encarguen de filtrar sus argumentos antes de usar el operador, o haciendo un manejo de excepciones de los argumentos para los cuales el operador no está definido.

Podemos resumir diciendo que la especificación de un TAD T, es una terna

$$ESP_T = (D, O, E)$$

donde D es el conjunto de dominios, O el conjunto de operadores y E es el conjunto de expresiones de lógica de primer orden que constituyen los axiomas. Cuando describimos un TAD, daremos el dominio (D), en la parte titulada sintaxis se da la forma de los operadores y en la parte semántica se da el conjunto E.

TAD Conjunto Un conjunto es una colección de **Elementos** donde cada uno de ellos ocurre a lo sumo una vez. A continuación presentamos la especificación del TAD Conjunto. A algunos axiomas se les coloca un título que describe la intención del mismo.

TAD Conjunto[Elemento]

$$D = \{Boolean, Conjunto, Elemento\}$$

Sintaxis

$$\begin{array}{llcl}
vacio: & & \Rightarrow & Conjunto \\
esvacio: & Conjunto & \Rightarrow & Boolean \\
insertar: & Conjunto \times Elemento & \Rightarrow & Conjunto \\
eliminar: & Conjunto \times Elemento & \Rightarrow & Conjunto \\
pertenece: & Conjunto \times Elemento & \Rightarrow & Boolean
\end{array}$$

Semántica

$\forall c \in Conjunto, e, e_1, e_2 \in Elemento, e \neq_{elemento} e_1, e_1 \neq_{elemento} e_2,$

1. $esvacio(vacio) = true$
2. $esvacio(insertar(c, e)) = false$
3. $esvacio(eliminar(insertar(vacio, e), e)) = true$
4. $esvacio(c) \rightarrow pertenece(c, e) = false$
5. $pertenece(insertar(c, e), e) = true$

6. $pertenece(insertar(c, e), e_1) = pertenece(c, e_1)$

7. $pertenece(eliminar(insertar(vacio, e), e_1), e_2) = (e =_{Elemento} e_2)$

8. **Axioma de Conmutatividad**
   $\neg esvacio(c) \rightarrow pertenece(eliminar(insertar(c, e), e_1), e_2) =$
   $pertenece(insertar(eliminar(c, e_1), e), e_2)$

9. **Axioma de Unicidad**
   $\neg esvacio(c) \rightarrow pertenece(eliminar(c, e), e) = false$

Fin-Conjunto;

En la definición nos estamos refiriendo a Elemento donde identificamos a los objetos que constituirán un conjunto. Debe resaltarse que no estamos caracterizando a los elementos, en particular pueden ser cualquier tipo de elementos (enteros, reales, sillas, elefantes). Mas adelante veremos una restricción para estos elementos[1].

A continuación daremos una lectura de los axiomas que se utilizan para dar la semántica a los operadores.

La primera línea después de la palabra **Semántica**, describe los tipos de las variables y algunas propiedades, como por ejemplo, $e \neq_{Elemento} e_1$. Es de hacer notar que en esta línea también se indica que todas las variables que ocurren en los axiomas están cuantificadas universalmente, satisfaciendo las propiedades asociadas a las variables. Esto hace que los axiomas no se recarguen tanto.

Axiomas 1,2 y 3

- $esvacio(vacio) = true$
- $esvacio(insertar(c, e)) = false$
- $esvacio(eliminar(insertar(vacio, e), e)) = true$

Definen al operador esvacio indicando que aplicado al conjunto llamado vacio da como resultado true (axioma 1), al aplicarlo a cualquier conjunto que

[1] En especial podemos pedir que los elementos sean ordenables o siendo mas formales posean una relación de orden total entre ellos.

sea el resultado de una operación insertar, da como resultado false (axioma 2) y al aplicarlo al conjunto resultante de eliminar el único elemento de un conjunto unitario da como resultado true. Podemos concluir que esvacio es un predicado que distingue al conjunto vacio de los demás conjuntos ya que un conjunto al que se le insertó un elemento no es vacío[2].

Axiomas 4 y 5

- $esvacio(c) \rightarrow pertenece(c; e) = false$
- $pertenece(insertar(c; e); e) = true$

De manera similar a los anteriores, estos axiomas definen al predicado pertenece que nos indicará si un cierto elemento (segundo operando del operador pertenece), es miembro del conjunto (primer operando).

El axioma 4 indica que independientemente del elemento a buscar (e), es falso que sea miembro del conjunto vacio (corresponde al conocido axioma de conjuntos).

El axioma 5 indica que una vez que agregamos un elemento a un conjunto, este pertenece al conjunto.

Si bien puede pensarse que se ha definido completamente al operador pertenece, no se sabe que sucede cuando se le aplica, a un conjunto que es el resultado de agregar un elemento ($e_1 \neq_{Elemento} e$). Esto se define en los axiomas 6 y 7.

Axiomas 6 y 7

- $pertenece(insertar(c; e); e_1) = pertenece(c; e_1)$
- $pertenece(eliminar(insertar(vacio; e); e_1); e_2) = (e =_{Elemento} e_2)$

---

[2] Los axiomas 1 y 3 parecerían decir lo mismo, pero lo que establecemos en el axioma 3 es que las implementaciones no requiere verificar que $eliminar(insertar(vacio; e); e) = vacio$, dejándole mayor grado de libertad a la implementación.

Axioma 6 En este axioma se indica que si un conjunto (c) es el resultado de agregar un elemento (e), la operación pertenece sobre ese conjunto y e_1 (con la hipótesis $e_1 \neq_{Elemento} e$), será igual al operador pertenece aplicado al conjunto original (c), buscando al mismo elemento.

Axioma 7

- $pertenece(eliminar(insertar(vacio; e); e_1); e_2) = (e =_{Elemento} e_2)$

Dado que el operador eliminar no está definido para conjuntos vacíos y que $e \neq_{Elemento} e_1$, este axioma indica que el elemento e_2 está en el conjunto unitario considerado si es igual al único elemento del conjunto, es decir, igual a e. En este axioma aparece la restricción sobre los elementos ya que se requiere que los elementos puedan ser comparados con un operador de igualdad ($=_{Elemento}$). En los ejemplos usados anteriormente deberíamos poder comparar enteros, reales, sillas o elefantes.

Axioma 8

- **Axioma de Conmutatividad**
  $\neg esvacio(c) \rightarrow pertenece(eliminar(insertar(c; e); e_1); e_2) =$
  $pertenece(insertar(eliminar(c; e_1); e); e_2)$

Este axioma indica que siempre que el conjunto a considerar (c) no sea vacío, el orden en que se agregan los elementos no tiene importancia.

Axioma 9

- **Axioma de Unicidad**
  $\neg esvacio(c) \rightarrow pertenece(eliminar(c; e); e) = false$

Este axioma llamado de Unicidad, indica que la eliminación de un elemento (e), no deja rastros en el conjunto considerado. Es de hacer notar que en todos los axiomas se utiliza la igualdad de los booleanos (resultado de pertenece, o la igualdad de los elementos ($e =_{Elemento} e_2$)). No hay igualdades entre conjuntos, pues no hemos definido ese operador para el TAD. Esto significa que usando la definición de conjunto no podemos tener conjuntos cuyos elementos sean conjuntos.

A continuación, definiremos un tipo abstracto muy similar al de conjunto.

TAD Multiconjunto Un multiconjunto se define como una colección de **Elementos** donde cada uno de ellos puede ocurrir más de una vez. Debido a esta propiedad los conjuntos son un caso particular de multiconjuntos.

TAD $Multiconjunto[Elemento]$

$$D = \{Boolean, Conjunto, Elemento\}$$

Sintaxis

$$\begin{array}{lrcl}
vacio: & & \Rightarrow & Multiconjunto \\
esvacio: & Multiconjunto & \Rightarrow & Boolean \\
insertar: & Multiconjunto \times Elemento & \Rightarrow & Multiconjunto \\
eliminar: & Multiconjunto \times Elemento & \Rightarrow & Multiconjunto \\
pertenece: & Multiconjunto \times Elemento & \Rightarrow & Boolean
\end{array}$$

Semántica
$\forall c \in Conjunto, e, e_1, e_2 \in Elemento, e \neq_{elemento} e_1, e_1 \neq_{elemento} e_2,$

1. $esvacio(vacio) = true$
2. $esvacio(insertar(m, e)) = false$
3. $esvacio(eliminar(insertar(vacio, e), e)) = true$
4. $esvacio(c) \rightarrow pertenece(c, e) = false$
5. $pertenece(insertar(m, e), e) = true$
6. $pertenece(insertar(m, e), e_1) = pertenece(m, e_1)$
7. $pertenece(eliminar(insertar(vacio, e), e_1), e_2) = e =_{Elemento} e_2$
8. **Axioma de Conmutatividad**
   $\neg esvacio(m) \rightarrow pertenece(eliminar(insertar(m, e), e_1), e_2) = pertenece(insertar(eliminar(m, e_1), e), e_2)$

9. **Axioma de No-Unicidad**
   pertenece(m; e) ! pertenece(eliminar(insertar(m; e); e); e) = true

Fin-Multiconjunto;

N¶otese que en las especi¯caciones de los TAD's Conjunto y Multiconjunto, el axioma 9 de cada TAD establece la diferencia entre el comportamiento de los objetos de cada tipo, en particular para los conjuntos, el axioma 9 de conjuntos indica que con la eliminaci¶on de un elemento, no quedan rastros de ¶el en el conjunto resultado, mientras que para el caso de multiconjuntos, el axioma 9 indica que un multiconjunto puede albergar m¶ultiples copias de un elemento.

3.3 Sobre los TADs y sus Representaciones en el Computador

La axiom¶atica de los tipos abstractos nos permite conocer sus propiedades y los operadores que permiten actuar sobre ellos. Sin embargo la tecnolog¶ia de la computaci¶on a¶un no permite que usemos este lenguaje para de¯nir estos objetos sobre los cuales queremos hacer algoritmos que los utilicen. En particular la presentaci¶on axiom¶atica que hemos hecho nos dice **QUE** son las propiedades que tienen que veri¯car nuestros tipos, pero no dice nada de **C¶OMO** lograr que se comporten de esa manera. Veamos con un ejemplo conocido la diferencia entre **QUE** y **COMO**. Dado un n¶umero entero X, podemos de¯nir que el operador DIVIDIDO2(X) de la siguiente manera: DIVIDIDO es un operador que toma un argumento entero y devuelve un resultado entero y si X = 2 ¤ Z o X = 2 ¤ Z + 1 entonces DIVIDIDO2(X) = Z. Esta de¯nici¶on del operador no nos dice como es que calcularemos el resultado. Una forma puede ser dividiendo por dos (usando el operador de divisi¶on de enteros) o puede hacerse con una operaci¶on de traslaci¶on a la derecha aprovechando la representaci¶on binaria de los enteros. Hay varias maneras de implementar **COMO** el **QUE**.

Implementar: Acci¶on de construir programas (generalmente en un lenguaje de programaci¶on) que permitan programar los operadores indicados en la especi¯caci¶on de tal manera que las acciones dadas en el c¶odigo rveri¯quen los axiomas dados en la parte sem¶antica de la especi¯caci¶on. En ciertas especificaciones los operadores no están totalmente definidos, por lo que la implementación deberá completarlos.

A continuación veremos varias formas de representar conjuntos en el computador. Cada una de las formas puede tener ventajas o desventajas respecto a la eficiencia de ciertos operadores, pero cualquiera de las implementaciones deberá verificar los axiomas de la especificación y por lo que cualquier sistema que utilice las propiedades de la especificación puede hacer uso de cualquier implementación o intercambiarlas. Se debe resaltar que las especificaciones que hemos visto hasta ahora se caracterizan por dar valores de verdad a los elementos que pueden ser **observados**. Los axiomas son independientes de la representación y en general no son constructivos. En ciertos casos parecen dar un procedimiento para realizar la operación, pero una implementación no está obligada a seguir los mismos pasos. Basta que los resultados de los operadores para iguales operandos sean iguales.

3.3.1 Implementación del TAD Conjunto

Representación Estática 1: Arreglo de Booleanos

Si conocemos la variedad de los elementos que pueden pertenecer al conjunto, lo que equivale a decir el conjunto Universal y este es **FINITO y razonablemente pequeño**, podemos, como puede hacerse con todo conjunto finito, hacer una correspondencia entre los elementos del conjunto Universal y los naturales de manera que cada posición de un arreglo corresponda a un elemento del conjunto Universal.

Si el conjunto Universal fuera {azul,rojo,verde}, representaríamos un conjunto por un arreglo de 3 elementos booleanos. El conjunto presentado a continuación (después de hacer corresponder a la primera posición el verde, a la segunda el azul y a la tercera el rojo) sería el conjunto que tiene el elemento verde pero no el azul ni el rojo.

verde	azul	rojo
true	false	false

En el caso en que el Universo es finito y pequeño, vimos que se puede implementar el conjunto con un arreglo de tamaño del Universo. El tiempo requerido, en esta implementación de las operadores de pertenece, insertar, eliminar son de tiempo constante mientras que el operador esvacio es de tiempo proporcional al tamaño del Universo. Puede hacerse una modificación de la estructura de datos, llevando adicionalmente la cuenta de la cardinalidad del conjunto, de manera tal que todas las operaciones se realicen a tiempo constante.

En el caso de Universo finito, esta representación puede interpretarsecomo representar cada conjunto con la función característica del conjunto:

$$F : A \to \text{Boolean}$$

$$F(x) = \begin{cases} \text{true} & \text{si } x \in A \\ \text{false} & \text{si no} \end{cases}$$

En el Apéndice A se presenta una implementación.

Representación Estática 2: Arreglo con política de no permitir duplicados

En el caso en que el Universo es finito pero muy grande (por ejemplo un subconjunto de enteros) y los conjuntos con los que se trabaja tienen una cantidad pequeña de elementos(relativa al tamaño del Universo) no parece conveniente dedicar tanta memoria para almacenar un conjunto, ya que la mayoría de las posiciones tendrá valor de falso. Un procedimiento es almacenar los valores de los elementos en el arreglo o referencias a ellos, ocupando las primeras posiciones del mismo y llevando la cuenta de las posiciones ocupadas del arreglo. La implementación propuesta hace que un conjunto sea un par (índice, arreglo), donde el arreglo puede ser de tipo elemento o de tipo referencia a elemento.

- La operación de inserción puede hacerse mediante el agregado del elemento en la primera posición libre del arreglo, revisando que no esté ya presente (tiempo proporcional al tamaño del arreglo).

- La operación de verificar si es vacío (esvacio) consistirá en revisar si la primera posición libre del arreglo coincide con la primera posición (tiempo constante).

- La implementación del operador pertenece puede ser la de recorrer el arreglo hasta encontrar el valor buscado (respuesta true) o hasta la última posición ocupada (respuesta false). El tiempo de este operador será proporcional al tamaño del conjunto.

- La operación de eliminar puede implementarse como recorrido del vector desde la primera posición. En caso de encontrarse el elemento se deberá desplazar los elementos hacia la izquierda o mover el último elemento del vector a la posición que se libera, corrigiendo el valor de la posición libre del arreglo. Esta operación tendrá un tiempo proporcional al tamaño del arreglo.
  En el Apéndice A se presenta una implementación.

Representación Estática 3: Arreglo con política de permitir duplicados

Esta implementación busca mejorar el tiempo de inserción respecto a la implementación anterior.

- La operación de inserción puede hacerse mediante el agregado del elemento en la primera posición libre del arreglo (tiempo constante).

- La operación de verificar si es vacío consistirá en revisar si la primera posición libre del arreglo coincide con la primera posición (tiempo constante).

- La implementación del operador pertenece puede ser la de recorrer el arreglo hasta encontrar el valor buscado (respuesta true) o hasta la última posición ocupada (respuesta false). El tiempo de este operador será proporcional al tamaño del conjunto.

- La operación de eliminar es la más costosa de esta representación. El elemento a ser eliminado puede estar repetido en el vector ya que para abaratar el operador de inserción no revisamos si el elemento ya pertenecía al conjunto, por lo que una implementación posible es el recorrido del vector desde la primera posición. En caso de encontrarse el elemento a ser eliminado se podrá usar uno de los dos mecanismos siguientes:

  - Se deberá disponer de dos cursores para completar el recorrido, uno para registrar la posición que se está revisando y otro para indicar a cual posición debe desplazarse el elemento revisado para lograr eliminar todas las apariciones del elemento que se desea eliminar.

– Se reemplazará el elemento a eliminar por el último del arreglo (reduciendo el valor del lugar libre), y se continuará el recorrido hasta agotar el arreglo.

Esta operación tendrá un tiempo proporcional al tamaño del arreglo.

Esta representación es muy conveniente para las aplicaciones que usan conjuntos que tienen una cantidad reducida de eliminaciones. Sin embargo la operación de pertenece parece tener un costo alto para conjuntos grandes. En el Capítulo 7 veremos soluciones mas eficientes para este operador.
En el Apéndice A se presenta una implementación.

3.3.2 Representación Dinámica

Los Conjunto se representan a través de estrucuturas simplemente encadenadas mediante referencias (o apuntadores). Cada elemento está representado como un nodo de la forma siguiente

De tal forma la estructura puede ser declarada como

```
Type
    Apunt-Nodo = ^Nodo
    Nodo    = record
      entrada: Tipo-Elem
      Sig-Nodo:Apunt-Nodo
    end
```

Un Conjunto vacío lo vamos a representar con un apuntador en **NIL**. En el apéndice A se presentan algunas implementaciones.

[2] La operación de inserción puede hacerse mediante el agregado del elemento en la primera posición dela lista (tiempo constante).

- La operación de verificar si es vacío consistirá en revisar si la lista es vacía(tiempo constante).

- La implementación del operador pertenece puede ser la de recorrer la lista hasta encontrar el valor buscado (respuesta true) o hasta el último elemento de la lista (respuesta false). El tiempo de este operador será proporcional al tamaño del conjunto.

- La operación de eliminar consistirá en una búsqueda del elemento a eliminar, recordando el que lo precede en la lista y al encontrarlo, cambiar el enlace del que lo precede a apuntar al que sucedía al elemento a eliminar.

3.4 Resumen del capítulo 3

En este capítulo hemos presentado dos tipos abstractos, el conjunto y el multiconjunto, con los operadores básicos usados en teoría de conjuntos. Podemos ver que las presentaciones se centran en establecer el qué y no el cómo. En general las presentaciones de los TAD no son constructivas. En las secciones siguientes se muestran algunas implementaciones posibles del tipo conjunto. No hemos desarrollado la parte correspondiente a las validaciones de las implementaciones por no recargar el material del capítulo. Sin embargo debe mencionarse que las implementaciones deben verificar todos y cada uno de los axiomas de la especificación. La implementación tendrá propiedades adicionales, o casos en que la implentación es válida. Por ejemplo, la implementación usando la función característica para representar un conjunto está restringida a la representaciones de subconjuntos de un universo finito y manejable. La segunda implementación parece más efectiva cuando los conjuntos a manipular son pequeños respecto al Universo. Sin embargo cuando se diseñan aplicaciones usando conjuntos se deberá hacer sin pensar en la implementación de los mismos. Una vez probada la aplicación puede pasarse a la fase de elegir la implementación que más se ajuste a las necesidades de la aplicación (tomando en cuenta la variedad y frecuencia de uso de los operadores en la aplicación).

3.5 Ejercicios

En los ejercicios se piden acciones que definimos a continuación:

- **Implementar** un TAD consiste en darle una representación concreta en un lenguaje de programación. La representación concreta consta de dos partes
  1. Representación de los objetos que constituyen el TAD mediante una estructura de datos soportada por el lenguaje.
  2. Representación de los operadores del TAD mediante programas que verifican los axiomas establecidos en la semántica del TAD. Las operaciones parciales debberán ser completadas.
- **Extender un TAD**: Se refiere a agregar operadores (sintaxis) y su descripción (semántica) que refleja el comportamiento esperado. Una extensión de

  $$ESP_T = (D, O, E)$$

  será

  $$ESP'_T = (D, O \cup O', E \cup E')$$

  donde O' corresponde a los nuevos operadores y E' corresponde a los axiomas que definen suficientemente a los operadores de O'.

1. Implemente el TAD Conjunto.
2. Extienda el TAD Conjunto con las operaciones del algebra de conjuntos (unión, intersección y diferencia).
3. Implemente el TAD Multiconjunto.
4. Considere la siguiente la siguiente extensión para $ESP_{Multiconjunto} = (D, O \cup \{\#ocurr\}, E \cup \{E_1, E_2, E_3\})$, con

   $E_1 : \#ocurr(vacio, e) = 0$
   $E_2 : \#ocurr(insertar(m, e), e) = 1 + \#ocurr(m, e)$
   $E_3 : \#ocurr(insertar(m, e), e_1) = \#ocurr(m, e_1)$

Proponga una representación para el TAD Multiconjunto que permita realizar eficientemente el operador #ocurr.

5. Considere $ESP_{Complejo}$ como sigue:

TAD Complejo

$$D = \{Real, Complejo\}$$

Sintaxis

comp :	$Real \times Real$	$\rightarrow$	Complejo
sumar :	$Complejo \times Complejo$	$\rightarrow$	Complejo
restar :	$Complejo \times Complejo$	$\rightarrow$	Complejo
mult :	$Complejo \times Complejo$	$\rightarrow$	Complejo
div :	$Complejo \times Complejo$	$\rightarrow$	Complejo
real :	Complejo	$\rightarrow$	Real
imag :	Complejo	$\rightarrow$	Real

Semántica
$\forall c_1, c_2 \in Complejo$;

1 $real(comp(r_1, r_2)) = r_1$

2 $imag(comp(r_1, r_2)) = r_2$

3 $sumar(c_1, c_2) = sumar(c_2, c_1)$

4 $real(sumar(c_1, c_2)) = real(c_1) + real(c_2)$

5 $imag(sumar(c_1, c_2)) = imag(c_1) + imag(c_2)$

Fin-Complejo;

(a) Indique cuáles son los conjuntos D, O y E de la terna

$$ESP_{Complejo} = (D, O, E).$$

(b) Usando sus conocimientos sobre números complejos, escriba los axiomas correspondiente a los operadores restar, mult y div del TAD Complejo.

(c) Implemente el TAD Complejo:

(c.1) Utilice representación cartesiana

(c.2) Utilice representación polar

(d) Extienda $ESP_{Complejo}$ con los operadores opuesto y conjugado, dando la sintaxis y la semántica.

6. Considere $ESP_{Racional}$ como sigue:

TAD Racional

$$D = \{Entero, Racional\}$$

Sintaxis

$rac:$	$Entero \times Entero$	$\rightarrow$	$Racional$
$sumar:$	$Racional \times Racional$	$\rightarrow$	$Racional$
$restar:$	$Racional \times Racional$	$\rightarrow$	$Racional$
$mult:$	$Racional \times Racional$	$\rightarrow$	$Racional$
$div:$	$Racional \times Racional$	$\rightarrow$	$Racional$
$numerador:$	$Racional$	$\rightarrow$	$Entero$
$denominador:$	$Racional$	$\rightarrow$	$Entero$

Semántica

$\forall r, r_1, r_2 \in Racional, e, e_1, e_2 \in Entero;$

1 $sumar(r_1, r_2) = sumar(r_2, r_1)$

2 $numerador(rac(e_1, e_2)) = e_1$

3 $denominador(rac(e_1, e_2)) = e_2$

4 $denominador(r) \neq 0$

Fin-Racional;

(a) Indique cuáles son los conjuntos D, O y E de la terna

$$ESP_{Racional} = (D, O, E).$$

(b) Escriba los axiomas correspondientes a los operadores restar, mult y div del TAD Racional.

(c) Implemente el TAD Racional.

7. Considere $ESP_{Polinomio}$ como sigue:

TAD Polinomio[Real]

$$D = \{Real; Polinomio; Boolean; Natural\}$$

Sintaxis

cero :		$\Rightarrow$	Polinomio
escero :	Polinomio	$\Rightarrow$	Boolean
defterm :	Polinomio $\times$ Natural $\times$ Real	$\Rightarrow$	Polinomio
multterm :	Polinomio $\times$ Natural $\times$ Real	$\Rightarrow$	Polinomio
coef :	Polinomio $\times$ Natural	$\Rightarrow$	Real
grado :	Polinomio	$\Rightarrow$	Natural

Semántica
$\forall p, p_1, p_2 \in Polinomio, n, n_1 \in Natural, r, r_1 \in Real;$

1 $escero(cero) = true$

2 $escero(p) = true \leftrightarrow coef(p, n) = 0$

3 $escero(p) = true \rightarrow escero(multterm(p, n, r)) = true$

4 $multterm(defterm(p, n, r), n_1, r_1) =$
$defterm(multterm(p, n_1, r_1), n + n_1, r * r_1)$

5 $coef(defterm(p, n, r), n) = r$

6 $escero(p) = false \rightarrow coef(defterm(p, n, r), n_1) = coef(p, n_1)$

7 $escero(p) = true \rightarrow grado(p) = 0$

8 $escero(p) = true \rightarrow grado(defterm(p, n, r)) = n$

9 $n > grado(p) \rightarrow grado(defterm(p, n, r)) = n$

10 $n < grado(p) \rightarrow grado(defterm(p, n, r)) = grado(p)$

Fin-Polinomio;

(a) Implementente el TAD Polinomio.

(b) Extienda $ESP_{Polinomio} = (D, O \cup \{sumar, multiplicar, derivar\}, E \cup E')$, donde E' es el conjunto de axiomas correspondiente a los operadores de suma,multiplicación y derivación de polinomios. Ud. debe dar el conjunto E'.

8. Considere la siguiente especificación para funciones de dominios finitos ($f : A \to B$):

TAD $Funcion[A, B]$

$$D = \{Funcion, A, B\}$$

Sintaxis

$new :$		$\Rightarrow$	$Funcion$
$def :$	$Funcion \times A \times B$	$\Rightarrow$	$Funcion$
$aplicar :$	$Funcion \times A$	$\Rightarrow$	$B \cup \{?\}$
$preimagen :$	$Funcion \times B$	$\Rightarrow$	$Conjunto[A]$
$definida? :$	$Funcion \times A$	$\Rightarrow$	$Boolean$
$inyectiva? :$	$Funcion$	$\Rightarrow$	$Boolean$

Semántica
$\forall a, a_1 \in A, b, b_1 \in B, f, f_1 \in Funcion,$

1 $f \neq new \wedge aplicar(f, a) = b \wedge aplicar(f, a) = b_1 \to b = b_1$

2 $aplicar(new, a) = ?$

3 $aplicar(def(f, a, b), a) = b$

4 $a \neq a_1 \to aplicar(def(f, a, b), a_1) = aplicar(f, a_1)$

5 $aplicar(f, a) = b \to pertenece(preimagen(f, b), a) = true$

6 $definida?(f, a) = (aplicar(f, a) \neq ?)$

7 $(aplicar(f, a) = aplicar(f, a_1) \to a = a_1) \leftrightarrow (inyectiva?(f) = true)$

Fin-Funcion;

(a) Proponga una representación concreta para el TAD $Funcion[A, B]$.

(b) Utilizando la representación propuesta, implemente los operadores del TAD.

(c) Complete la siguiente tabla con los órdenes de cada uno de los operadores, utilizando la representación propuesta:

new	def	aplicar	preimagen	definida?	inyectiva?

(d) Extienda el TAD $Funcion[A, B]$ con los siguientes operadores:

  (a) Composición de funciones.

  (b) Determinar si una función es sobreyectiva.

  (c) Restricción del dominio de la función, es decir, $f : A \to B$ y $C \subset A$ entonces definir $f|_C$.

  En cada caso, dé la sintaxis y la semántica de los operadores.

9. **Definición**: Una expresión en postorden sobre los naturales en se define como sigue:

   (a) $\forall a \in Z$, a es una expresión en postorden (expresión constante).

   (b) Si E_1 y E_2 son expresiones en postorden y $op \in \{+, -, \times, /\}$, entonces $E_1 E_2\, op$ es una expresión en postorden.

   Considere la siguiente especificación del TAD ExpPost:

   TAD $ExpPost[Natural]$

$$D = \{ExpPost, Operador, Natural, 1, 2\}$$

   Sintaxis

$$\begin{array}{lrcl}
ctte: & Natural & \Rightarrow & ExpPost \\
esctte: & ExpPost & \Rightarrow & Boolean \\
defexp: & ExpPost \times ExpPost \times Operador & \Rightarrow & ExpPost \\
subexp: & ExpPost \times \{1, 2\} & \Rightarrow & ExpPost \\
operador: & ExpPost & \Rightarrow & Operador
\end{array}$$

Semántica

$\forall e_1, e_2 \in ExpPost, n \in Natural, p \in \{1, 2\}, op \in Operador;$

1 $esctte(ctte(n)) = true$

2 $esctte(defexp(e_1, e_2, op)) = false$

3 $subexp(defexp(e_1, e_2, op), 1) = e_1$

4 $subexp(defexp(e_1, e_2, op), 2) = e_2$

5 $operador(defexp(e_1, e_2, op)) = op$

Fin-ExpPost;

(i) Implemente en el TAD ExpPost.

(ii) Defina un algoritmo para evaluar expresiones que satisfaga la siguiente especificación:

$evaluar : ExpPost \Rightarrow Natural$

(e1) $evaluar(ctte(n)) = n$

(e2) $evaluar(defexp(e_1, e_2, op)) = aplicar(op, evaluar(e_1), evaluar(e_2))$

3.6 Bibliografía

1. EHRIG, H. MAHR,B. & OREJAS, F. “Introduction to Algebraic Specifications. Part 1: Formal Methods for Software Development”. The Computer Journal. Vol. 35. #5. pp. 460-467. 1992.

2. EHRIG, H. MAHR,B. & OREJAS, F. “Introduction to Algebraic Specifications. Part 2: From Classical View to Foundations of System Specifications”. The Computer Journal. Vol. 35. #5. pp. 468-477. 1992.

3. GUTTAG, John. “ Abstract Data Types and the Development of Data Structures”. Comm. of ACM. Vol. 20, No. 6, June1977.

4. GUTTAG, John, HOROWITZ,Ellis & MUSSER,David. The Design of Data Specifications en “Current Trends in Programming Methodology. Vol IV”. Yeh Editor, 1978.

5. GUTTAG, John. Notes on Type Abstraction en \Software Speci¯cation Techniques". International Computer Science Series. Addison Wesley, 1986.

6. LIN, Huimin. \Procedural Implementations of Algebraic Speci¯cations". ACM TOPLAS, Vol. 15, No. 5, Nov. 1993. pp.876-895.

Cap¶tulo 4

TAD Secuencia. Especializaciones

4.1 Marco Conceptual

En esta secci¶n nos ocuparemos del tipo de datos con estructura. Estos objetos, conocidos como secuencias poseen la propiedad de ser elementos que est¶n organizados linealmente. Ejemplos de secuencias son:

(i) $< \mathrm{blanco}; \mathrm{azul}; \mathrm{rojo}; \mathrm{verde} >$

(ii) $< 1; 2; 4; 8; 12; 14 >$

(iii) $< 7; 6; 5; 4; 3; 2; 1 >$

(iv) $< 7; 5; 5; 4; 3; 3; 1 >$

Como vemos en los ejemplos, una secuencia puede tener elementos iguales en las diferentes posiciones, pero todos los elementos son del mismo tipo (homog¶nea), por lo que podemos considerar una secuencia como una posible presentaci¶n de un multiconjunto.

En la secci¶n siguiente se dar¶ el TAD Secuencia y se analizar¶n algunas de sus posibles representaciones. Cada una de las representaciones tendr¶ comportamientos diferentes para las operaciones del TAD.

4.2 Especificación del TAD Secuencia

En esta sección, definimos el TAD Secuencia que nos permite manejar objetos estructurados de la forma:

$$s = \langle e_1, e_2, \ldots, e_n \rangle$$

Intuitivamente podemos caracterizar estos objetos como sigue:

1. Todos los elementos en la secuencia son del mismo tipo (secuencias homogéneas).

2. $\forall i, e_i$ ocupa la i-ésima posición en la secuencia, por lo que hay un puesto asociado a cada elemento que ocurra en una secuencia.

3. Una secuencia puede crecer y decrecer dinámicamente (no existe un tamaño fijo).

A continuación presentamos $ESP_{Secuencia}$:

TAD Secuencia[Elemento]

$$D = \{Secuencia, Elemento, Natural\}$$

Sintaxis

vacia :		$\Rightarrow$	Secuencia
esvacia :	Secuencia	$\Rightarrow$	Boolean
insertar :	Secuencia $\times$ Natural $\times$ Elemento	$\Rightarrow$	Secuencia
eliminar :	Secuencia $\times$ Natural	$\Rightarrow$	Secuencia
proyectar :	Secuencia $\times$ Natural	$\Rightarrow$	Elemento
esta :	Secuencia $\times$ Elemento	$\Rightarrow$	Boolean
long :	Secuencia	$\Rightarrow$	Natural

Semántica

$\forall s \in Secuencia, p, p_1 \in Natural, e, e_1 \in Elemento,$

1. $esvacia(vacia) = true$

2. $0 < p \leq long(s) + 1 \rightarrow esvacia(insertar(s, p, e)) = falso$

3. $0 < p \leq long(s) + 1 \rightarrow$
   $proyectar(insertar(s, p, e), p) = e$

4. $0 < p_1 < p \cdot long(s) + 1 \; !$
   $proyectar(insertar(s; p; e); p_1) =$
   $\qquad\qquad proyectar(s; p_1)$

5. $0 < p < p_1 \cdot long(s) + 1 \; !$
   $proyectar(insertar(s; p; e); p_1) = proyectar(s; p_1 \text{ ¡ } 1)$

6. $0 < p_1 < p \cdot long(s) \; !$
   $proyectar(eliminar(s; p); p_1) = proyectar(s; p_1)$

7. $0 < p < p_1 \cdot long(s) \text{ ¡ } 1 \; !$
   $proyectar(eliminar(s; p); p_1) = proyectar(s; p_1 + 1)$

8. $0 < p \cdot long(s) + 1 \; !$
   $proyectar(eliminar(insertar(s; p; e); p); p_1) = proyectar(s; p_1)$

9. $esta(vacia; e) = falso$

10. $0 < p \cdot long(s) + 1 \; ! \; esta(insertar(s; p; e); e) = true$

11. $e \,6\!\!= e_1 \wedge 0 < p < long(s) + 1 \; !$
    $esta(insertar(s; p; e); e_1) = esta(s; e_1)$

12. $0 < p \cdot long(s) + 1 \; !$
    $esta(eliminar(insertar(s; p; e); p); e_1) = esta(s; e_1)$

13. $long(vacia) = 0$

14. $0 < p \cdot long(s) + 1 \; ! \; long(insertar(s; p; e)) = 1 + long(s)$

15. $0 < p \cdot long(s) + 1 \; !$
    $long(eliminar(insertar(s; p; e); p)) = long(s)$

Fin-Secuencia;

N¶tese que cuando se da la sem¶antica de los operadores no se dice nada en relaci¶on a las inserciones donde la posici¶on $p > long(s) + 1$, donde $long(s)$ es la longitud de la secuencia s.

Una de las diferencias entre conjuntos y las secuencias, es que en ¶estas hay una posici¶on asociada a cada elemento lo que llamamos en Ap¶endice B posici¶on espacial.

Hemos presentado operaciones asociadas a un TAD, de una manera algo abstracta. El lenguaje natural es naturalmente ambiguo y si se utiliza para de¯nir operadores puede ser una fuente de error al momento de desarrollar aplicaciones usando los operadores, veamos un ejemplo:

Se quiere agregar una operaci¶on para borrar al tipo Secuencia. Se propone la siguiente: Sintaxis

$$\mathrm{borrar}: \ \mathrm{Secuencia} \ £ \ \mathrm{Secuencia} \ ) \ \mathrm{Secuencia}$$

(a) borrar$(s; s_1)$ borra todas las ocurrencias de s_1 en s, siendo $s; s_1$ secuencias.

Analizando la de¯nici¶on dada para el operador borrar, vemos que la de¯nici¶on es ambigua debido a la palabra ocurrencia, ya que pudiera ser interpretada como: Sem¶antica

- Las ocurrencias de s_1 existentes en s

  Ej: borrar$(< b; a; b; a; l; a; l; a >; < b; a; l; a >) = < b; a; l; a >$

- Todas las ocurrencias de s_1 ya sean porque originalmente estaban en s y aquellas que se generen al hacer eliminaciones de la secuencia s_1.

  Ej: borrar$(< b; a; b; a; l; a; l; a >; < b; a; l; a >) = \mathrm{vacia}$

Para trabajar este ejemplo generalizaremos el operador esta de secuencias

$$\mathrm{esta}: \ \mathrm{Secuencia} \ £ \ \mathrm{Secuencia} \ ) \ \mathrm{Secuencia}$$

con la sem¶antica:

$\mathrm{esta}(s_2 \ k \ s_1 \ k \ s_3; s_1) = \mathrm{verdadero}$ siendo k el operador de concatenaci¶on de secuencias.

Con esta de¯nici¶on adicional podemos establecer la sem¶antica para los casos:

- $\mathrm{esvacia}(s_1) \ ! \ s_2 \ k \ s_1 = s_2$
- $\mathrm{esvacia}(s_1) = \mathrm{false} \ ! \ s_2 \ k \ s_1 = \mathrm{insertar}(s_2; \mathrm{proyectar}(s_1; 1) \ k \ \mathrm{eliminar}(s_1; 1)$
- $\mathrm{esta}(s; s_1) = \mathrm{falso} \ ! \ \mathrm{borrar}(s; s_1) = s$
- $\mathrm{borrar}(s_2 \ k \ s_1 \ k \ s_3; s_1) = \mathrm{borrar}(\mathrm{borrar}(s_2; s_1) \ k \ \mathrm{borrar}(s_3; s_1); s_1)$

- $\mathrm{esta}(\mathrm{borrar}(s; s_1); s_1) = \mathrm{falso}$

Antes de pasar a estudiar algunas representaciones del TAD Secuencia, consideremos el siguiente problema:

Problema: Escriba una funci¶n que dada una secuencia de letras genere una secuencia de 26 enteros, donde la i-¶sima posici¶n de la misma corresponde al n¶mero de ocurrencias de la letra que ocupa esa posici¶n en el alfabeto.

A continuaci¶n presentamos dos soluciones, la primera de ellas (Soluci¶n 1) resuelve el problema eligiendo un car¶cter y eliminando de la secuencia todas las apariciones de ese car¶cter y contando la cantidad de caracteres eliminados.

Soluci¶n 1:

```
function ContarLetras(s: Secuencia[Char]): Secuencia[Natural];
var s1: Secuencia[Natural];
    c: Char; cont,i: Natural;
begin
    s1 := SecCero(26);
    while not(esvacia(s)) do
    begin
        c := proyectar(s,1);
        cont := 1;
        s := eliminar(s,1);
        while esta(s,c) do
        begin
            cont := cont + 1;
            i := 1;
            while (c<>proyectar(s,i)) do
                  i := i + 1;
            s := eliminar(s,i)
        end;
        s1 := insertar(eliminar(s1,PosABC(c)),PosABC(c),cont)
    end;
    return(s1)
end;
```

La Solución 2 consiste en recorrer una sola vez la secuencia y dependiendo del carácter que se observa, se suma 1 al contador correspondiente a ese carácter.

Solución 2:

```
function ContarLetras(s: Secuencia[Char]): Secuencia{Entero];
var s1: Secuencia[Entero];
    i,len,p,c: Entero;

begin
   s1 := SecCero(26);
   len := long(s);
   for i:= 1 to len do
       p := PosABC(proyectar(s,i));
       c := proyectar(s1,p) + 1;
       s1 := insertar(eliminar(s1,p),p,c)
   od;
   return(s1)
end;
```

La función PosABC recibe un carácter y devuelve su posición en el alfabeto. La función SecCero(i) devuelve una secuencia de i ceros.

En la siguiente sección se presentan algunas implementaciones del TAD Secuencia y para cada una de ellas se analizan las dos soluciones propuestas arriba.

4.3 Implementación del TAD Secuencia

A continuación se presentan dos formas de representar el TAD Secuencia. Para la implementación se requiere definir cómo será representado en la memoria del computador el conjunto soporte del tipo. En este caso particular, debemos saber cómo se representa la secuencia vacía y el resto de las secuencias. Seguidamente se deben programar las operaciones del tipo abstracto haciendo uso de la estructura de datos establecida para la representación de las secuencias.

La implementación de las operaciones deben satisfacer los axiomas dados en la semántica de la especificación. Las representaciones que se dan a continuación corresponden a posibles implementaciones en lenguajes de programación donde el manejo de memoria se deja al programador [1].

4.3.1 Representación Estática

Cuando una secuencia se representa mediante un arreglo, se dice que la representación es estática. Este tipo de representación se caracteriza porque los elementos de la secuencia se encuentran en posiciones de almacenamiento de memoria físicamente contiguas. En PASCAL podemos representar estáticamente el tipo secuencia como sigue:

```
Type Secuencia = record
                    Casillas: array[1..N] of Elemento;
                    Ult:0..N
                 end;
```

Donde `N` es una constante predefinida, `Casillas` es el arreglo donde se almacenan los elementos y `Ult` indica la posición del último elemento de la secuencia dentro del arreglo. La secuencia estará almacenada en las primeras posiciones del arreglo ya que la longitud de la secuencia coincide con la posición del arreglo donde se almacena el último elemento de la secuencia. Hay que hacer notar que si la definición del arreglo dentro del registro tuviese otro rango lo anterior no sería válido.

En el apéndice A se presenta una implementación del TAD Secuencia en PASCAL utilizando representación estática.

Ventajas

- Acceso directo a cada elemento de la secuencia, por lo que el costo es O(1).
- La implementación del operador long es O(1). En esta implementación el operador long será `long(s) = s.Ult`.

---

[1] El tipo que aqui definimos puede ser realizado en JAVA utilizando la clase java.util.Vector

Desventajas

- Desperdicio de memoria. La definición asigna a cada secuencia un arreglo de tamaño predefinido y este sólo se aprovecha hasta la posición `Ult`. La representación de la secuencia vacía será aquella donde `Ult` tiene el valor cero.

- Rigidez para el tamaño de las secuencias. El tamaño máximo permitido sera igual al tamaño del arreglo.

- Dificultades para hacer inserciones y eliminaciones. La estructura debe reorganizarse constantemente por lo que las operaciones de inserción y eliminación tienen una complejidad lineal respecto al tamaño de la secuencia.

4.3.2 Representación Dinámica

Cuando una secuencia se representa dinámicamente, cada elemento de la lista es un registro que consta de dos campos, el primero almacenará el elemento, y el segundo la dirección del próximo elemento de la secuencia. Este último campo se denomina referencia o apuntador. Una característica fundamental de este tipo de representación es que los elementos no ocupan, necesariamente, posiciones contiguas en memoria. Además, los elementos (instancias del registro) se van creando a medida que se van insertando elementos a la secuencia.

Típicamente en PASCAL, el tipo Secuencia se representa dinámicamente como sigue:

```
Type Secuencia = ^Casilla;
     Casilla = record
                     Elem:Elemento;
                     Sig:Secuencia
               end;
```

En este caso, un objeto de tipo Secuencia, es un apuntador a una sucesión de `Casillas`. Este apuntador se denomina cabeza de la lista y es el que se utiliza para la representación de la secuencia vacía.

Este encadenamiento de elementos es lo que comúnmente se denomina **listas lineales**; de tal forma que podemos decir que representar dinámicamente una secuencia es equivalente a representarla utilizando listas con una cabeza. Para crear un nuevo elemento de una lista en PASCAL, se utiliza el procedimiento predefinido **new(p)**, que busca en memoria un espacio acorde al tipo del elemento apuntado por **p** y devuelve en **p** la dirección de memoria obtenida. Claro está, **p** debe ser de tipo apuntador a los objetos que constituyen la secuencia (en el ejemplo de tipo `Casilla`).

Además, PASCAL tiene una constante predefinida llamada **nil**, que representa la dirección nula. Si se hace la siguiente declaración:

```
Var s: Secuencia;
```

y inicializa s como una secuencia vacía de la siguiente manera:

```
s := vacia();
```

luego de ejecutarse esa instrucción, s tendrá como valor `nil`.

En el apéndice A se presenta una implementación del TAD Secuencia en PASCAL utilizando representación dinámica.

Ventajas

- Utilización eficiente de la memoria ya que la memoria utilizada es proporcional al tamaño de la secuencia.
- Elimina la necesidad de limitar el tamaño de las secuencias ya que el límite será la memoria disponible en el computador.

Desventajas

- Mayor costo en el acceso a los elementos ya que hay que recorrer los e_i, $\forall i \leq p$, para acceder a e_p por lo que la proyección es un operador de complejidad la longitud de la secuencia.
- se aumenta la memoria requerida para almacenar un elemento debido a la adición del campo `Sig` en el registro `Casilla`.

  Las operaciones de inserción y eliminación (al igual que en la representación estática) son de complejidad proporcional a la longitud de la secuencia.

4.4 Especializaci¶n del TAD Secuencia

Como se ha visto hasta ahora, el TAD Secuencia es un tipo muy general y puede considerarse el punto de partida para la de¯nici¶n de otros tipos cuyos objetos son secuencias con esquemas espec¶¯cos de crecimiento y decrecimiento. Tal es el caso de los tipos Cola y Pila que estudiaremos en las pr¶ximas secciones. Estos tipos son considerados especializaciones del TAD Secuencia, ya que restringen el conjunto de operaciones que lo caracterizan, para adquirir un comportamiento propio [2].

Otro ejemplo de especializaci¶n del TAD Secuencia es el TAD Dipolo. Un dipolo es una secuencia en la cual s¶lo se permite insertar y eliminar elementos tanto al principio como al ¯nal de la secuencia.

4.4.1 Especi¯caci¶n del TAD Pila

El TAD Pila es una secuencia, donde la forma de insertar y eliminar elementos de la misma, sigue una pol¶tica LIFO (Last In First Out), lo que signi¯ca que cuando se elimina un elemento, este es el ¶ltimo que fue insertado. Esto quiere decir que toda inserci¶n de elementos se har¶ por el extremo ¯nal o tope y que la supresi¶n de un elemento corresponde al ubicado en su extremo ¯nal (¶ltimo elemento insertado). Esto se re°eja en los axiomas donde usando la noci¶n de de¯nir un nuevo tipo abstracto como especializaci¶n de otro tipo abstracto (TAD $Pila$ < Secuencia[Elemento] , $Pila$ como especializaci¶n (<) de secuencia). Es por ser una especializaci¶n que los operadores del nuevo tipo ser¶n de¯nidos usando algunos de los operadores del TAD base.

TAD $Pila$ < Secuencia[$Elemento$]

$$D = fPila; Elementog$$

Sint¶xis

$vacia_P$:		)	$Pila$
$esvacia_P$:	$Pila$	)	Boolean
empilar :	$Pila$ £ $Elemento$	)	$Pila$
desempilar :	$Pila$	)	$Pila$
tope :	$Pila$	)	$Elemento$

[2] En JAVA se llama extensiones (java.util.Stack es una extensi¶n de java.util.Vector) pero las especializaciones consideradas en este libro se re¯ere a un subconjunto del dominio que puede ser creado mediante el uso de los operadores pero con ciertas restricciones

Semántica

$\forall p \in Pila; e \in Elemento;$

1. $esvacia_P(p) = esvacia(p)$
2. $empilar(p; e) = insertar(p; 1; e)$
3. $\neg esvacia_P(p) \rightarrow desempilar(p) = eliminar(p; 1)$
4. $\neg esvacia_P(p) \rightarrow tope(p) = proyectar(p; 1)$

Fin-Pila;

La semántica dada considera que la pila se simula haciendo los ingresos y egresos de elementos en la primera posición de la secuencia. Otra semántica alternativa sería la de considerar que las operaciones de empilar y desempilar se realizan por el fin de la secuencia:

Semántica

$\forall p \in Pila; e \in Elemento;$

1. $esvacia_P(p) = esvacia(p)$
2. $empilar(p; e) = insertar(p; long(e) + 1; e)$
3. $\neg esvacia_P(p) \rightarrow desempilar(p) = eliminar(p; long(e))$
4. $\neg esvacia_P(p) \rightarrow tope(p) = proyectar(p; long(e))$

Implementación del TAD Pila

En esta sección presentamos el TAD Pila como una especialización del TAD Secuencia por lo que se puede utilizar cualquiera de las representaciones que se utilizan para Secuencia. Si consideramos la representación estática dada en la sección 4.3.1, el campo `Ult` del registro correspodería al tope de la pila y sería conveniente empilar en la posición `Ult+1` para garantizar O(1) en esta operación. En cuanto a la representación dinámica dada en la sección 4.3.2, bastaría con empilar en la cabeza de la lista para garantizar O(1). En el apéndice A se encuentran las implementaciones mencionadas arriba para el TAD Pila.

4.4.2 Ejemplo de uso

Invertir una secuencia utilizando una pila

precond: sec = $< x_1; x_2; :::x_n >$

postcond: Invertir-Sec = $< x_n; x_{n ¡ 1}; :::x_1 >$

```
function Invertir-Sec ( sec :secuencia) : secuencia
var p: PILA
 i: Integer
  p = vaciap();
  while not( esvacia(sec)) do
   p := empila(p,proyectar(sec,1))
   sec := eliminar(sec,1)
  od
  i := 1
  while not (esvaciap(p)) do
   sec := insertar(sec,i,tope(p));
   i := i+1;
   p := desempilar(p)
  od
return(sec)
```

4.4.3 Especi¯caci¶n del TAD Cola

El TAD Cola por su parte, es al igual que el TAD Pila, una especializaci¶n de una secuencia, cuya pol¶tica de manejo es FIFO (First In First Out) , es decir, el primer elemento que entra es el primero en salir (se elimina el m¶s antiguo en la cola). Esto se re°eja en los axiomas.

TAD Cola < Secuencia[Elemento]

$$D = fCola; Elementog$$

Sint¶xis

$vacia_C$:		$\Rightarrow$	Cola
$esvacia_C$:	Cola	$\Rightarrow$	Boolean
encolar :	Cola $\times$ Elemento	$\Rightarrow$	Cola
desencolar :	Cola	$\Rightarrow$	Cola
frente :	Cola	$\Rightarrow$	Elemento

Semántica

$\forall c \in Cola; e \in Elemento;$

1. $esvacia_C(c) = esvacia(c)$

2. $encolar(c; e) = insertar(c; long(c) + 1; e)$

3. $\neg esvacia_C(c) \rightarrow desencolar(c) = eliminar(c; 1)$

4. $\neg esvacia_C(c) \rightarrow frente(c) = proyectar(c; 1)$

Fin-Cola;

4.4.4 Ejemplo de uso: Invertir una cola

Implementación recursiva

function Invertir-Cola(col:Cola): Cola

precond: col $= < x_1; x_2; \ldots x_n >$
postcond: Invertir-Cola $= < x_n; x_{n-1}; \ldots x_1 >$

```
var X : Elemento
  if esvaciac(col)
     Invertir-Cola := vaciac()
  else
     X := frente(col);
     Invertir-Cola := encolar(Invertir-Cola(desencolar(col)),X)
  _
```

Implementación iterativa

```
function Invertir-Cola(col:Cola): Cola
```

precond: col = $< x_1, x_2, \ldots x_n >$
postcond: Invertir-Cola = $< x_n, x_{n-1}, \ldots x_1 >$

```
var X : Elemento
sec : secuencia;
col-aux : Cola;
col-aux = Copiar-cola(col);
sec := vacia();
  while not( esvaciac(col-aux)) do
        X := frente(col-aux);
        sec := insertar(sec,1,X);
        col-aux := desemcolar(col-aux);
  od
  while (long(sec) ≠ 0) do
        X := proyectar(sec,1);
        col-aux := encolar(col-aux,X);
        sec := eliminar(sec,1);
  od
Invertir-Cola := col-aux
end
```

4.4.5 Ejemplo

Supongamos que un funcionario en una taquilla recibe planillas que entrega al público; los usuarios las llenan y las devuelven. Las planillas entregadas por los usuarios serán revisadas, firmadas y devueltas al público. Esta operación la realiza el empleado en las horas en que la taquilla no está abierta al público. Para facilitar la búsqueda en el momento de entregar las planillas el funcionario ha decidido tenerlas apiladas de manera tal que todas las planillas de solicitantes cuyo apellido empiece por una misma letra, estén contiguas, pero que se conserve el orden por día y hora de llegada. Consideremos primero una solución manual a este problema:

Inicialmente el empleado puede “apilar” las planillas con lo cual conserva el orden de llegada de las planillas. Conclu¶do el horario de recepci¶n, puede distribuir la pila en 26 pilas de acuerdo a la inicial del primer apellido. En estas pilas las planillas estar¶n en orden inverso al orden de llegada. Si a continuaci¶n recorre las pilas (en orden alfab¶tico) y las coloca en un archivador (cola), esta cola tendr¶ dentro de cada letra el orden de llegada a la taquilla.

A continuaci¶n presentamos el programa asociado al proceso descrito arriba.

```
procedure Taquilla(HorasRecepcion: Integer);

{ Los procedimientos ``espera_evento'', ``avanza_reloj''  y
 ``recibe_planilla'', se encargan de detectar  la ocurrencia
 de los eventos de reloj o de entrega de planilla, actualizar
 la hora, y registrar los datos en un formulario, respectivamente.}

const reloj: 0;
      entrega: 1;
type Formulario = record
                        Apellido: String;
                        Hora: 0..23;
                        Minutos: 0..59
                   end;
     Letras: `A'..`Z';
var P: Pila[Formulario];
    Piletas: array[Letras] of Pila[Formulario];
    Archivador: Cola[Formulario];
    F: Formulario; HorasTranscurridas : Integer;
    Evento: reloj..entrega; l: Letras;

begin

    { Inicializaciones}

    HorasTranscurridas := 0; P := vacia;
    Archivador := vacia;
    for l:= `A' to `Z' do
```

```
        Piletas[l] := vacia
    od;

    { Recepcion de planillas}

    while (HorasTranscurridas <= HorasRecepcion) do
        evento := espera_evento;
        case evento of
        reloj: avanza_reloj(HorasTranscurridas);
        entrega: begin
                     F := recibe_planilla;
                     empilar(P,F)
                 end
        endcase;
    od;

    { Organizacion de las planillas en orden alfabetico y de llegada,
      ya que ha transcurrido el horario de atencion al publico      }

    while not(esvacia(P)) do
        F := tope(P);
        desempilar(P);
        empilar(Piletas[Inicial(F.Apellido)],F);
    od;
    for l:= `A' to `Z' do
        while not(esvacia(Piletas[l]) do
           F := tope(Piletas[l]);
           desempilar(Piletas[l]);
           encolar(Archivador,F)
        od
    od;
end;
    { Los formularios quedan en el archivador en el orden deseado }
```

Es de hacer notar que a este código está expresado usando solamente operadores del TAD y este es el momento en que deban analizar diversas implementaciones de los TAD's Pila y Cola a fin de identificar cuales son más apropiadas teniendo en cuenta la frecuencia del uso de los operadores en la aplicaci¶n, sin que el c¶digo deba ser alterado.

4.4.6 Especi¯caci¶n del TAD Dipolo

El TAD Dipolo es una secuencia, cuya pol¶tica de manejo es que tanto los ingresos como los egresos a la secuencia se realizan en los extremos de la secuencia. Esto se re°eja en los axiomas.

TAD Dipolo < Secuencia[Elemento]

$$D = fDipolo; Elemento; Boolleang$$

Sint¶xis

$vacia_D$:		)	Dipolo
$esvacia_D$:	Dipolo	)	Boolean
ingresarFin :	Dipolo £ Elemento	)	Dipolo
egresarInicio :	Dipolo	)	Dipolo
ingresarFin :	Dipolo £ Elemento	)	Dipolo
egresarInicio :	Dipolo	)	Dipolo
frente :	Dipolo	)	Elemento
fin :	Dipolo	)	Elemento

Sem¶ntica

8d 2 Dipolo; e 2 Elemento;

1. $esvacia_D(d) = esvacia(d)$
2. $ingresarFin(d; e) = insertar(d; long(d) + 1; e)$
3. $ingresarInicio(d; e) = insertar(d; 1; e)$
4. $: esvacia_D(d)\ !\ egresarFin(d) = eliminar(d; long(d))$
5. $: esvacia_D(d)\ !\ egresarInicio(d) = eliminar(d; 1)$
6. $: esvacia_D(d)\ !\ ingresarFin(d) = insertar(d; long(c) + 1; e)$
7. $: esvacia_D(d)\ !\ ingresarInicio(d) = insertar(d; 1; e))$
8. $: esvacia_D(d)\ !\ frente(d) = proyectar(d; 1)$

9. $: esvacia_D(d) \;\; ! \;\; fin(d) = proyectar(d; long(d))eliminar$

Fin-Dipolo;

En todo los casos de especializaciones del TAD secuencia, utilizamos los operadores de secuencias, con valores pre¯jados para describir la sem¶antica de las operaciones del nuevo tipo.

4.5 Resumen del cap¶itulo 4

El TAD secuencia es el tipo abstracto mas utilizado para organizar una colecci¶on de elementos que requieran estar linealmente organizados (organizaci¶on espacial). Es por eso que podemos considerar a la secuencia como un enriquecimiento de los multiconjuntos, ya que el operador pertenece llamado esta puede combinarse con el operador proyectar para obtener el elemento y eliminar para quitarlo de la secuencia.

Debe recordarse siempre el ingrediente requieran estar linealmente organizados ya que es muy com¶un el uso de este tipo para representar simplemente conjuntos. No se debe utilizar tipos que posean propiedades por encima de las requeridas ya que esto restringe las implementaciones que se puedan usar.

Entre las colecciones que requieran estar linealmente organizadas, podemos distinguirlas seg¶un la pol¶itica de ingreso y egreso de elementos, siendo la secuencia la mas libre de las pol¶iticas, la de cola la que re°eja la ordenaci¶on de los elementos por el orden de aparici¶on de los mismos en el sistema y la pila que ha demostrado su utilidad en procesos de evaluaci¶on de procedimientos recursivos, en la evaluaci¶on de expresiones algebraicas. En este cap¶itulo mostramos que PILA y COLA son especializaciones del tipo secuencia (sub-¶algebra), ya que pueden ser expresadas sus operaciones en funci¶on de los operadores de secuencia instanciando en forma constante alguno de sus operandos.

4.6 Ejercicios

1. Calcule la complejidad de la soluci¶on al problema del ejemplo.

2. Considere la siguiente funci¶on:

   ```
   function F(s: Secuencia[Entero]): Secuencia[Entero];
   ```

```
var e,e1,i,j,len,x: Entero;
begin
   len := long(s);
   for i := len downto 1 do
       x := 0;
       e := proyectar(s,i);
       for j := 1 to i-1 do
           e1 := proyectar(s,j);
           if (e1 <= e) then
              x := x + e1
           fi
       od;
       s := insertar(s,i+1,x)
   od;
   return(s)
end;
```

(a) Dé la secuencia resultante para la siguiente entrada:

$$s = < 67; 10; 4; 18; 15 >$$

(b) Calcule el $T(N)$ de la función F si el TAD Secuencia[Entero] está implementado mediante:

(b.1) Representación Estática.

(b.2) Representación Dinámica.

(c) Justifique con razones de eficiencia el uso de las variables locales e, e1 y len en la función F.

3. Considere la siguiente función:

```
function MisterioSecuencial(s: Secuencia[Natural]):Secuencia[Natural];
var
  s1: Secuencia[Natural];
  a,i,j,k,len: Natural;
begin
    i := 1;  len := long(s);
    vacia(s1); k := 0;
    repeat
```

```
            j := i+1; a := 0;
            while (proyectar(s,i)<>proyectar(s,j)) and (j<len) do
            begin
                a := a + proyectar(s,j); j := j+1
            end;
            k := k+1;
            if (proyectar(s,i) = proyectar(s,j)) then
               insertar(s1,k,a)
            else insertar(s1,k,0);
            i := i+1;
       until (i=len);
       return(s1)
   end;
```

(a) Dé la secuencia resultante para la siguiente entrada:

$$s = < 4; 1; 3; 7; 4; 3; 1; 3 >$$

(b) Dé la caracterización de la secuencia de entrada para el peor caso. Calcule el orden del $T(N)$ para la caracterización de s dada arriba en los siguientes casos:

(b.1) La secuencia está representada estáticamente.

(b.2) La secuencia está representada dinámicamente.

4. El TAD String es un caso particular del TAD Secuencia:

$$\text{TAD String} = \text{Secuencia}[\text{Caracter}]$$

(a) Extienda ESP_{String} con los operadores:

$$\begin{array}{rrcl} \#ocurr: & String \times String & \to & Natural \\ concatenar: & String \times String & \to & String \\ palindrome: & String & \to & Boolean \end{array}$$

donde

(a.1) $\#ocurr(s; s_1)$ determina el número de ocurrencias del string s_1 en el string s.

(a.2) $concatenar(s; s_1)$ concatena los strings argumentos.

(a.3) palindrome(s) es true si s es palíndromo, es decir, si se lee igual de izquierda a derecha y de derecha a izquierda. Ejm: arepera.

(b) Implemente los operadores definidos en (a).

5. Especifique el TAD Dipolo en términos de los axiomas del TAD Secuencia. Sugiera una representación dinámica. Justifique.

6. Extienda la especificación dada del TAD Secuencia como sigue:

$$ESP_{\text{Secuencia}} = (D, O \cup \{agrupar, invertir\}, E \cup E')$$

donde

(a) agrupar(s) reorganiza los elementos de s, de forma tal que todos los duplicados ocurran en posiciones contiguas de s,

(b) invertir(s) invierte los elementos de s.

(c) E' es el conjuntos de axiomas correspondiente a agrupar e invertir

7. Usando el axioma 5 de la especificación de Secuencia y el axioma 4 de la especificación de Pila, obtener:

$$tope(desempilar(empilar(p, e)) = tope(p)$$

8. Usando el axioma 2 de la especificación de Secuencia y el axioma 2 de la especificación de Pila, obtener:

$$esvacia(empilar(p, e)) = falso$$

9. En la editorial “La Estrella” tienen un problema. El encargado del análisis de los textos literarios, aunque es muy bueno analizando, no es muy cuidadoso al momento de hacer citas textuales y generalmente olvida o abrir las comillas o cerrar las comillas para una cita. Por esta razón, se desea que Ud. implemente un algoritmo que, utilizando el TAD Pila, verifique si las comillas están balanceadas en un texto dado. Recuerde que existen comilla-derecha y comilla-izquierda. Recuerde que una cita puede contener citas.

(a) Realizar el ejercicio sin utilizar aritmética (enteros, reales).

(b) Realizar el ejercicio utilizando enteros.

10. Dé un ejemplo donde sea preferible utilizar el TAD Cola al TAD Pila.

11. Dé un ejemplo donde sea preferible utilizar el TAD Pila al TAD Cola.

12. Para organizar un conjunto de objetos que van a ser despachados se puede utilizar una organización de secuencia (o alguna especialización). Describa un proceso según el cual, organizando los objetos en una cola, existirán objetos que nunca serían despachados.

13. Se desea que implemente en PASCAL el TAD Cola[Estudiante] que desean solicitar tickets del Comedor. Notar que los datos importantes del estudiante, en este caso, son el carnet y el número de tickets que solicita.

14. Sea $ESP_{Cola} = (D_{Cola}, O_{Cola}, E_{Cola})$. Especifique el TAD ColaconColeados como

    $ESP_{ColaconColeados} = (D_{Cola}, O_{Cola} \cup \{colear\}, E_{Cola} \cup E')$,

    donde E' es el conjunto de axiomas correspondientes al operador colear.

    $colear(c, i, e)$ "colea" al elemento e delante de la i-ésima posición en c. Complete la especificación.

15. Sea $ESP_{Pila} = (D_{Pila}, O_{Pila}, E_{Pila})$. Especifique el TAD PilaconIntercalados como

    $ESP_{PilaconIntercalados} = (D_{Pila}, O_{Pila} \cup \{intercalar\}, E_{Pila} \cup E')$,

    donde E' es el conjunto de axiomas correspondientes al operador intercalar.

    $intercalar(p, i, e)$ "intercala" al elemento e sobre de la iésima posición p.

16. Se quiere agregar una operación para editar al tipo String. Se propone la siguientes:

    (b) $reemplazar(s, s_1, s_2)$ reemplaza en el string s, las ocurrencias del string s_1 por el string s_2.

Sin embargo,la descripcion es ambigÄa. Proponga de¯niciones no ambigÄas compatibles con la de¯nici¶n.

17. Dada una secuencia de 7 elementos que representa los 7 primeros t¶rminos de una progresi¶n geom¶trica:

    (a) Calcular la raz¶n.

    (b) Calcular 5 t¶rminos adicionales.

4.7 Bibliograf¶a

1. AHO, HOPCROFT & ULLMAN,\Data Structures and Algorithms". Addison Wesley Series in Computer Science.

2. KNUTH, D.,\The Art of Computer Programming. Vol 1. Fundamental Algorithms". Addison Wesley.

3. WIRTH, N.,\Algorithms+Data Structures=Programs". Prentice Hall

4. Manuales de PASCAL describiendo estructuras din¶micas (pointers).

Capítulo 5

El problema de la Búsqueda

5.1 Marco Conceptual

Consideremos el problema de implementar el operador esta del TAD Secuencia, definido en la sección 4.2 del capítulo 4. Este problema tiene bastante importancia ya que la búsqueda en una secuencia de gran tamaño puede tener un costo proporcional al tamaño de la secuencia ($O(N)$), que puede resultar una operación costosa [1].

5.1.1 Búsqueda Secuencial

Una manera de implementar el operador esta es realizar una búsqueda secuencial del elemento en la secuencia. Este método se basa en la idea de que para determinar si un elemento ocurre en una secuencia, se deben revisar uno a uno los elementos de ésta hasta encontrar el elemento o hasta agotarla (no está). De esta forma, si el elemento buscado ocupa la posición p, antes de encontrarlo, se habrán revisado los $p - 1$ elementos que ocurren antes que él en la secuencia.

Podemos considerar este método de búsqueda como un proceso repetitivo donde se mantiene una secuencia de elementos por revisar (originalmente la secuencia completa) y el resto son los elementos revisados (inicialmente la secuencia vacía) y que en cada paso de la búsqueda se logra acortar la secuencia por revisar.

[1] Si consideramos a la secuencia una forma de realización del conjunto o multiconjunto, el análisis del operador esta es equivalente a analizar el operador pertenece

$$\overbrace{\boxed{\text{Revisada}}}^{\text{long(Revisada)}}\overbrace{\boxed{\text{PorRevisar}}}^{\text{long(s)} - \text{long(Revisada)}}$$

$\uparrow$

1^{ra} pos. por revisar

Una solución al problema planteado es la siguiente:

```
function BusqSecuencial(PorRevisar:Secuencia;
                        e:Elemento): Boolean;
begin
     if (esvacia(PorRevisar)) then
        return(false)
     else
        if (e = proyectar(PorRevisar,1)) then
           return(true)
        else
           return(BusqSecuencial(eliminar(PorRevisar,1),e))
        fi
     fi
end;
```

La función `BusqSecuencial(s,e)`, busca el elemento e en la secuencia `s`.

El proceso termina ya que en cada nueva instancia de la función, la secuencia sobre la que se hace la búsqueda es de tamaño menor, en una unidad, que la secuencia de la instancia anterior. El proceso puede decidir si el elemento buscado estaba en la secuencia original ya que sólo se elimina un elemento de la secuencia a revisar cuando se tiene la certeza de que no es el elemento buscado.

Es de hacer notar que en este algoritmo no se hace uso de relaciones que pudieran existir entre los elementos. Sólo se requiere poder decidir si dos elementos son iguales que es la expresión usada en el segundo condicional.

En la sección siguiente se requerirá que sobre los elementos se pueda establecer un orden total.

5.1.2 Búsqueda Binaria

Consideremos ahora el caso particular en el cual la secuencia está ordenada crecientemente, según $\prec$ [2]. Esta caracterización de la secuencia, ofrece ciertas ventajas que permiten implementar un método de búsqueda más inteligente llamado búsqueda binaria.

La idea de este método es que la secuencia por revisar, de longitud N, se divide en dos subsecuencias: una, de longitud $[N/2] - 1$, donde se encuentran elementos menores o iguales al que ocupa la posición media, $[N/2]$, y la otra, de longitud $N - [N/2]$ donde se encuentran elementos mayores o iguales:

$$\boxed{\quad \le e_{med} \quad}\; e_{med} \;\boxed{\quad \ge e_{med} \quad}$$

$$\uparrow$$

elemento en la pos. media

Una vez particionada la secuencia por revisar de este modo, se compara si el elemento buscado es mayor, menor o igual a e_{med} para determinar si se detiene la búsqueda (son iguales) o dónde debe seguir buscando. Note que a diferencia de la búsqueda secuencial, en cada paso, se acorta mucho más la secuencia por revisar.

Para dar el algoritmo correspondiente a la búsqueda binaria, consideremos una nueva operación sobre las secuencias,

$$\mathrm{bisectar}: \ \mathrm{Secuencia} \ \Rightarrow \ \mathrm{Secuencia} \times \mathrm{Elemento} \times \mathrm{Secuencia}$$

$\mathrm{bisectar}(s)$, retorna una terna $(s_i; med; s_d)$

Supongamos que están definidas las proyecciones sobre una terna y que se llaman $\pi_1; \pi_2$ y π_3, para la primera, segunda y tercera componente, respectivamente.

donde,

- med, es la posición del elemento medio de la secuencia s.

  $long(s) > 1 \ \Rightarrow \ \pi_2(\mathrm{bisectar}(s)) = \mathrm{proyectar}(s; [long(s)/2])$

- s_i, es la subsecuencia de s donde se encuentran elementos menores o iguales que $\mathrm{proyectar}(s; med)$.

[2] El símbolo $\prec$ correponde a una relación de orden válida en el tipo elemento de la secuencia

$$\mathrm{long}(s) > 1 \wedge s_i = \pi_1(\mathrm{bisectar}(s)) \rightarrow \bigwedge_{j=1}^{\mathrm{long}(s_i)} (\mathrm{proyectar}(s_i; j) = \mathrm{proyectar}(s; j))$$

- s_d, es la subsecuencia de s donde se encuentran elementos mayores o iguales que proyectar(s; med).

$$\mathrm{long}(s) > 1 \wedge s_d = \pi_3(\mathrm{bisectar}(s)) \rightarrow \bigwedge_{j=1}^{\mathrm{long}(s_d)} (\mathrm{proyectar}(s_d; j) = \mathrm{proyectar}(s; \mathrm{long}(\pi_2(\mathrm{bisectar}(s))) + j))$$

Una vez definida la operación bisectar(s), presentamos el algoritmo de búsqueda binaria como sigue:

```
function BusqBinaria(PorRevisar:Secuencia;
                    e:Elemento): Boolean;
var e_med:Elemento; t: Terna;
begin
     if (esvacia(PorRevisar)) then
        return(false)
     else if (long(s) = 1) then
             return(proyectar(s,1) = e)
          else
             t := bisectar(PorRevisar);
             e_med:= proyectar(PorRevisar,pi2(t));
             if (e = e_med) then return(true)
             else if (e < e_med) then
                     return(BusqBinaria(pi1(t),e))
                  else
                     return(BusqBinaria(pi3(t),e))
                  fi
             fi
        fi
     fi
end;
```

Donde pi1, pi2 y pi3, se deben interpretar como $\pi_1; \pi_2$ y π_3, respectivamente, < como la relación $\prec$ y el tipo Terna = Secuencia $\times$ Elemento $\times$ Secuencia .

5.1.3 An¶alisis del algoritmo de bisecci¶on

Haciendo el an¶alisis del algoritmo de la secci¶on anterior podemos deducir su complejidad.

$$\begin{aligned} T_{bus}(n) &= T_{proy}(t) + T_{bis}(n) + \\ &\quad max(T_{bus}(long(pi_1(t))); T_{bus}(long(pi_3(t)))) \end{aligned} \qquad (5.1)$$

donde long(pi¤(t)) corresponde a la longitud de la secuencia que va a ser revisada en la b¶usqueda. Debido a que tenemos el m¶aximo (max), lo conveniente es que las dos secuencia ¼$_1$ y ¼$_3$ tengan apr¶oximadamente la misma longitud que representaremos por n=2. Adem¶as, considerando $t_{proy}(n)$ y $T_{bus}(n)$ de tiempo constante, resulta:

$$\begin{aligned} T_{bus}(n) &= c + T_{bus}(n=2) \\ &= 2c + T_{bus}(n=2^2) \\ &= rc + T_{bus}(n=2^r) \end{aligned}$$

Haciendo $n=2^r < 1$ tenemos $r > log(n)$, por lo que la complejidad nos queda:

$$T_{bus}(n) = c(log(n) + 1)$$

5.2 Posici¶on de los Elementos

Los algoritmos dados anteriormente, s¶olo determinan si un elemento e ocurre o no en una secuencia s. Sin embargo, en muchas aplicaciones es importante, recuperar la posici¶on que ocupa e en s; para lograr esto, podemos rede¯nir el operador buscar de la siguiente manera:

$$\text{buscar} : \text{Secuencia} \times \text{Elemento} \Rightarrow \text{Natural}$$

Debido a que las posiciones en una secuencia se numeran a partir del uno, utilizaremos el cero para indicar que un elemento no ocurre en una secuencia.

5.2.1 Posici¶n seg¶n la B¶squeda Secuencial

Para resolver el problema de determinar la posici¶n del elemento, necesitamos conocer la posici¶n del primer elemento a ser revisado en la secuencia original:

$$\overbrace{\boxed{\text{revisada}}}^{\text{long(revisada)}}\overbrace{\boxed{\text{por–revisar}}}^{\text{long(s) - long(revisada)}}$$

$\uparrow$

1^{ra} pos. por revisar
long(revisada) + 1

Lo que hacemos es agregar un tercer par¶metro, que mantenga en todo momento la longitud de la lista de elementos revisados, y de¯nimos la funci¶n BusqSecuencial1(s,e,p) como sigue:

```
function BusqSecuencial1(PorRevisar:Secuencia; e:Elemento;
                         LongRev:natural): Natural;
begin
    if (esvacia(PorRevisar)) then
       return(0)
    else if (e = proyectar(PorRevisar,1)) then
            return(LongRev+1)
         else
             return(BusqSecuencial(eliminar(PorRevisar,1),
                                   e,LongRev+1))
         fi
    fi
end;
```

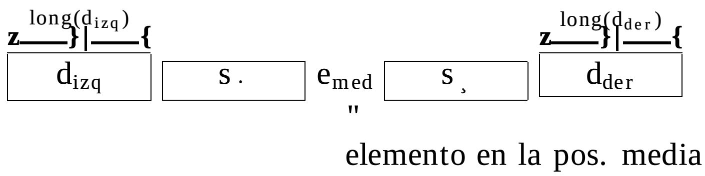
La funci¶n BusqSecuencial se de¯ne como sigue:

```
function BusqSecuencial(s,e):Natural;
begin
     return(BusqSecuencial1(s,e,0))
end;
```

debido a que en el par¶metro formal por_revisar se copia el valor de la secuencia s, la primera posici¶n de la secuencia a ser revisada corresponde exactamente a la primera posici¶n de la secuencia original. De este modo, se resuelve el problema de recuperar la posici¶n del elemento e en la secuencia s mediante b¶squeda secuencial.

5.2.2 Posici¶n seg¶n la B¶squeda Binaria

Consideremos la bisecci¶n de la secuencia por revisar en un instante cualquiera de la ejecuci¶n del algoritmo: tanto a la izquierda como a la derecha de la secuencia por revisar hay subsecuencias donde se ha descartado la b¶squeda que llamaremos d_{izq} y d_{der}, respectivamente:



En este caso, el problema se resuelve nuevamente agregando un tercer par¶metro a la funci¶n que mantenga la longitud de la subsecuencia de elementos descartados a la izquierda.

```
function BusqBinaria1(PorRevisar:Secuencia; e:Elemento;
                      LongDizq:Natural): Natural;
var e_med:Elemento; t:Terna;
begin
    if (esvacia(PorRevisar)) then return(0)
    else if (long(s) = 1) then
            if (proyectar(s,1) = e) then return(LongDizq+1)
            else return(0)
            fi
         else
            t:= bisectar(PorRevisar);
            e_med:= proyectar(PorRevisar,pi2(t));
            if (e = e_med) then return(LongDizq +1)
            else if (e < e_med) then
                 return(BusqBinaria1(pi1(t)),e,LongDizq)
                 else
                    return(BusqBinaria1(pi3(t),e,
                                      LongDizq+long(pi1(t)+1)))
                 fi
            fi
        fi
    fi
end;
```

La función BusqBinaria se define como sigue:

```
function BusqBinaria(s,e):Natural;
begin
    return(BusqBinaria1(s,e,0))
end;
```

Hasta este momento hemos analizado el problema de búsqueda en una secuencia y con las implementaciones vistas hasta ahora, el orden de la búsqueda es O(n). Si podemos organizar la secuencia mediante un orden (y podemos pagar por el costo de mantener ese orden en las inserciones) vimos que podíamos reducir la búsqueda a O(log(n)). Para poblaciones grandes este costo es muy alto. En capítulos posteriores (en particular en el cap. 7), usaremos el recurso de organizar los elementos (en general, consumiendo memoria adicional) de manera de poder realizar la operación de búsqueda más eficientemente.

5.3 Resumen del capítulo 5

En este capítulo hemos comenzado el análisis del problema de búsqueda, motivado por la frecuencia del uso de ese operador y por que un enfoque ingenuo de la solución puede dar tiempos inadmisibles para aplicaciones que requieran una respuesta a tiempo real (consulta en bancos, cajeros automáticos, etc.). En particular se ataca el problema introduciendo la técnica del precondicionamiento que consiste en hacer un trabajo previo (en este caso el ordenamiento de la secuencia) que si bien tiene un costo inicial alto hace que la operación de búsqueda pase de O(n) a O(ln(n)) lo que significa para poblaciones grandes una reducción drástica en el tiempo requerido para la ejecución de la búsqueda. Veremos sin embargo que hay casos en que esta velocidad no es suficiente o existen problemas con el mantenimiento del precondicionamiento (ordenamiento) que hacen buscar otros mecanismos mas efectivo para la realización de la operación de búsqueda como veremos en el cap. 7. Los algoritmos dados en este capítulo tienen una presentación recursiva. Esta presentación hace mas fácil su comprensión que la versión iterativa y por sobre todo simplifica la evaluación de su complejidad.

5.4 Ejercicios

1. Dado el siguiente predicado:

   $ord(s) \equiv$ Los elementos de la secuencia s están ordenados crecientemente según $\leq$.

   Decidir si son ciertas las siguientes proposiciones:

   (a) $ord(s) \rightarrow BusqSecuencial1(s; e; 0) = BusqBinaria1(s; e; 0)$

   (b) $ord(s) \rightarrow proyectar(s; buscar1(s; e; 0)) =$
   $proyectar(s; BusqBinaria1(s; e; 0))$

2. La compañía “Distribuidora Las Américas C.A.” mantiene un conjunto de clientes. Mensualmente, el número de clientes aumenta en un 5%, y decrece en 2.5% con respecto al total. Además, el promedio de consultas en el mes es del 40% del número de clientes.

   (a) Identifique el TAD asociado al conjunto de clientes.

   (b) Haga un análisis comparativo de las operaciones de ingreso, egreso y consulta de clientes, si el TAD se implementa como:

   (b.1) Secuencia Ordenada

   (b.2) Secuencia no ordenada

   En ambos casos, suponga que la secuencias se representan estáticamente.

   (c) Sugiera implementaciones alternas para el TAD. Haga un análisis comparativo.

3. Considere la ecuación (5.1) de la sección 5.1.3.

   (a) ¿Qué ocurre si con la complejidad si $T_{proy}(n)$ es de orden n.

   (b) ¿Qué ocurre en la misma ecuación si la función bisectar se cambia por una función cuya complejidad es lineal?

5.5 Bibliografía

1. KNUTH, Donald. “The Art of Computer Programming. Vol 3. Sorting and Searching.

2. WIRTH, Niklaus. \Algorithms + Data Structures = Programs". Prentice Hall.

Capítulo 6

Ordenamiento de una Secuencia

6.1 Marco Conceptual

El problema del ordenamiento de una secuencia de elementos de un dominio E, consiste en organizarlos ascendente o descendentemente según un orden total μ definido sobre E. En este capítulo nos ocuparemos solamente del ordenamiento ascendente utilizando el tipo abstracto secuencia definido en el capítulo 4.

La función SORT transforma secuencias en secuencias

$$\mathrm{SORT} : \mathrm{Secuencia} \to \mathrm{Secuencia}$$

Es posible representar el problema del ordenamiento mediante una expresión que defina el conjunto de soluciones. A esta expresión la llamaremos especificación del problema. En este sentido, los algoritmos de ordenamiento han sido planteados como una transformación de la secuencia a ordenar, x, para obtener una secuencia y, que satisfaga el predicado

$$y = \mathrm{SORT}(x) \Rightarrow \mathrm{perm}(x;y) \wedge \mathrm{ord}(y) \qquad (6.1)$$

donde perm(x,y) corresponde al predicado “La secuencia y es una permutación de la secuencia x”, y ord(y), al predicado “la secuencia y está ordenada”.

El primer predicado podemos caracterizarlo por

$$\forall i\ \exists j \Rightarrow \mathrm{proyectar}(x;i) = \mathrm{proyectar}(y;j) \qquad (6.2)$$

$$\forall i\ \exists j \Rightarrow \mathrm{proyectar}(y;i) = \mathrm{proyectar}(x;j) \qquad (6.3)$$

$$long(x) = long(y) \tag{6.4}$$

Este segundo predicado podemos caracterizarlo como:

$$i \leq j \Rightarrow proyectar(y, i) \neq proyectar(y, j) \tag{6.5}$$

En caso de existir claves repetidas, la expresión (6.1) caracteriza un conjunto de operadores que verifican las dos propiedades requeridas, y como operador no está univocamete definido. En realidad, caracteriza un conjunto de valores: la familia de operadores de clasificación.

Un método particular de ordenamiento obtendrá un valor específico dentro del conjunto solución usando un mecanismo determinado para obtener el elemento solución del problema.

6.1.1 Estabilidad de los algoritmos de ordenamiento

Cuando todas las claves por las que estamos ordenando son diferentes, cualquier algoritmo de ordenamiento dará el mismo resultado. Recordemos que mencionamos resultado y no el costo de alcanzar el resultado. No ocurre lo mismo cuando tenemos claves repetidas. Sea las secuencias x y s con las siguientes propiedades:

$perm(x, s)$ y $s = < s_1, s_2, \ldots, s_8 >$

donde s verifica

$$s_1 < s_2 < s_3 = s_4 = s_5 < s_6 < s_7 < s_8 > \tag{6.6}$$

por lo tanto

$$perm(x, s) \wedge ord(s)$$

Esta secuencia es solución de ordenar x, pero las secuencias

$$< s_1, s_2, s_5, s_4, s_3, s_6, s_7, s_8 >$$

$$< s_1, s_2, s_5, s_3, s_4, s_6, s_7, s_8 >$$

$$< s_1, s_2, s_3, s_5, s_4, s_6, s_7, s_8 >$$

$$< s_1, s_2, s_4, s_3, s_5, s_6, s_7, s_8 >$$

$$< s_1, s_2, s_4, s_5, s_3, s_6, s_7, s_8 >

son tambien soluciones de ordenar x.

Diremos que un método de clasificación es **estable** cuando, para claves iguales, preserva el orden que tenian los elementos en la secuencia de entrada. El método será estable si la solución es la secuencia de la ecuación 6.6.

6.1.2 Ejemplo

Sea la secuencia a clasificar conformada por elementos que pertenecen al producto cartesiano de:

$$Apellido \times Cedula \times NivelEducacional \times ExperienciaLaboral$$

donde un ejemplo de elemento puede ser:

$$(Perez; V - 3181437; 4; 12)$$

Si desamos visualizar a los empleados ordenados por apellido y en el caso de igual apellido, ordenados en forma creciente por años de experiencia laboral, el algoritmo de visualización pudiera ser ordenar primero por años de experiencia y luego por apellido. Si el método de clasificación es estable habremos logrado el objetivo. Habiendo ordenado primero por años de ExperienciaLaboral y luego por apellido, siendo el algoritmo estable, para todos los empleados Perez, los ordenará respetando el orden que traía la secuencia original, es decir en forma creciente de cantidad de años de ExperienciaLaboral. Si tomamos como secuencia de entrada

$$< s_8; s_6; s_1; s_4; s_5; s_3; s_7; s_2 >$$

y usando las relaciones de los elementos establecidas en 6.6

por lo que $s_3 = s_4 = s_5$, las únicas secuencias resultados posible con un ordenamiento estable serán aquellas secuencias que tengan la subsecuencia $< s_4; s_5; s_3 >$ lo que equivale a la secuencia:

$$< s_1; s_2; s_4; s_5; s_3; s_6; s_7; s_8 >$$

Veremos en los métodos de ordenamiento que la propiedad de estabilidad es una propiedad de la implementación de un método y no del método en si.

6.2 M¶todos de ordenamiento de Secuencias

En esta secci¶n nos ocuparemos de cuatro m¶todos cl¶sicos para el ordenamiento de secuencias, todos ellos basados en la construcci¶n de la secuencia ordenada partiendo de la secuencia vac¶a y aument¶ndola hasta que la secuencia (y) veri¯que $perm(x;y) \wedge ord(y)$.

En la descripci¶n de los algoritmos utilizaremos el TAD Secuencia.

Si bien en la presentaci¶n usaremos el TAD secuencia, esta familia de algoritmos est¶ particularmente adaptada a una implementaci¶n est¶tica de secuencias, en que no requiere mas espacio que el requerido para almacenar la secuencia original.

Para todos los m¶todos se inicia el algoritmo con dos secuencias, una de ellas desordenada y la otra ordenada, y en cada paso de la transformaci¶n lo que se logra es acortar la secuencia desordenada y alargar la secuencia ordenada hasta lograr que la secuencia desordenada sea vac¶a y la ordenada sea la soluci¶n al problema de ordenamiento.

B¶sicamente, los cuatro algoritmos di¯eren en la forma en que se elimina en cada paso un elemento de la secuencia desordenada y c¶mo se inserta el mismo elemento en la secuencia ordenada. En cada caso, hay un costo asociado a la eliminaci¶n y a la inserci¶n. Veremos, en diversas representaciones, que cuando se reduce uno de ellos se incrementa el otro.

DESORDENADA [1::i]	ORDENADA[i + 1::n]

Inicialmente la secuencia ordenada es vac¶a ($i = n$) (condici¶n inicial) y la condici¶n ¯nal de los algoritmos ser¶ que la secuencia desordenada se ha vaciado ($i = 0$) y todos los elementos de la secuencia original han ingresado en la secuencia ordenada.

Un invariante de estos m¶todos es, llamando n a la cardinalidad de la secuencia

$$long(DESORDENADA) + long(ORDENADA) = n \qquad (6.7)$$

6.2.1 por Selección

El mecanismo para el método de selección es tomar un elemento, eliminarlo de la secuencia desordenada e **insertarlo** en la secuencia ordenada, logrando así aumentar en uno la longitud de la secuencia ordenada. Por lo que debe repetirse este proceso hasta que no sea posible elegir un elemento de la secuencia desordenada (ya que esta será vacía). Una forma de facilitar la inserción, es **seleccionar** el máximo de la secuencia desordenada, y se coloca como primer elemento de la secuencia ordenada (ya que en cada selección, el elemento escogido es menor o igual que todos los elementos de la secuencia ordenada).

```
function Seleccion(desordenada,ordenada:Secuencia):Secuencia;
 begin
   if esvacia(desordenada) then retornar(ordenada)
   else return(
         Seleccion(eliminar(desordenada,max(desordenada)),
                   insertar(ordenada,1,
                            proyectar(desordenada,
                                      max(desordenada)))))
   fi
 end;
```

Para ordenar una secuencia A, se realizará una llamada a:

```
x = Seleccion(A,vacia)
```

En este algoritmo el operador max toma una secuencia y devuelve la posición donde se encuentra el elemento máximo.

Haciendo un análisis del algoritmo, vemos que efectivamente el algoritmo presentado es un algoritmo de clasificación. Requerimos de operadores auxiliares para el análisis. Sea conc el operador de concatenación de secuencias. La precondición del algoritmo es:

$$perm(A; conc(desordenada; ordenada)) \tag{6.8}$$

ya que inicialmente, llamandolo con

$$x = Seleccion(A; vacia); \; conc(A; vacia) = A$$

por lo que el predicado (6.8) se veri¯ca. Asimismo(6.3) es un invariante del algoritmo, ya que cada nueva llamada, lo que se ha transformado es que un elemento se elimina de la secuencia desordenada y se incluye en la secuencia ordenada, por lo que la concatenaci¶n de ambas secuencia sigue siendo una permutaci¶n de la secuencia original. Para asegurar que el algoritmo ordena efectivamente la secuencia deberemos demostrar que ord(ordenada) es verdadero.

Para todo par i,j ¶ndices v¶lidos para las secuencias

$$
\begin{array}{lcl}
\text{proyectar(desordenada; i)} & \cdot & \text{max(desordenada)} \\
\text{proyectar(desordenada; i)} & \cdot & \text{proyectar(ordenada; j)} \\
\text{max(desordenada)} & \cdot & \text{proyectar(ordenada; j)}
\end{array}
$$

Usando la propiedad de ord dada en (6.5) tendremos

$$\text{max(desordenada)} \quad \cdot \quad \text{proyectar(ordenada; j)}$$

y luego de insertar el max(desordenada) en la primera posici¶n de ordenada resultara:

$$\text{max(desordenada)} \quad \cdot \quad \text{proyectar(insertar(ordenada; 1; max(desordenada)); j)}$$

por lo la nueva ordenada veri¯ca la propiedad (6.5).

En cada momento de la recursi¶n, la concatenaci¶n del primer par¶metro y el segundo corresponden a una permutaci¶n de la secuencia inicial de entrada, ya que que el elemento que es eliminado de la primera es insertado en la segunda. Por lo que se veri¯ca

$$\text{perm(A; conc(desordenada; ordenada))}$$

conc se re¯ere al operador de concatenaci¶n de secuencias.

Finalmente, dado que desordenada es vac¶a tenemos que el resultado veri¯ca el predicado anterior y dada la construcci¶n de ordenada tambi¶n veri¯ca el predicado

$$\text{ord(ordenada)}$$

.

Estabilidad

Como hemos dicho en 6.1.1, la estabilidad del algoritmo de ordenamiento reside en preservar el orden relativo que tenian los elementos en la secuencia de entrada:

Si $x =< x_1; x_2; \ldots; x_n >$ y $perm(x; s) \wedge ord(s)$

(a) $s_1 \leq s_2 \leq s_3 \leq \cdots \leq s_n$

(b) $s_i = s_{i+1} \cdots = s_j$

(c) Si $s_i = x_k$
$s_{i+1} = x_r \;\; k < r$
$s_{i+2} = x_l \;\; r < l$
$\vdots$
$s_j = x_u$ siendo $u = max_t \{x_k = x_t\}$

En el algoritmo de selecci¶n, el elemento seleccionado es el max(desordenada), por lo que para lograr que el algoritmo sea estable requerimos que se extraiga como max, el elemento de desordenada con mayor ¶ndice, cuando sea el caso de claves repetidas. Bastar¶a con recorrer desde el principio la secuencia desordenada y cambiando la elecci¶n del m¶ximo siempre que el nuevo elemento fuera mayor o igual que el m¶ximo ya previamente seleccionado. Otro enfoque puede lograrse recorriendo la secuencia desordenada desde el ¶ltimo elemento hasta el primero.

6.2.2 Ordenamiento por Inserci¶n

El mecanismo para el m¶todo de inserci¶n es tomar un elemento (en principio cualquiera), eliminarlo de la secuencia desordenada e **insertarlo** ordenadamente en la secuencia ordenada, logrando as¶ hacer crecer en uno la longitud de la secuencia ordenada. Por lo que debe repetirse este proceso hasta que no sea posible elegir un elemento de la secuencia desordenada (ya que esta ser¶ vac¶a). Para simpli¯car la operaci¶n de selecci¶n el elemento que se elige es el ¶ltimo de la secuencia desordenada. Esta simpli¯caci¶n es v¶lida para el caso de la implementaci¶n de secuencias usando arreglos. Si la implementaci¶n de secuencias es listas simplemente enlazadas, la operaci¶n mas econ¶mica es tomar el primero de la lista desordenada.

```
function Insercion(desordenada,ordenada:Secuencia):Secuencia;
 begin
   if esvacia(desordenada) then return(ordenada)
   else return(
        Insercion(eliminar(desordenada,long(desordenada)),
                  InsertOrd(ordenada,
                           proyectar(desordenada,
                                     long(desordenada)))))
   fi
 end;
```

Donde la funci¶n InsertOrd inserta en la secuencia (primer argumento) el elemento (segundo argumento) de manera que si la secuencia de entrada estaba ordenada el resultado ser¶ una secuencia ordenada, que incluye al elemento insertado.

```
function InsertOrd(ordenada:Secuencia,e:Elemento):Secuencia;
 begin
   if esvacia(ordenada) then
      return(insertar(ordenada,1,e))
   else i:=1;
        while  (i <= long(ordenada)) and
                (proyectar(ordenada,i) < e) do
                i := i + 1
        od;

            return(insertar(ordenada,i,e))

   fi
 end
```

El while se detiene en un puesto donde hay que insertar el nuevo elemento (caso i <= long(ordenada) y hay alg¶n elemento en la secuencia que es mayor que el que se quiere insertar , o con i > long(ordenada) lo que indica que todos los elementos de la secuencia son menores que el elemento a insertar. Es por eso que i indica en cada uno de los caso el puesto donde debe insertarse el nuevo elemento para que la secuencia resultado este ordenada [1]

Para ordenar una secuencia A, se realizar¶ una llamada a:

```
x = Insercion(A,vacia)
```

Otra versi¶n posible para insertar puede ser la de buscar el puesto donde debe ser colocado el elemento mediante b¶squeda binaria y luego insertarlo.

Es importante hacer resaltar que el esquema de los dos algoritmos presentados es muy similar: en el primero (Selecci¶n) el trabajo de cada paso consiste en recorrer la parte desordenada para obtener el m¶ximo de los elementos, mientras que en el segundo el trabajo de cada paso es recorrer la parte ordenada para insertar el nuevo elemento.

Estabilidad

Por la presentaci¶n del algoritmo de inserci¶n, este es naturalmente estable si en la implementaci¶n conserva la propiedad que el elemento elegido para insertar sea el ¶ltimo de la secuencia ordenada y que la operaci¶n insertarOrd inserte adelante del elemento que en el recorrido de izquierda a derecha, no sea menor que el seleccionado para insertar. Es de hacer notar el mecanismo sugerido para hacer mas econ¶mica la selecci¶n en listas simplemente enlazadas, da como resultado una implementaci¶n no estable del algoritmo.

Si se utiliza el mecanismo de b¶squeda del puesto para insertar mediante b¶squeda binaria, la estabilidad no es inmediata ya que si el elemento est¶ ya en la parte ordenada se deber¶a buscar todos los elementos iguales de manera tal de insertar el nuevo elemento como primero del conjunto de elementos iguales.

6.2.3 Burbuja

El m¶todo de la burbuja hace que la secuencia de entrada sufra transformaciones de tal forma que los elementos m¶s pesados suban acerc¶ndose a sus puestos de¯nitivos. Esto se logra haciendo comparaciones de un elemento con su vecino en la secuencia desordenada y en el caso en que no est¶n ordenados se intercambian.

---

[1] Esta presentaci¶n del algoritmo, si bien es muy compacta requiere elementos del lenguaje de programaci¶n en que sea implementada: que las expresiones booleanas no eval¶en todos sus argumentos cuando el resultado parcial permita saber el valor de la expresi¶n. En el caso que nos ocupa si $i >$ long(ordenada) proyectar no est¶ de¯nida

```
function Burbuja(desordenada,ordenada:Secuencia):Secuencia;
 begin
   if esvacia(desordenada) then
      return(ordenada)
   else return(Burbuja(eliminar(Intercambio(desordenada),
                      long(Intercambio(desordenada))),
                   insertar(ordenada,1,
                           proyectar(Intercambio(desordenada),
                                    long(desordenada))))))
   fi
 end;
```

Y la función Intercambio, se define como sigue:

```
function Intercambio(s: Secuencia): Secuencia;
 begin
   if (long(s) > 1) then
     if (proyectar(s,1) > proyectar(s,2)) then
        return(insertar(Intercambio(eliminar(s,2)),1,
                        proyectar(s,2)))
     else
        return(insertar(Intercambio(eliminar(s,1)),
                        1, proyectar(s,1)))
     fi
   else
     return(s)
   fi
 end;
```

Nótese que en una implementación usando arreglos, la eliminación del último elemento en la primera secuencia y la inserción en la segunda se traducirían en cambiar la frontera entre ambas.

Una propiedad que puede tomarse como post-condición de la invocación y:=Intercambio(s) es que

$$1 \le j \le long(y) \quad proyectar(y; j) \le proyectar(y; long(y)) \tag{6.9}$$

Explotando esta propiedad es podemos aseverar que

$$\forall (j,k)\ \ proyectar(desordenada, j) \le proyectar(ordenada, k) \qquad (6.10)$$

y debido a la forma de hacer crecer desordenada tenemos que otro invariante es

$$ord(ordenada) \qquad (6.11)$$

Con lo anterior demostramos que construimos usando el método de la burbuja la secuencia ordenada.

Puede reconocerse nuevamente en este algoritmo, la partición de la secuencia en dos subsecuencias, la de la izquierda desordenada y la de la derecha ordenada.

DESORDENADA [1..i+1]	ORDENADA[i+2..n]

Estabilidad

Si la implementación del algoritmo de la burbuja toma en cuenta no intercambiar elementos iguales, el máximo elemento de la secuencia desordenada ocupará la última posición de la secuencia desordenada. Asimismo, si este valor esta duplicado, estos valores estarán en las posiciones anteriores conservando el orden relativo que tenían en la entrada.

6.2.4 HeapSort

Este método de ordenamiento usa el mismo principio de comenzar con una secuencia vacía que va creciendo hasta tener todos los elementos de la secuencia inicial, diferenciándose de los métodos anteriores en que no se inicia el proceso de crecimiento de la secuencia ordenada a partir de una secuencia desordenada cualquiera sino de una semi-ordenada: un heap, lo cual constituye una pre-condición para la función MetodoHeap.

Definición: Un heap es una secuencia $< h_1, h_2, \ldots, h_n >$, que cumple la siguiente propiedad:

$$\forall i\ \ 1 \le i < n/2\ (h_i \le h_{2i} \wedge h_i \le h_{2i+1})$$

Un ejemplo de heap de 9 elementos (este caso será $4 < n/2$):

$$a = \langle 3, 5, 7, 9, 11, 14, 8, 10, 10 \rangle$$

ya que para todo índice entre $1, \ldots, 4$:

$a_1 \leq a_2$ y $a_1 \leq a_3$
$a_2 \leq a_4$ y $a_2 \leq a_5$
$a_3 \leq a_6$ y $a_3 \leq a_7$
$a_4 \leq a_8$ y $a_4 \leq a_9$

Que una secuencia tenga una organización de heap no significa que este ordenada, pero por las propiedades de heap, el primer elemento de la secuencia es el min(a).

En general, un heap corresponde a un árbol parcialmente ordenado por lo que el primer elemento suele llamarse raíz del heap. Estos objetos los estudiaremos en el capítulo 8

```
function MetodoHeap(heap,ordenada:Secuencia):Secuencia;
begin
   if esvacia(heap) then return(ordenada)
   else
      return(MetodoHeap(heapify(sacar(heap)),
              insertar(ordenada,long(ordenada)+ 1,
                                 proyectar(heap,1)))
   fi
end;
```

La función sacar devuelve una secuencia donde se ha reemplazado el primer elemento del heap por el elemento que ocupaba la última posición. Nótese que la secuencia desordenada decrece al aplicarsele esta operación.

```
function sacar(s: Secuencia): Secuencia;
var s': Secuencia; l': Natural;
begin
    s' := eliminar(s,1);
    l' := long(s');
```

```
    return(insertar(eliminar(s',l'),1,proyectar(s',l')))
end;
```

Nótese que la secuencia resultante podría no ser un heap. La función `heapify` devuelve un heap construido a partir de la secuencia resultante de la función `sacar`.

El algoritmo de ordenamiento de una secuencia consistirá en transformar la secuencia desordenada en un heap y luego utilizar `MetodoHeap` para completar el ordenamiento.

```
function HeapSort(desordenada:Secuencia):Secuencia;
begin
   return(MetodoHeap(HacerHeap(desordenada),vacia)))
end;
```

Donde `HacerHeap` recibe una secuencia y devuelve una secuencia que verifica las propiedades de un heap, dejando en el primer elemento de la secuencia el mínimo del conjunto.

Estabilidad

El método es inestable. Esta propiedad lo hace útil solamente cuando se está ordenando una sola vez una secuencia.

6.3 Análisis de los algoritmos presentados

6.3.1 Análisis de los Métodos de Ordenamiento sin Precondicionamiento

En los dos primeros algoritmos (Inserción y Selección), el esquema de solución es idéntico: se chequea la condición de parada, y si no se cumple, se hace hace una llamada recursiva, modificando las dos secuencias de entrada, como se ve en los siguientes trozos de código:

```
Seleccion(eliminar(desordenada,max(desordenada)),
          insertar(ordenada,1,proyectar(desordenada,
                                        max(desordenada))))
```

```
Insercion(eliminar(desordenada,long(desordenada)),
          InsertOrd(ordenada,
          proyectar(desordenada,long(desordenada))))
```

considerando que el máximo se obtiene una sola vez, la complejidad del algoritmo `Seleccion`, $T_s(n)$, estará dada por la evaluación de los argumentos y el costo de la llamada recursiva:

$$T_s(n) = \begin{cases} c & \text{si } n = 0 \\ T_e(n) + T_m(n) + T_i(N - n; 1) + T_p(n) + T_s(n - 1) & \text{si } n > 0 \end{cases}$$

donde $T_e(n)$, $T_m(n)$, y $T_p(n)$ corresponden a las funciones de complejidad de las funciones `eliminar`, `max`, y `proyectar`, respectivamente. $T_i(N - n; 1)$, corresponde a la complejidad de la función `insertar`, específicamente en la primera posición de una secuencia. n es el tamaño de la secuencia desordenada en una instancia cualquiera de `Seleccion` y N es el tamaño del la secuencia desordenada en la primera instancia.

Resolviendo $T_s(n)$ para $n > 0$ queda:

$$\begin{aligned} T_s(n) &= T_e(n) + T_m(n) + T_i(N - n; 1) + T_p(n) + T_s(n - 1) \\ &= T_e(n) + T_m(n) + T_i(N - n; 1) + T_p(n) + T_e(n - 1) + \\ &\quad T_m(n - 1) + T_i(N - n + 1; 1) + T_p(n - 1) + T_s(n - 2) \\ \vdots \\ &= \sum_{j=0}^{i-1} T_e(n - j) + \sum_{j=0}^{i-1} T_m(n - j) + \sum_{j=0}^{i-1} T_i(N - n + j; 1) + \\ &\quad \sum_{j=0}^{i-1} T_p(n - j) + T_s(n - i) \\ \vdots \\ T_s(n) &= \sum_{j=0}^{n-1} T_e(n - j) + \sum_{j=0}^{n-1} T_m(n - j) + \sum_{j=0}^{n-1} T_i(N - n + j; 1) + \\ &\quad \sum_{j=0}^{n-1} T_p(n - j) + c \end{aligned}$$

Para calcular el orden de $T_s(n)$ debemos hacer el análisis bajo dos presuposiciones:

(A) La secuencia est¶ representada est¶ticamente. Esto implica que $T_e(n)$, $T_m(n)$ y $T_i(n; 1)$ son $O(n)$, mientras que $T_p(n)$ es $O(1)$. Por lo tanto, nos queda:

$$
\begin{aligned}
T_s(n) &= O(\textstyle\sum_{j=0}^{n-1}(n - j) + \sum_{j=0}^{n-1} +(n - j) \sum_{j=0}^{n-1}(N - n + j) + \\
&\quad \textstyle\sum_{j=0}^{n-1} 1 + 1) \\
&= O(2 \textstyle\sum_{j=0}^{n-1}(n - j) + Nn - n^2 + 1/2(n^2 - n) + n + 1) \\
&= O(2n^2 - n^2 + n + Nn - n^2 + 1/2(n^2 - n) + n + 1) \\
&= O(n^2/2 + Nn + 3/2n + 1)
\end{aligned}
$$

Recordemos que el tama˜no de la secuencia a ordenar n, en la llamada original es igual a N, por lo tanto, reemplazando en $T_s(n)$ nos queda:

$$T_s(N) = O(3/2N^2 + 3/2N + 1) = O(N^2)$$

(B) La secuencia est¶ representada din¶micamente. En este caso, $T_e(n)$, $T_m(n)$ y $T_p(n)$ son $O(n)$, mientras que $T_i(n; 1)$ es $O(1)$.

Para calcular el $T_s(n)$, se procede de manera an¶loga al caso anterior.

Como puede verse el m¶todo resulta de orden cuadr¶tico. En el caso de Insercion se puede reducir un poco el tiempo de ejecuci¶n del algoritmo haciendo la inserci¶n mediante b¶squeda binaria, sin embargo no se cambia el orden del m¶todo.

Partiendo de la misma descripci¶n de los algoritmos, vemos que el algoritmo Seleccion se llama recursivamente tantas veces como la longitud de la secuencia original (N) y en cada llamada la longitud de la secuencia ordenada la llamaremos i y por lo tanto la secuencia desordenada tendr¶ una longitud de $N - i$. En el cuerpo del algoritmo se realizan las operaciones:

- 2 veces la operaci¶n de **max(desordenada)**
- la operaci¶n **eliminar(desordenada,k)** donde k es la posici¶n del m¶ximo elemento de desordenada
- la operaci¶n insertar(ordenada,1, proyectar(desordenada, k))

Usando los operadores de secuencia, la secuencia desordenada perdería un elemento en el proyectar, siendo esa operación en una representación estática de $O(i)$ y la operación de insertar $O(N - i)$, dando como resultado un costo de $O(N)$. Usando un operador adicional al TDA secuencia de SWAP, siendo este operador

TAD Secuenciaconswap[Elemento]

Sintaxis

vacia :		$\Rightarrow$	Secuencia
esvacia :	Secuencia	$\Rightarrow$	Boolean
insertar :	Secuencia $\times$ Natural $\times$ Elemento	$\Rightarrow$	Secuencia
eliminar :	Secuencia $\times$ Natural	$\Rightarrow$	Secuencia
proyectar :	Secuencia $\times$ Natural	$\Rightarrow$	Elemento
esta :	Secuencia $\times$ Elemento	$\Rightarrow$	Boolean
long :	Secuencia	$\Rightarrow$	Natural
MAX :	Secuencia $\times$ Natural $\times$ Natural	$\Rightarrow$	Natural
SWAP :	Secuencia $\times$ Natural $\times$ Natural	$\Rightarrow$	Secuencia

Semántica

$\forall s \in \text{Secuencia}; p; p_1; P \in \text{Natural} P \neq p; P \neq p_1; e; e_1 \in \text{Elemento};$

1. si $\text{proyectar}(s; p) = e$ y $\text{proyectar}(s; p_1) = e_1$

$$\text{proyectar}(\text{SWAP}(s; p; p_1); p) = e_1$$

$$\text{proyectar}(\text{SWAP}(s; p; p_1); p_1) = e$$

$$\text{proyectar}(\text{SWAP}(s; p; p_1); P) = \text{proyectar}(s; P)$$

2. $\text{proyectar}(s; p) \leq \text{proyectar}(s; \text{MAX}(s; 1; \text{long}(s)))$

Podemos transformar el algoritmo:

```
function Seleccion(desordenada,ordenada:Secuencia):Secuencia;
 begin
   if esvacia(desordenada) then retornar(ordenada)
   else return(
        Seleccion(eliminar(desordenada,max(desordenada)),
                 insertar(ordenada,1,
                         proyectar(desordenada,
```

```
                                              max(desordenada)))))
   fi
 end;
```

en el algoritmo que almacena en una secuencia de longitud N el par de secuencias (ordenada y desordenada), y recibiendo como parámetros la secuencia y el entero que indica el último elemento de la secuencia desordenada, sobreentendiendo que la secuencia ordenada se inicia en la siguiente posición.

```
function Seleccion(s:Secuencia,i:Natural):Secuencia;
 begin
 if esvacia(desordenada) then retornar(ordenada)
   if i = 0 then return(s)
   else   begin
                  k:= MAX(desordenada,1,i);
                 retornar(Seleccion( SWAP(s,k,i),i-1))
          end
   fi
 end;
```

haciendo el análisis de esta versión, llamando T_w al tiempo de SWAP (que es constante) y $T_m(k)$ al tiempo requerido para calcular el máximo en una secuencia de k elementos:

$$
\begin{aligned}
T_s(n) &= T_w + T_m(n) + T_s(n-1) \\
&\vdots \\
T_s(n) &= \sum_{j=0}^{n-1} T_w + \sum_{j=0}^{n-1} T_m(n-j) \\
T_s(n) &= \sum_{j=0}^{n-1} T_w + \sum_{j=0}^{n-1} (n-j) \\
&\vdots \\
T_s(n) &= n \cdot T_w + \frac{n \cdot (n+1)}{2}
\end{aligned}
$$

La complejidad encontrada es nuevamente cuadrática, sin embargo las constantes son mucho menores que en los análisis precedentes. Esta versión del algoritmo explota la propiedad de los arreglos de poder intercambiar el último elemento de desordenada con su máximo, alargando así la longitud de la parte ordenada.

Se deja al lector como ejercicio el cálculo del T(n) para los métodos de inserción y burbuja.

6.3.2 Análisis del HeapSort

El análisis de este método lo haremos bajo la suposición de que las secuencias se representan estáticamente. La función HacerHeap construye el heap a partir de la secuencia desordenada. El costo asociado a la misma es O(n), siendo n la longitud de la secuencia desordenada. Falta ver el costo asociado a la función MetodoHeap para aplicar la regla de la suma y obtener el orden del HeapSort.

```
MetodoHeap(heap,ordenada:Secuencia):Secuencia;
begin
   si esvacia(heap) entonces return(heap)
   sino
      return(MetodoHeap(heapify(sacar(heap)),
            insertar(ordenada,1,
                              proyectar(heap,1)))
end;
```

se reemplazo long(ordenada) por 1

Si reconocemos que utilizando representación estática, para obtener los nuevos argumentos para la recursión basta con dividir el arreglo en dos partes: la parte izquierda correponderá a la secuencia desordenada (heap) y la parte derecha a secuencia ordenada.

HEAP [1..i+1]	ORDENADA[i+2..n]

La función sacar consistiría en intercambiar el primero y el último elemento del heap, 1 y i+1, respectivamente, y en ese caso todo el trabajo se reduciría a crear el heap inicial y a llevar a su posición (heapify) el elemento mal ubicado en la raíz del heap. El proceso heapify cuesta O(log(n)). Nótese que tal como se describió el proceso sacar arriba, se obtiene una secuencia ordenada decrecientemente, pero se preserva O(1) en la inserción del elemento seleccionado en la secuencia ordenada. Para obtener una secuencia ordenada ascendentemente se trabaja con un max-heap donde se pide:

$$\forall 1 \leq i < n/2\ (h_i \geq h_{2i} \wedge h_i \geq h_{2i+1})$$

Como se hacen n inserciones y estas se realizan en O(log(n)) la complejidad de MetodoHeap resulta O(n log(n)). Aplicando la regla de la suma, nos queda que el orden de HeapSort es O(nlog(n)).

6.4 Otros algoritmos de ordenamiento

Existe una variedad de enfoques adicionales para ordenar una secuencia. Los trabajados en las secciones anteriores est¶n particularmente adaptados para ordenar una secuencia que se almacena en su totalidad en la memoria principal del computador. Otros algoritmos pueden adaptarse facilmente a los casos en que la memoria principal es insu¯ciente para almacenar toda la secuencia. Mencionaremos dos m¶todos, uno particularmente adaptado a memoria secundaria que recibe el nombre de Merge Sort y el segundo llamado Quick Sort que se caracteriza generalmente en tener un buen comportamiento en tiempo de ejecuci¶n, pero posee el inconveniente que para ciertas secuencias su comportamiento puede decaer sensiblemente.

6.4.1 Merge Sort

El esquema de este m¶todo de ordenamiento se basa en el enfoque de "Divide y reinar¶s".

Si llamamos MERGESORT al operador que toma una secuencia y devuelve la secuencia ordenada deber¶ este operador veri¯car las condiciones dadas en 6.1.

Usaremos dos operadores auxiliares que llamaremos SPLIT y MERGE con la siguiente ¯rma:

SPLIT :	Secuencia	)	(Secuencia; Secuencia)
MERGE :	(Secuencia; Secuencia)	)	Secuencia

Sem¶ntica

8s 2 Secuencia; $(p; p_1)$ 2 Secuencia £ Secuencia; e 2 Elemento
i 2 Natural

1. SPLIT(s) = $(p; p_1)$!
   long(p) aproximadamente igual a long(p_1) ^
   perm(p$k$$p_1$; s) = true

2. MERGE($p; p_1$) = s !
   ORD(p) = true ^ ORD(p_1) = true ^ ORD(s) = true
   ^ perm(p$k$$p_1$; s) = true

Ahora estamos en condiciones de describir la sem¶ntica de MERGESORT:
MERGESORT(vacia) = vacia

MERGESORT(s) =
MERGE(MERGESORT(¼$_1$(SPLIT(s));MERGESORT(¼$_2$(SPLIT(s)))

Usando las propiedades de los operadores MERGE, SPLIT dados en 1 y 2 podemos veri¯car que el operador MERGESORT veri¯ca las condiciones requeridas en 6.1 para que un m¶etodo sea de ordenamiento. El operador de MERGE puede ser implementado en O(n), el operador de SPLIT en O(n), por lo que resulta que el algoritmo de MERGESORT es de O(n log(n)).

6.4.2 Quick sort

Este algoritmo es muy similar a MERGESORT, s¶olo di¯ere en el operador de separaci¶on de la secuencia, que se realiza mediante la elecci¶on de un elemento (que puede o no ser un elemento de la secuencia),que se denomina pivote y cuyo valor debe ser pr¶oximo a la mediana de la secuencia, de manera tal que la secuencia sea dividida en dos subsecuencias de aproximadamente la misma longitud. El operador de separaci¶on SPLIT1

1. SPLIT1(s; e) = (p; p$_1$) !
   proyectar(p; i) · e ^ proyectar(p$_1$; i) > e ^
   perm(pkp$_1$; s) = true

Ahora estamos en condiciones de describir la sem¶antica de QUICKSORT:

QUICKSORT(s) =
QUICKSORT(¼$_1$(SPLIT1(s; e))kQUICKSORT(¼$_2$(SPLIT1(s; e))

Si observamos la ecuaci¶on 6.4.2 que como SPLIT1 es un operador que hace que la primera secuencia del resultado tiene elementos menores o iguales al pivote y los de la segunda secuencia son mayores que el pivote, nos basta concatenar las secuencias resultantes de las llamadas al QUICKSORT para tener una secuencia ordenada.

El valor de e utilizado para la realizaci¶on del operador SPLIT1 es cr¶³tico para asegurar el orden del algoritmo. En particular, si se logra en cada uso del operador SPLIT1 una buena elecci¶on del elemento que divide a las secuencias (por ejemplo, divide la secuencia en dos secuencias de aproximadamente la misma longitud) el algoritmo de QUICKSORT es de O(n log(n)). Sin embargo si e es un elemento que hace que long(¼$_2$(SPLIT(s; e)) = 0, la expresi¶on en 6.4.2 es una computaci¶on in¯nita. En el caso en que long(¼$_1$(SPLIT(s; e)) = 1 el algoritmo de MERGESORT ser¶a de O(n^2). Existen varias t¶ecnicas seguras (en el sentido de apartarnos de un caso degenerado o del caso de orden O(n^2)) de elecci¶on del pivote. Si se conoce una distribución de las claves a ordenar puede hacerse una buena aproximación a la mediana. Otra forma es tomar el promedio entre dos valores de la secuencia. En todo caso el costo asociado a la elección del pivote debe ser de $O(1)$, ya que si se invirtiera mayor trabajo en la elección el método dejaría de ser $O(n \log(n))$.

6.5 Resumen del capítulo 6

En este capítulo se revisaron algunos métodos de ordenamiento, básicamente los llamados métodos de ordenamiento interno, sin y con precondicionamiento, haciendo el análisis de su complejidad. Asimismo se revisa la propiedad de estabilidad que asegura la posibilidad de realizar ordenamientos sucesivos, sobre diferentes claves, logrando ordenar un conjunto de elementos por tuplas de claves.

Haciendo una recapitulación de los métodos vistos en este capítulo, podríamos llegar a la conclusión que el algoritmo de HEAPSORT es el mas eficiente, lo cual es erroneo. Existen en la literatura otros algoritmos de orden $O(n\log(n))$, tal como Quicksort que en el caso promedio es $O(n\log(n))$, siendo su peor caso $O(n^2)$, siendo tambien un método no estable, el Shellsort con una complejidad de $O(n^{1.25})$. Es importante recordar que en el capítulo 1 en la sección 1.2.2 vimos que cualquiera sea el método de ordenamiento que utilice comparaciones de claves para ordenar, requerirá al menos $O(n\log(n))$.

6.6 Ejercicios

1. Dada la siguiente definición del predicado mismo:

$$\text{mismo}(x, z) = \begin{cases} \text{vedadero} & \text{si } \text{long}(x) = \text{long}(z) \wedge \forall i \exists j (x_i = z_j) \\ \text{falso} & \text{sino} \end{cases}$$

   Indique si la definición anterior es equivalinte a decir que el predicado dará como resultado verdadero solamente cuando los dos argumentos sean iguales. Justifique su respuesta.

2. Ordene las siguientes secuencias, utilizando los algoritmos estudiados en este capítulo:

(a) $< 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 >$

(b) $< 1, 3, 5, 7, 9, 11, 2, 4, 6, 8, 10, 12 >$

(c) $< 12, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11 >$

(d) $< 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 >$

(e) $< 1, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2 >$

(i) Analice el número de comparaciones y de intercambios obtenidos con cada algoritmo.

(ii) Elabore las gráficas de rendimiento de cada una de las secuencias según el algoritmo ejecutado (número de comparaciones vs número de elementos en la secuencia).

3. Considere los siguientes patrones de la secuencia:

$$s = < s_1\, s_2 \cdots s_n >$$

donde n es par:

(a) $s_2 < s_3 < \cdots < s_{n-1}$;
$s_1 > s_{n-1}$; $s_n < s_2$

(b) $s_1 < s_3 < \cdots < s_{n-1}$;
$s_2 > s_4 > \cdots > s_n$

(c) $s_1 < s_3 < s_4 < \cdots < s_{n-2}$;
$s_{n-1} > s_n$;
$s_{n-1} < s_1$;
$s_2 > s_{n-2}$

Para cada patrón, determinar el número de movimientos y de comparaciones si la secuencia s se ordena con los métodos presentados en este capítulo.

4. Considere la siguiente secuencia:

$$s = < s_1, s_2, \ldots, s_n >$$

donde

(a) $s_1 < s_2 < \cdots < s_{(n/2)-1}$

(b) $s_{(n/2)+2} < s_{(n/2)+3} < \cdots < s_n$

(c) $s_{n/2} < s_1$

(d) $s_n < s_{(n/2)+1}$

Determinar el número de movimientos y de comparaciones si la secuencia s se ordena con los métodos presentados en este capítulo.

5. Considere la siguiente secuencia:

$$s = < s_1, s_2, \ldots, s_n >$$

donde

(a) $s_1 < s_3 < s_5 \cdots$ es una subsecuencia de s estrictamente decreciente.

(b) $s_2 > s_4 > s_6 > \cdots$ es una subsecuencia de s estrictamente creciente.

Determinar el número de movimientos y de comparaciones si la secuencia s se ordena con los métodos presentados en este capítulo.

6. El Club de Gimnasia de la USB, ha extendido el horario de las clases de Aerobics que ofrece cada trimestre, por lo que la demanda de dichas clases ha aumentado. Para reforzar este aumento, el Club ha creado un Plan de Oferta que consiste en hacer un descuento del 20% a aquellas personas que se inscriban por un período de un año para tomar dichas clases. Las inscripciones normalmente se realizan a principios de trimestre, aunque hay oportunidad de inscribirse en el transcurso del mismo.

El Club de Gimnasia desea tener un pequeño sistema que le permita verificar automáticamente los datos de las personas inscritas y controlar el número de personas por clase. El conjunto de personas inscritas hasta una cierta fecha se mantendrán en orden creciente, por número de carnet, en una secuencia. Los datos de las personas que se inscriban a "destiempo" (en el transcurso del trimestre) se mantendrán por orden de inscripción en otra secuencia. Al final de cada trimestre hay que unir ambos vectores, y eliminar del conjunto de personas inscritas a aquellas a las que se les haya vencido su período. Se desea que usted elija, de los algoritmos considerados en el poblema anterior, aquel que resulte más eficiente para esta operación. Justifique su elección.

7. Suponga que las claves que se utilizan para el ordenamiento de una secuencia poseen la caracter¶stica que comparar dos claves tiene un costo muy superior al de una comparaci¶n de enteros.

   i) En la versi¶n del m¶todo de inserci¶n haga una modi¯caci¶n que reduzca, para el caso promedio, el n¶mero de comparaciones de claves necesarias para lograr el ordenamiento de la secuencia.
   ii) Encuentre una secuencia de entrada que usando la versi¶n modi¯cada del algoritmo requiera mayor n¶mero de comparaciones de claves que cuando es clasi¯cada con la versi¶n original.
   iii) Muestre que si se trata de implementar el m¶todo de QUICKSORT calculando en cada paso la elecci¶n del pivote como la mediana del conjunto, el m¶todo ya no es de orden O(n log(n)).

8. Muestre que si se trata de implementar el m¶todo de QUICKSORT calculando en cada paso la elecci¶n del pivote como la mediana del conjunto, el m¶todo ya no es de orden O(n log(n)).

6.7 Bibliograf¶a

1. AHO, Alfred, HOPCROFT,John & ULLMAN, Je®rey. \Data Structures and Algorithms". Addison Wesley Series in Computer Science. 1985.

2. BAASE, Sara. \Computer Algorithms. Introduction to Design and Analysis". Addison Wesley Series in Computer Science. 1978.

3. BINSTOCK,Andrew & REX, John. \Practical Algorithms for Programmers". Addison Wesley Series in Computer Science. 1995.

4. CLARK, K.L. & DARLINGTON,J.\Algorithm Clasi¯cation through Synthesis". The Computer Journal. Vol. 23. No. 1. 1980.

5. GONNET, G.H. & BAEZA-YATES, R. \Handbook of Algorithms and Data Structures in Pascal and C". 2_{nd}Ed. Addison Wesley Series in Computer Science, 1.991.

6. KNUTH, D. \The Art of Computer Programming, Vol. 3: Sorting and Searching". Addison Wesley Series in Computer Science, 1.973.

7. LUCENA, C.J. & PEQUEÑO, T.M. \An approach for Data Type Speci¯cation and its Use in Program Veri¯cation". Information Processing Letters. Vol. 8. No. 2. Feb. 1979.

8. WIRTH, Niklaus. \Algorithms and Data Structures". Prentice Hall, 1.986.

Capítulo 7

TAD Diccionario

7.1 Marco Conceptual

Consideremos el diccionario de la Real Academia de la Lengua Española y llamémosle C; periódicamente esta institución **incorpora** nuevas palabras a C. También hay palabras que caen en desuso y son entonces **desincorporadas** de C.

Sin embargo, la mayor utilidad de C y en general del diccionario de cualquier idioma consiste en que mantiene la relación existente entre las palabras y el significado de cada una de ellas, debido a esto, la operación que se realiza con más frecuencia sobre un diccionario es la **recuperación** del significado de una palabra. Evidentemente, para poder hacer esto último es necesario saber cuál es la palabra a la que se le va a buscar el significado.

Si imaginamos una página de C vemos que las palabras se listan en orden lexicográfico y justificadas a la izquierda. A derecha de cada una de ellas encontramos párrafos donde se dan las diferentes acepciones que puede tener. Si consideramos estos párrafos como **el significado** de cada palabra, podemos imaginarnos a C como una colección de pares ordenados de la forma:

$$(palabra; conjuntodesignificado)$$

con la característica de que los pares no se repiten y si consideramos la primera proyección de un par, palabra, ésta no ocurre en ningún otro como primera proyección porque sólo tiene un significado (recordemos que todas las acepciones de una palabra constituyen su significado). Esta propiedad de los pares de la relación nos dice que más que una relación, C es una función.[1] Ahora bien, así como hemos hablado del diccionario de un idioma, como un objeto donde la operación fundamental es la recuperación de información, podemos extender esta noción a dominios diferentes al dominio de las palabra y sus significados. Podemos hablar entonces de diccionarios de estudiantes de la USB, de empleados de una empresa, de vehículos registrados en un estado, de cursos de una carrera, personas de un país, etc. En cada uno de los ejemplos antes mencionados observamos el hecho de que estamos hablando de conjuntos, por lo tanto, no existen elementos repetidos. Consideremos el diccionario de estudiantes de la USB, llamémosle E_{USB}, y supongamos que las coordinaciones de las diferentes carreras realizan anualmente un proceso (computarizado) que determina para cada estudiante un plan para finalizar la carrera lo antes posible y lo envia a la dirección de habitación de cada estudiante. Para realizar el mencionado proceso, es necesario recuperar para cada estudiante, a partir del carnet, toda la información necesaria: cursos aprobados, carrera a la que pertenecen y dirección de habitación. Si utilizamos nuestro esquema de pares para los elementos de un diccionario, E_{USB} puede visualizarse como sigue:

$$(carnet_1; datos_1)$$

$$(carnet_2; datos_2)$$

$$(carnet_3; datos_3)$$

$$\vdots$$

$$(carnet_N; datos_N)$$

La característica que queremos resaltar y que antes no vimos en C es el hecho de que los elementos del dominio de E_{USB} forman parte de los elementos del rango, ya que al igual que el nombre, el apellido, la dirección, etc. el carnet es un dato más de un estudiante; en general el rango de un diccionario es de un tipo compuesto, por ejemplo, en la información de una persona se encuentra su cédula de identidad, nombre, apellido, etc.; en la de un automóvil se encuentra su número de matrícula, color, modelo, etc.

---

[1]Una función $f : A \to B$ es una relación binaria, R_f, de A en B tal que para todo $a \in A$ existe un único $b \in B$ para el cual $(a; b) \in R_f$. Si existe un elemento en A que no está relacionado con ningún elemento de B, entonces f es una función parcial.

Ahora bien, los elementos del dominio de un diccionario no se escogen al azar entre los tipos que componen el tipo del rango, sino que se escoge entre los que tengan la propiedad de que identifica unívocamente un elemento, lo cual como ya hemos visto, ocurre con el carnet de los estudiantes, la cédula de identidad de las personas, los números de matrícula de los vehículos, etc. Usualmente los elementos del dominio de un diccionario se denominan **claves**. El tipo del dominio de un diccionario lo llamaremos tipo clave y al del rango tipo elemento.

En resumen, un diccionario es una colección de pares ordenados D que satisfacen la propiedad de ser una función,

$$D : TipoClave \Rightarrow TipoElemento$$

donde la operaciones de insertar nuevos items y recuperar información, pueden interpretarse como definir y aplicar la funcion D, respectivamente. El operador eliminar hace indefinida la función D para un punto del dominio TipoClave.

En base a la semántica intuitiva de los operadores, dada arriba, diremos que el TAD Diccionario puede caracterizarse como una extensión del TAD Funcion[2] con el operador de restricción del dominio de la función en un punto; se deja al lector como ejercicio la extensión de este TAD.

7.2 Especificación del TAD Diccionario

La especificación del TAD Diccionario es la siguiente:
TDA Diccionario = $Funcion[TipoClave, TipoElemento]$
Sintaxis

$$\begin{array}{llcl}
vacio : & & \Rightarrow & Diccionario \\
esvacio : & Diccionario & \Rightarrow & Boolean \\
insertar : & Diccionario \times TipoClave & & \\
 & \times TipoElemento & \Rightarrow & Diccionario \\
eliminar : & Diccionario \times TipoClave & \Rightarrow & Diccionario \\
recuperar : & Diccionario \times TipoClave & \Rightarrow & TipoElemento
\end{array}$$

Semántica
$\forall d \in Diccionario, e, e_1 \in Elemento, e \neq_{Elemento} e_1,$

[2] Ver la especificación del TAD $Funcion[A, B]$ en la sección 3.5 del capítulo 3

1. $vacio = new$
2. $esvacio(vacio) = true$
3. $esvacio(insertar(d;k;e)) = false$
4. $insertar(d;k;e) = def(d;k;e)$
5. $recuperar(d;k) = aplic(d;k)$
6. $eliminar(d;k) = restringir(d;k)$

Fin-Diccionario;[3]

7.3 Representación Mediante Tablas de *Hash*

En la sección anterior, cuando hablamos de las operaciones que caracterizan a los diccionarios, dimos parámetros de frecuencia de las mismas que nos conviene analizar para determinar si una representación es adecuada o no:

- insertar y eliminar son operaciones frecuentes.
- recuperar es **muy** frecuente.

En base a la frecuencia de uso de recuperar, se requiere que esta operación no se realice por búsqueda secuencial, ataríamos al tiempo de acceso a ser proporcional al tamaño del diccionario. En el caso de la búsqueda binaria, se requiere que los elementos estén ordenados por la clave. En particular, la base de datos de estudiantes de la USB cuenta con más de 10000 estudiantes.

A continuación presentamos un esquema de representación del TAD Diccionario conocido como representación mediante tablas de hash[4]. La idea básica es que el conjunto de elementos se particiona en clases de equivalencia (disjuntas) mediante la aplicación de una función, denominada función de hash. Por ejemplo, en el caso de C, es posible particionarlo en clases: la clase de las palabras que empiezan con la letra A, la clase de las que empiezan con

[3] El operador restringir se refiere al ejercicio propuesto en la sección de ejercicios del capítulo 3

[4] Una traducción de hash (que no usaremos) sería picadillo

la letra B y así hasta la letra Z, generando 27 clases. Esto se logra aplicando la función H:

$$
\begin{aligned}
H(\text{amarillo}) &= A\\
H(\text{borrar}) &= B\\
&\vdots
\end{aligned}
$$

y así sucesivamente.

En general, supongamos que podemos particionar los elementos de un diccionario mediante la relación de equivalencia "los elementos con igual imagen bajo la función de hash". Esta relación deberá crear un número de clases de equivalencia menor que el universo potencial; estas clases las numeraremos de 0 a M–1.

Para representar un diccionario mediante tablas de hash, necesitamos definir lo siguiente:

- La función de hash (H).

  Hasta ahora sólo sabemos que $\forall e \in \mathit{Elemento};\ 0 \leq H(clave(e)) \leq M-1$

- Representar las clases. Para representar las clases podemos distinguir dos enfoques:
  - Hashing Abierto.
  - Hashing Cerrado.

A continuación discutiremos cada uno de los enfoques propuestos.

7.3.1 Hashing Abierto

En el esquema de hashing abierto, puede distinguirse claramente el conjunto cociente y las clases; el conjunto cociente se representa mediante un arreglo de M componentes (tantas como clases de equivalencia defina la función de hash), donde cada elemento del arreglo se asocia a una clase, y a su vez, cada clase que puede tener varios miembros, se representa mediante cualquier estructura que sea capaz de almacenar un conjunto.

```
Type RgoClases = 0..M-1;
     Clase = Conjunto[Elemento];
     TablaDeHash = array[RgoClases]of Clase;
     Diccionario = TablaDeHash;
```

	0	1	2	···	C
0				···	
1				···	
⋮				···	
M-1				···	

Tabla 7.1: Almacenamiento

Una posible implementación es un vector cuyos elementos sean vectores, es decir, las clases se implementan como vectores. Gráficamente la estructura es la que se muestra en la tabla 7.1.

Los elementos se asignan en el primer lugar de cada vector, y si está ocupado, se recorre el vector horizontalmente buscando una posición vacía.

Otra implementación posible es la que utiliza representación dinámica (listas). Es decir, la tabla de hash sería un vector de listas donde cada una de ellas representa una clase.

En este último caso, el mecanismo de búsqueda de una clave e, sería el cálculo de H(e), lo que daría acceso a la clase en la cual debería encontrarse el elemento, y luego se realizaría un recorrido de la lista hasta encontrar el elemento buscado. Debe notarse que en el caso en que la clave buscada no se encuentre en el diccionario, el costo de la búsqueda será proporcional al tamaño de la clase de equivalencia donde hubiera estado la clave buscada.

Si las clases son pequeñas con respecto al tamaño total del diccionario, podemos decir que el tiempo de recuperación es casi constante.

Para obtener un buen comportamiento de las operaciones con esta representación es necesario elegir H de forma tal que cada clase de equivalencia tengan más o menos el mismo tamaño, es decir que los elementos del diccionario estén uniformemente distribuídos entre las clases. Otra cosa deseable es que las clases no sean demasiado grandes. Si las condiciones anteriores se dan y el diccionario tiene N elementos, cada clase tendrá en promedio N/M elementos y este es el tamaño del conjunto donde se realizará la búsqueda.

Este modelo de organización de datos requiere la utilización de listas y por lo tanto, manejo dinámico de memoria, por lo que hay un incremento en el uso de memoria. Es bueno recordar que el almacenamiento en vectores es lo más económico como forma de guardar información ya que no hace uso de memoria adicional al requerido para guardar los datos.

7.3.2 Hashing Cerrado

En este modelo, se almacenan todas las entradas del diccionario en un vector, usando la función de hashing para determinar la posición en el vector donde una entrada será colocada. Debido a que las clases de equivalencia de claves definidas por la función no son en general de tamaño unitario, se producirán colisiones, lo que corresponde a tener varias claves que pretenden ocupar la misma posición.

Bajo este enfoque, no hay una diferenciación entre el conjunto cociente y las clases ya que los elementos del diccionario se almacenan dentro del arreglo. Esta representación presenta el problema de las colisiones ya que cuando dos elementos tienen la misma imagen bajo la función de hash, sólo uno de los dos puede ser almacenado en esa posición del arreglo (el primero que encontró la posición vacia). Para resolver este problema, se define una secuencia de funciones de rehash, mediante las cuales se examinan otras posiciones dentro del arreglo hasta encontrar una que esté vacia, donde finalmente se coloca el elemento. Si no se encuentra ninguna posición vacia no será posible hacer la inserción.

```
Type RgoEntradas = 0..K-1; (* K > M *)
     TablaDeHash = array[RgoEntradas]of Elemento;
     Diccionario = TablaDeHash;
```

A continuación presentamos algunos métodos para resolver colisiones mediante rehashing:

1. **Prueba Lineal**
   Este método de solución de colisiones consiste en tener una función de hash (H_0) y si hay colisión, intentar en posiciones contiguas considerando el arreglo como un anillo (la posición contigua a $M-1$ es 0).

$$H_0(e) = H(e)$$

$$H_i(e) = (H_0(e) + i) \bmod M;\ i > 0 \tag{7.1}$$

0	1	2	3	4	5	6	7	8	9
		11			50	23	15	24	14
		y_2			y_1	y_3	y_4	y_5	y_6

Tabla 7.2: Almacenamiento de valores

Esta función presenta el problema de que no distribuye los elementos uniformemente en la tabla y puede ocurrir que parte del arreglo esté vacía y otra muy llena; sin embargo, el tamaño total del diccionario está limitado por el tamaño del vector. Consideremos el siguiente ejemplo: sea D un diccionario de a lo sumo 10 elementos, representando mediante una tabla de hashing cerrado, cuyas claves tienen un máximo de dos dígitos decimales, denominados d_1 y d_0. La función de hash correspondiente es

$$H_0(10d_1 + d_0) = (d_1 + d_0) \bmod 10;$$

y H_i es como (7.1). Si los elementos se ingresan en el siguiente orden

$$(50; y_1); (11; y_2); (23; y_3); (15; y_4); (24; y_5); (14; y_6)$$

donde y_i corresponde a la información asociada a la clave, el vector resultante es el que se muestra en la tabla 7.2.

Es de hacer notar que si bien la clase de equivalencia de

$$H_0(14) = H_0(23) = H_0(50) = 5;$$

es de tamaño 3, la recuperación de la clave 14, requiere 5 comparaciones de claves.

2. **Prueba Cuadrática**

   $H_0(e) = H(e)$

   $H_i(e) = (H_0 + i^2)$ **mod** M, $i > 0$

   el vector resultante es el que se muestra en la tabla 7.3.

0	1	2	3	4	5	6	7	8	9
24		11			50	23	15		14
y_5		y_2			y_1	y_3	y_4		y_6

Tabla 7.3: Resultado

No hay demasiada diferencia entre las tablas usando el hashing lineal o cuadrático, sin embargo puede notarse, aún en un ejemplo tan pequeño que en el hashing lineal las clases quedan pegadas entre si mientras que en el hashing cuadrático hay mayor dispersión de claves. Esto hace que las búsquedas en cada clase se hagan mas cortas.

Una desventaja de esta función de rehash es que no todas las posiciones son revisadas para insertar, por lo tanto podría darse el caso de no poder insertar un elemento aunque hubiera espacio disponible. Una característica deseable para este tipo de hashing es que el porcentaje de llenado de la tabla no supere el 50% del total de la misma.

7.4 Resúmen del capítulo 7

Hemos visto un método de organizar un conjunto de elementos que poseen la característica que el conjunto tiene escaso crecimiento, escaso respecto a la frecuencia de utilización del operador de recuperación o búsqueda.

El esquema presentado en este capítulo logra un tiempo constante (independiente del tamaño del conjunto) para la operción de búsqueda. Esto es particularmente promisorio para conjuntos muy grandes. No debemos dejarnos llevar por el entusiasmo y aceptar que este método es el mejor ya que presenta la mas baja complejidad respecto a los otros métodos. Esta organización requiere características de la claves que nos permitan establecer la funciones de hashing (y eventualmente de rehashing) que logren alcanzar la complejidad anunciada para la operación de búsqueda. Otro inconveniente a tener en consideración al momento de elegir este método de representación es el manejo del crecimiento del conjunto. En particular los conjuntos organizados con este método en una aplicación tienen la particularidad que el comportamiento del operador de búsqueda puede degradarse con la utilización de la aplicación, principalmente debido a ingresos. Es por eso que aplicaciones que incluyan datos organizados por hashing deben ser monitoreados (administradas) en su comportamiento, veri¯cando que las hip¶otesis sobre los datos se mantienen a pesar de los ingresos, que la bondad de las funciones sigue siendo v¶alida y eventualmente realizar operaciones de mantenimiento que pueden pasar por la reorganizaci¶on de los datos o el cambio de las funciones de hashing (las funciones de rehashing).

El el cap¶ıtulo 8 se revisan organizaciones de conjunto que son menos sensibles al crecimiento (decrecimiento) del conjunto sobre el cual se realiza la b¶usqueda.

7.5 Ejercicios

1. Calcular el n¶umero de palabras posibles de no m¶as de 14 letras.

2. Veri¯car el n¶umero de palabras que est¶an de¯nidas en su diccionario favorito.

3. Considere la siguiente representaci¶on para el TAD Diccionario:

   ```
   Type Diccionario = array[0..10] of Elemento;
   ```

   con la siguiente funci¶on de hash: $H_0(e) = e^2 \bmod 11$

   (a) Construya la tabla de hash abierta para los siguientes elementos:

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

   (b) Con los mismos elementos de (a), construya la tabla de hash cerrada, considerando para el rehashing el m¶etodo de prueba lineal.

   (c) C¶omo se alterar¶ia (b) si se utiliza para el rehashing el m¶etodo de prueba cuadr¶atica.

4. Hasta ahora se ha especi¯cado el TAD Diccionario como una especializaci¶on del TAD Funcion[A; B]. >Cu¶al es la diferencia si se especi¯ca como una especializaci¶on del TAD Conjunto[Elemento]?.

5. Supongamos que se mantiene el diccionario de estudiantes de la USB, representado mediante hashing abierto, donde las claves vienen dadas por los carnets. Suponga adem¶as que 1=3 de los estudiantes est¶an cursando el ciclo b¶asico.

(a) Defina una función de hash que agrupe a todos los estudiantes del ciclo básico en la misma clase de equivalencia.

(b) Defina una función de hash que distribuya la información de los estudiantes "aleatoriamente" en la tabla de hash que representa el diccionario.

(c) Compare las funciones definidas en (a) y (b).

(d) Implemente las funciones definidas en (a) y (b) utilizando hashing abierto y hashing cerrado con prueba lineal.

6. Considere los siguientes métodos de hashing:

(i) Método de la División (H_d):
H_d se obtiene tomando el resto de dividir la suma de los dígitos de la clave por M. Es decir, $H_d(clave) \in \{0, \ldots, M-1\}$. Es decir: Si clave $= d_n 10^n + \ldots + d_1 10 + d_0$ entonces

$$H_d(clave) = \left(\sum_{i=0}^{n} d_i\right) \bmod M$$

(ii) Midsquare Hashing (H_m):

(ii.1) Elevar al cuadrado parte de la clave, o la clave completa si es posible.

(ii.2a) Extraer n dígitos decimales a partir de la mitad del resultado de (ii.1) para obtener $H_m(clave) \in \{0, 1, \ldots, 10^n - 1\}$.

(ii.2b) Extraer n bits a partir de la mitad del resultado de (ii.1) para obtener$H_m(clave) \in \{0, 1, \ldots, 2^n - 1\}$.

(iii) Folding (H_f):
Se divide la clave en varias partes que luego se suman para obtener H_f en el rango deseado. Por ejemplo si deseamos H_f de dos dígitos y la clave tiene siete, hacemos lo siguiente:

$$H_f(1761439) = 17 + 61 + 43 + 9 = 30$$

Nótese que el dígito de 10^n, donde n es el número de dígitos deseados, se ignora.

(a) Implemente los métodos de hashing descritos arriba.

(b) Compare el resultado de aplicar éstos métodos a un conjunto de claves dado. Cuál método distribuye mejor?

7. El Sistema nacional de Bibliotecas (SNB), ha decidido ofrecer a sus usuarios un nuevo servicio llamado **Préstamos a Distancia**, que consiste en poner a la disposición no sólo los títulos que se encuentran en la Biblioteca Central, sino también los que se encuentran en cada una de las bibliotecas adscritas al sistema. Ya han resuelto los problemas de envío y transporte de libros, pero se han dado cuenta que deben organizar adecuadamente la información. Lo primero que han hecho es codificar las diferentes bibliotecas del país, adscritas o no, así como los títulos para identificarlos univocamente. Con esto podrán ofrecer un catálogos de bibliotecas y de títulos, donde se pueden consultar dichos códigos.

   El paso siguiente es la automatización de los procesos de mantenimiento y consulta de la información, para lo cual el SBN ha solicitado su colaboración. La idea es que Ud. les dé una solución desde el punto de vista computacional para:

   (a) Mantener la información: ingreso y desincorporación de títulos y ejemplares.

   (b) Consultar si una biblioteca determinada está adscrita o no al SBN.

   (c) Consultar si un título está disponible o no, para hacer la solicitud de préstamo.

   Ud. debe considerar la siguiente política:

   Es preferible aumentar la cantidad de títulos en cada biblioteca y restringir el número de adscripciones al SBN (presuponiendo que es muy costoso expandir la red de transporte).

   - Una biblioteca sólo puede prestar un libro a distancia si tiene más de un ejemplar del mismo.
   - Hay M bibliotecas adscritas al SBN.
   - Cada biblioteca tiene a lo sumo N títulos diferentes.
   - $M << N$.
   - Es **muy** frecuente la incorporación de nuevos títulos en una biblioteca, aunque en menor que $1/3$ del número de consultas.
   - La incorporación de bibliotecas al SBN es bajo.

(a) Analice la pol¶tica dada.

(b) Adoptando la pol¶tica, suponga que adem¶as de consultar las bibliotecas y los libros por sus c¶odigos, se desea hacer las consultas por nombres y autores, respectivamente. Proponga una soluci¶on para este problema utilizando tablas de hash. Justi¯que con par¶ametros num¶ericos y la relaci¶on entre ellos su soluci¶on.

7.6 Bibliograf¶ia

1. AHO, Alfred, HOPCROFT, John & Ullman,Je®rey. \ Data Structures and Algorithms". Addison Wesley Series in Computer Science.1985

2. BERZTISS, A.T.\Data Structures. Theory and Practice".Academic Press.

3. CAIRO, O. & GUARDATI, S. \Estructuras de Datos". McGraw-Hill. 1.993.

4. GONNET,G.H. & BAEZA-YATES, R. \Handbook of Algorithms and Data Structures. In PASCAL and C. 2^{nd} Ed. Addison Wesley.

5. WIRTH, Nicklaus. \Algorithms+Data Structures= Programs".Prentice Hall.

Capítulo 8

Árboles

8.1 Marco Conceptual

Algunas veces, cuando modelamos una situación encontramos que la mejor manera de representar la relación que existe entre los objetos involucrados es mediante una estructura jerárquica, como vemos en la figura 8.1. Tal es el caso de la relación entre las diferentes unidades administrativas, que describe una organización tradicional, llamada organigrama. Otro ejemplo lo podemos ver en el árbol genealógico de una persona, que vemos en la figura 8.2 donde A es hijo de B y C, teniendo como abuelos a D,E,F y G . Asimismo puede servir para organizar los elementos de un conjunto de manera de poder realizar operaciones sobre el mismo de manera más eficiente.

Este tipo de estructuras jerárquicas recibe el nombre de árbol. En el caso de un organigrama observamos que en cada nivel del mismo se encuentran, por lo general, más de dos unidades administrativas, por lo que se conoce como árbol M-ario. El árbol genealógico, en que cada individuo esta representado por un elemento del árbol, relacionandose con su padre y madre es un árbol binario (2-ario).

En los ejemplos mencionados arriba notamos que es irrelevante el orden en el que ocurren los objetos de cada nivel, por lo que se conocen como árboles no ordenados.[1]

Formalmente un árbol M-ario no ordenado podemos modelarlo como una

[1] Podemos considerar a un árbol como un conjunto de elementos y una relación particular sobre el producto cartesiano.

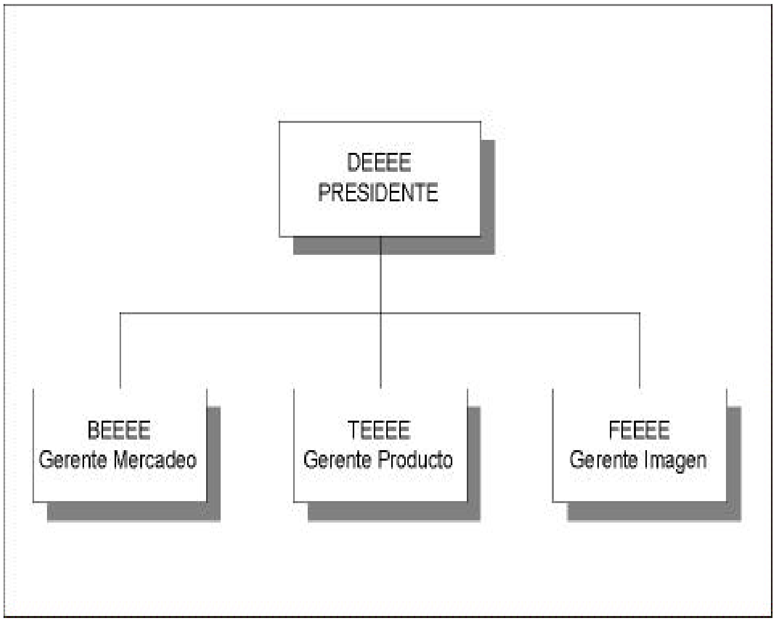


Figura 8.1: Organigrama

secuencia heterogénea [2] de dos elementos. A continuación daremos una definición recursiva.

Definición: Árbol M-ario no ordenado:

(i) La secuencia vacía ($<>$) corresponde al árbol nulo no ordenado.

(ii) Sea e un elemento del dominio y $a_1, \ldots, a_M$ árboles M-arios no ordenados, entonces, la secuencia $a = < e, [a_1, \ldots, a_M] >$, donde $[a_1, \ldots, a_M]$ denota un multiconjunto de árboles, es un árbol N-ario no ordenado y se definen las siguientes relaciones:

- e es la raíz de a.
- $\forall\, 1 \leq i \leq M$, a_i es un subárbol de a.

[2] Heterogéneas en el sentido de los elementos pueden ser de tipos diferentes.

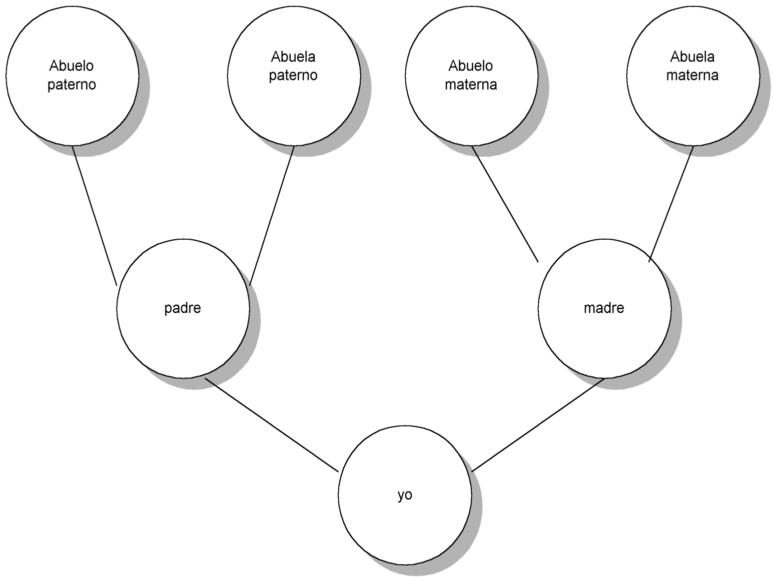


Figura 8.2: Árbol genealógico

- $\forall\, 1 \le i \le M$ e es padre de la raíz de a_i.
- $\forall\, 1 \le i \le M$ la raíz de a_i es hijo (sucesor) de e.

(iii) Sólo son árboles no ordenados los objetos generados a partir de (i) y de (ii).

En la definición anterior notamos que los sucesores están organizados como un multiconjunto. Sin embargo, si consideramos el árbol asociado a una expresión algebraica en la que cada nodo con hijos es el padre de sus operandos (sean árboles o variables simples), nos damos cuenta de que en ese caso es importante el orden en que se colocan los argumentos ya que para los operadores no conmutativos es importante distinguir cual es el primer operando y cual es el segundo. Otro ejemplo de esto es el árbol asociado a una instrucción condicional $\mathtt{if}(b; s_1; s_2)$. Si consideramos s_1 y s_2 como expresiones algebraicas y b como una expresión booleana, el operador $\mathtt{if}$ puede considerarse como un operador ternario, siendo s_1 el resultado en el caso en que la expresión b resulte verdadera y s_2 en el caso contrario. Este tipo de árboles se denominan árboles ordenados. Para los árboles ordenados es conveniente organizar a los sucesores como una secuencia, debido a que en ellas es importante la posición en que los elementos ocurren. En el caso del if sería: $< if, < b, s_1, s_2 >>$, con lo cual tenemos la siguiente definición:

Definición: Árbol M-ario ordenado:

(i) La secuencia vacía ($<>$) corresponde al árbol ordenado nulo.

(ii) Sea e un elemento del dominio y $a_1, \ldots, a_M$ árboles M-arios ordenados, entonces, la secuencia $a = < e, < a_1, \ldots, a_M >>$ es un árbol M-ario ordenado y se definen las mismas relaciones de la definición anterior.

(iii) Sólo son árboles ordenados los objetos generados a partir de (i) y de (ii).

En este capítulo sólo nos ocuparemos de los árboles ordenados, por lo cual utilizaremos el modelo donde los sucesores están representados mediante una secuencia.

8.2 Especificación del TAD Arbol (Árboles Ordenados)

Para definir el TAD Arbol requerimos además del TAD Secuencia visto en el capítulo 4, el TAD SecHet cuyos valores son secuencias heterogéneas. ESP_{SecHet} es igual a $ESP_{Secuencia}$ modificando el conjunto de dominios D a la unión de todos los dominios que intervienen. En el caso de los árboles los dominios de ESP_{SecHet} serán secuencias formadas por elementos y secuencias de árboles.
En el resto de este capítulo utilizaremos op_H para denotar los operadores del TAD SecHet.

Como los árboles son una especialización de las secuencias heterogeneas, usaremos para definir los operadores, los operadores de las secuencias heterogeneas.

8.2.1 Árboles Ordenados

TAD $Arbol < SecHet[Elemento, Secuencia[Arbol]]$

$$D = \{Arbol, SecHet, Secuencia, Boolean, Elemento\}$$

Sintaxis

nulo :		$\rightarrow$	Arbol
esnulo :	Arbol	$\rightarrow$	Boolean
creararbol :	Elemento $\times$ Secuencia	$\rightarrow$	Arbol
raiz :	Arbol	$\rightarrow$	Elemento
succ :	Arbol	$\rightarrow$	Secuencia
#elem :	Arbol	$\rightarrow$	Natural
eshoja :	Arbol	$\rightarrow$	Boolean
altura :	Arbol	$\rightarrow$	Natural

Semántica

$\forall a, a_1 \in \text{Arbol}; s : \text{Secuencia}; e \in \text{Elemento}; a_1 \neq \text{nulo};$

1. $long_H(a) = 0 \vee long_H(a) = 2$
2. $nulo = vacia_H$
3. $esnulo(a) = esvacia_H(a)$
4. $creararbol(e; s) = insertar_H(insertar_H(vacia_H; 1; e); 2; s)$
5. $\neg(esnulo(a_1)) \rightarrow raiz(a_1) = proyectar_H(a_1; 1)$
6. $\neg(esnulo(a_1)) \rightarrow succ(a_1) = proyectar_H(a_1; 2)$
7. $eshoja(nulo) = false$
8. $\neg esnulo(a) \rightarrow eshoja(a) = false$
9. $long(proyectar_H(a; 2)) = 0 \rightarrow eshoja(a) = true$
10. $eshoja(a) = \bigwedge_{i=1}^{long(succ(a))} proyectar(succ(a); i) = nulo$
11. $\#elem(nulo) = 0$
12. $\#elem(a_1) = 1 + \sum_{i=1}^{long(succ(a_1))} \#elem(proyectar(succ(a_1); i))$
13. $eshoja(a_1) \rightarrow altura(a_1) = 0$
14. $\neg(eshoja(a_1)) \rightarrow$
    $altura(a_1) = 1 + max_{(i=1;long(succ(a_1)))}(altura(proyectar(succ(a_1); i)))$

Fin-Arbol;

El árbol genealógico de la figura 8.2 es un árbol de altura 2.

Puede notarse que estamos usando en la especificación la propiedad que los operadores sum y max no serán evaluados cuando el límite superior sea inferior al límite inferior, dando como resultado de la evaluación cero (el neutro de la suma).

8.2.2 Conceptos de árboles

Algunos conceptos importantes relacionados con los árboles son los siguientes:

Grado : El grado de un árbol se refiere al número de hijos que tiene el elemento con más sucesores. Cuando hablamos de árboles M-arios, estamos suponiendo que el elemento con más sucesores tiene M hijos, por lo tanto el grado es M.

Nivel : Un nivel de un árbol se refiere a un estrato de la estructura jerárquica. Dependiendo de la convención que se tome, el nivel donde se encuentra la raíz, es el nivel 0 o el nivel 1. En este libro usaremos como nivel de la raíz cero.

Árbol Completo : Un árbol es completo si el número de hijos de cada elemento es igual al grado del árbol. Un árbol como el que presentamos en la figura 8.3 tiene en el iésimo nivel M^i elementos, por lo tanto el número de elementos del árbol es igual a

$$N = \sum_{i=0}^{k} M^i = (M^{k+1} - 1)/(M - 1)$$

donde k corresponde a la altura del árbol.

8.3 Recorrido en Árboles

Consideremos nuevamente el ejemplo del organigrama y supongamos que queremos obtener todas las unidades administrativas que constituyen la organización que representa. Para obtenerlas, debemos revisar el árbol de alguna

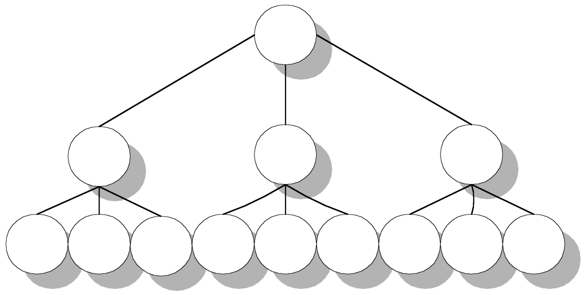


Figura 8.3: Árbol de altura 2

manera y listar cada una de las raíces de los sub-árboles. Esta operación se conoce como recorrido de un árbol. Existen diversas maneras de realizar el recorrido, pero en general, un recorrido es una función que va de árboles en listas de elementos donde cualquier lista de los elementos que ocurren en el árbol es considerada un recorrido[3]:

$$recorrido : Arbol \Rightarrow lista[Elemento]$$

Para especificar la semántica de esta operación, debemos extender el TAD Arbol con una operación que permita determinar si un elemento ocurre o no en un árbol:

$$ocurre : Arbol \times Elemento \Rightarrow Boolean$$

Con la siguiente semántica:

- $ocurre(nulo; e) = false$
- $ocurre(a; e) = (e = raiz(a)) \vee \bigvee_{i=1}^{long(succ(a))} ocurre(proyectar(succ(a); i); e)$

Una vez extendido el TAD Arbol con el operador ocurre podemos establecer la semántica de recorrido[4]:

$$\forall a (\forall e\, (ocurre(a; e) \rightarrow \exists i\, (scan(recorrido(a); i)) = e))$$
$$\wedge len(recorrido(a)) \geq \#elem(a))$$

[3]Ver la especificación del TAD Lista en el capítulo 3.

[4]scan(l,i) es un operador que extiende el TAD Lista y que devuelve el elemento que ocupa la i-ésima posición en la lista l.

Nótese que con esta semántica de la operación recorrido estamos pidiendo que todos los elementos de árbol están en la lista resultante y además estamos admitiendo como un recorrido de un árbol cualquier lista de los elementos que ocurran en él, aún cuando la longitud de esta sea mayor que su número de elementos, lo cual significa que se están repitiendo algunos o cada uno de ellos.

Como dijimos anteriormente, hay varias maneras de realizar el recorrido. En este capítulo nos interesaremos en tres métodos específicos que retornan una lista con los elementos del árbol (sin repetición). Los métodos a los cuales nos referimos son conocidos como preorden, postorden e inorden.

Debido a que cuando se realiza un recorrido se "visitan" todos los elementos ya que tenemos el operador raiz para "visitar" a la raíz de un árbol pero no tenemos un operador del TAD Arbol para acceder a cada uno de las raices de los sucesores. Es por eso que definiremos una función para realizar la operación en cada uno de las formas de recorrido que serán revisados, que retorna una lista con los elementos (sin repetición) de cada uno de los subárboles. Estos operadores son VisitarPre,VisitarPost y VisitarIn, cuya firma damos a continuación

$$\begin{aligned} &VisitarPre, VisitarPost, \\ &\qquad VisitarIn : \text{Secuencia[Arbol]} \Rightarrow \text{Lista[Elemento]} \end{aligned}$$

La semántica de cada uno de estos operadores se dará con el método.

8.3.1 Recorrido en Preorden

Recorrido en Preorden Consiste en listar primero la raíz del árbol y luego los elementos en los subárboles sucesores en el orden en el cual ocurren.

- $esvacia(succ(a)) \rightarrow$
  $preorden(a) = cons(raiz(a), null)$

- $\neg esvacia(succ(a)) \rightarrow$
  $preorden(a) = cons(raiz(a), VisitarPre(succ(a)))$

- $esvacia(s) \rightarrow VisitarPre(s) = null$

- $\neg esvacia(s) \rightarrow$
  $VisitarPre(s) = append(preorden(proyectar(s; 1)); VisitarPre(eliminar(s; 1)))$

8.3.2 Recorrido en Postorden

Recorrido en Postorden Consiste en listar primero los elementos en los subárboles sucesores en el orden en el cual ocurren y luego la raíz.

- $esvacia(succ(a)) \rightarrow$
  $postorden(a) = cons(raiz(a); null)$

- $\neg esvacia(succ(a)) \rightarrow$
  $postorden(a) = append(VisitarPost(succ(a)); cons(raiz(a); null))$

- $esvacia(s) \rightarrow VisitarPost(s) = null$

- $\neg esvacia(s) \rightarrow$
  $VisitarPost(s) = append(postorden(proyectar(s; 1)); VisitarPost(eliminar(s; 1)))$

8.3.3 Recorrido en Inorden

Recorrido en Inorden Consiste en listar primero los elementos del subárbol sucesor que se encuentran al principio de la secuencia de sucesores, luego la raíz y finalmente, los elementos en el resto de los subárboles sucesores.

- $esvacia(succ(a)) \rightarrow$
  $inorden(a) = cons(raiz(a); null)$

- $\neg esvacia(succ(a)) \rightarrow$
  $inorden(a) = append(inorden(proyectar(succ(a; 1))); cons(raiz(a); VisitarIn(eliminar(succ(a); 1))))$

- $esvacia(s) \rightarrow VisitarIn(s) = null$

- $\neg esvacia(s) \rightarrow$
  $VisitarIn(s) = append(inorden(proyectar(s; 1)); VisitarIn(eliminar(s; 1)))$

En la siguiente sección de este capítulo, presentaremos los árboles binarios y algunas especializaciones de los mismos; estas especializaciones se harán definiendo cada operador en términos del TAD más general, y añadiendo los axiomas que definen el comportamiento particular, que denominaremos Axiomas de Comportamiento.

8.4 Árboles Binarios

Un árbol binario es un árbol ordenado constituido por una raíz y dos subárboles binarios, por lo tanto, los modelaremos como secuencias heterogéneas de la forma

$$< e, < a_1, a_2 >>$$

donde $e \in$ Elemento y $< a_1, a_2 >$ es una secuencia de dos árboles binarios, donde la primera proyección la llamaremos izq y la segunda der. Nótese que por ser los árboles binarios ordenados, la secuencia de sucesores debe ser vacia o debe tener los dos subárboles. Un ejemplo típico de árbol binario es el árbol asociado a una expresión booleana, donde la máxima aridad de los operadores es igual a dos.

TDA ArbolBinario < Arbol[Elemento]

$$D = \{ArbolBinario, Boolean, Elemento\}$$

Sintaxis

nulo :		$\Rightarrow$	ArbolBinario
esnulo :	ArbolBinario	$\Rightarrow$	Boolean
crearbin :	Elemento $\times$ Secuencia	$\Rightarrow$	ArbolBinario
raiz :	ArbolBinario	$\Rightarrow$	Elemento
izq, der :	ArbolBinario	$\Rightarrow$	ArbolBinario

Semántica
$e \in$ Elemento, $s \in$ Secuencia[ArbolBinario];

$$\text{esvacia}(s) \vee \text{long}(s) = 2$$

$$\text{crearbin}(e, s) = \text{creararbol}(e, s)$$

$$izq(creararbol(e; s)) = proyectar(s; 1)$$

$$der(creararbol(e; s)) = proyectar(s; 2)$$

$$\vdots$$

Fin-ArbolBinario;

Los puntos suspensivos se refieren a los operadores #elem, eshoja y altura y los axiomas correspondientes se dejan como ejercicio al lector.

Convención: Debido a que la secuencia de sucesores tiene a lo sumo longitud 2, simplificaremos la notación de secuencia para los sucesores. En la definición de un árbol binario se debe indicar entonces los dos subárboles aunque sean nulos y la raíz. Nos queda entonces la siguiente especificación:

TDA ArbolBinario
Sintaxis

$nulo:$		$\Rightarrow$	$ArbolBinario$
$esnulo:$	$ArbolBinario$	$\Rightarrow$	$Boolean$
$crearbin:$	$Elemento \times ArbolBinario \times ArbolBinario$	$\Rightarrow$	$ArbolBinario$
$raiz:$	$ArbolBinario$	$\Rightarrow$	$Elemento$
$izq; der:$	$ArbolBinario$	$\Rightarrow$	$ArbolBinario$

Semántica
$\forall b_1; b_2 \in ArbolBinario; e \in Elemento; s \in Secuencia[ArbolBinario];$

$$raiz(crearbin(e; b_1; b_2)) = e$$

$$izq(crearbin(e; b_1; b_2)) = b_1$$

$$der(crearbin(e; b_1; b_2)) = b_2$$

$$\vdots$$

Fin-ArbolBinario;

Implementaci¶n del TAD ArbolBinario[Elemento]**:** Al pensar en implementaciones de ¶rboles, en particular binarios, debemos pensar en estructurar cada elemento del conjunto como una terna. En la primera componente colocaremos el elemento del conjunto y en las dos siguientes las referencias a los ¶rboles correspondientes. Estas referencias pueden ser posiciones de memoria (apuntadores) (al estilo PASCAL) o posiciones dentro de un vector que almacena los elementos (enteros) o ¶rboles (al estilo JAVA).

A continuaci¶n presentamos una implementaci¶n del TAD ArbolBinario haciendo uso de memoria din¶mica.

```
type ArbolBinario = ^NodoArbol;
     NodoArbol = record
                    elem: Elemento;
                    izq, der: ArbolBinario
                 end;
```

En el ap¶ndice A se encuetran las implementaciones de los operadores del TDA ArbolBinario

8.4.1 ¶Arboles Binarios de B¶squeda

Un ¶rbol binario de b¶squeda, tambi¶n conocido como ¶rbol de b¶squeda, es aquel donde la ra¶ces del sub¶rbol izquierdo y derecho son menor y mayor, respectivamente que la ra¶z del ¶rbol, seg¶n la relaci¶n Á, y ambos sub¶rboles son ¶rboles de b¶squeda. Dado que estos objetos deben mantener su organizaci¶n en relaci¶n al orden, surge la necesidad de dos nuevos operadores, insertarbol y eliminarbol, para permitir el ingreso y el egreso de elementos en el ¶rbol. N¶tese que si mantuvieramos el operador crearbin correr¶amos el riesgo de perder la organizaci¶n de ¶rbol de b¶squeda. Como su nombre lo indica, los ¶rboles de b¶squeda son objetos que permiten mantener y recuperar informaci¶n. Los operadores mencionados anteriormente est¶n orientados al mantenimiento. Para recuperar se dispone del operador buscarbol. Recordemos que un ¶rbol de b¶squeda es una forma de organizar un conjunto en que los elementos admiten una relaci¶n de orden total. De la misma manera se puede utilizar los ¶rboles para almacenar multiconjuntos, admitiendo la presencia de claves repetidas.

TAD ArbolBusqueda < ArbolBinario[Elemento]

Sintaxis

nulo :		$\Rightarrow$	ArbolBusqueda
esnulo :	ArbolBusqueda	$\Rightarrow$	Boolean
insertarbol :	ArbolBusqueda $\times$ Elemento	$\Rightarrow$	ArbolBusqueda
eliminarbol :	ArbolBusqueda $\times$ Elemento	$\Rightarrow$	ArbolBusqueda
buscarbol :	ArbolBusqueda $\times$ Elemento	$\Rightarrow$	ArbolBusqueda
raiz :	ArbolBusqueda	$\Rightarrow$	Elemento
izq; der :	ArbolBusqueda	$\Rightarrow$	ArbolBusqueda
numnodos :	ArbolBusqueda	$\Rightarrow$	Natural
eshoja :	ArbolBusqueda	$\Rightarrow$	Boolean
altura :	ArbolBusqueda	$\Rightarrow$	Natural
max :	ArbolBusqueda	$\Rightarrow$	Elemento

Semántica

$\forall b \in$ ArbolBusqueda; $b \neq$ nulo;

- **Axioma de Comportamiento**

  a. $\neg$esnulo(izq(b)) $\wedge$ $\neg$esnulo(der(b)) $\rightarrow$
     raiz(izq(b)) $\prec$ raiz(b) $\preceq$ raiz(der(b))

  b. $\neg$esnulo(izq(b) $\wedge$ esnulo(der(b)) $\rightarrow$
     raiz(izq(b)) $\prec$ raiz(b))

  c. esnulo(izq(b)) $\wedge$ $\neg$esnulo(der(b)) $\rightarrow$
     raiz(b) $\preceq$ raiz(der(b))

- insertarbol(nulo; e) = crearbin(e; nulo; nulo)

- e $\prec$ raiz(b) $\rightarrow$
  insertarbol(b; e) = crearbin(raiz(b); insertarbol(izq(b); e); der(b))

- raiz(b) $\preceq$ e $\rightarrow$
  insertarbol(b; e) = crearbin(raiz(b); izq(b); insertarbol(der(b); e))

- buscarbol(nulo; e) = nulo

- raiz(b) = e $\rightarrow$ buscar(b; e) = b

- raiz(b) $\prec$ e $\rightarrow$ buscarbol(b; e) = buscarbol(der(b); e)

² e Á raiz(b) ! buscarbol(b; e) = buscarbol(izq(b); e)

² eliminarbol(nulo; e) = nulo

² raiz(b) Á e ^ : esnulo(der(b)) !
eliminarbol(b; e) = crearbin(raiz(b); izq(b)); eliminarbol(der(b); e)))

² raiz(b) Á e ^ esnulo(der(b)) !

eliminarbol(b; e) = b

² e Á raiz(b) ^ : esnulo(izq(b)) !
eliminarbol(b; e) = crearbin(raiz(b); eliminarbol(izq(b); e); der(b)))

² e Á raiz(b) ^ esnulo(izq(b)) !

eliminarbol(b; e) = b

² e = raiz(b) ^ eshoja(b) ! eliminarbol(b; e) = nulo

² (e = raiz(b) ^ : eshoja(b)) !

(: esnulo(izq(b)) ^ : esnulo(der(b)) !
eliminarbol(b; e) = crearbin(max(izq(b)); eliminarbol(izq(b);
max(izq(b))); der(b)))

^(esnulo(izq(b)) ^ : esnulo(der(b)) !
eliminarbol(b; e) = der(b))

^(: esnulo(izq(b)) ^ esnulo(der(b)) !
eliminarbol(b; e) = izq(b))

² eshoja(b) _ (: eshoja(b) ^ esnulo(der(b))) !
max(b) = raiz(b)

² : eshoja(b) ^ : esnulo(der(b)) !
max(b) = max(der(b))

Fin-ArbolBusqueda;

Implementación del TAD ArbolBusqueda[Elemento]**:** La implementación del TAD ArbolBusqueda, basada en el TAD ArbolBinario, se encuentra en el apéndice A.

Si consideramos un árbol de búsqueda completo tenemos que el número de elementos es

$$N = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1$$

por lo que la altura $k = O(\log_2(N))$, lo que nos aseguraría el mismo orden para las operaciones insertarbol, eliminarbol y buscarbol. El problema está en que estos operadores no garantizan que los árboles de búsqueda creados sean completos, en particular una secuencia de inserción de claves en orden creciente hace que el árbol de búsqueda sea un árbol degenerado (los árboles izquierdos son siempre árboles nulos). Para aprovechar el orden $\log_2(N)$ que puede lograrse, surge la familia de árboles de búsqueda equilibrados que estudiaremos en la siguiente sección.

8.4.2 Árboles Equilibrados

Dentro de los árboles de búsqueda nos interesan aquellos que verifican que los subárboles tienen alturas similares. Esto requiere que la inserción y la eliminación de elementos preserve la propiedad que el árbol izquierdo y derecho poseen alturas similares. Dos subclases de estos son los árboles AVL y los árboles perfectamente balanceados. Estas estructuras particulares tienen la propiedad que el costo de mantener el equilibrio entre los subarboles no resulta demasiado costoso.

TAD ArbolAVL

Un árbol AVL [5] es un árbol de búsqueda donde las alturas de los subárboles izquierdo y derecho difieren a lo sumo en 1 y ambos son árboles AVL. Ver la Figura 8.4. En los árboles AVL debemos asegurar que las eliminaciones y las inserciones preserven la propiedad de AVL. Se introduce una operación balancear que se aplica a árboles de búsqueda que resultan de haber insertado (eliminado) un elemento a un árbol AVL y como resultado de esa inserción (eliminación) han perdido la propiedad de ser AVL.

[5] En los árboles AVL, la altura del árbol nulo es cero y la de una hoja es 1.

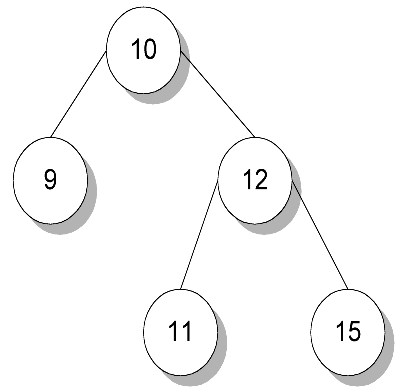


Figura 8.4: Árbol AVL de 5 elementos

TAD ArbolAVL < ArbolBusqueda[Elemento] Sintaxis

nulo :		$\Rightarrow$	ArbolAVL
esnulo :	ArbolAVL	$\Rightarrow$	Boolean
insertarAVL :	ArbolAVL $\times$ Elemento	$\Rightarrow$	ArbolAVL
eliminarAVL :	ArbolAVL $\times$ Elemento	$\Rightarrow$	ArbolAVL
buscarAVL :	ArbolAVL $\times$ Elemento	$\Rightarrow$	ArbolAVL
raiz :	ArbolAVL	$\Rightarrow$	Elemento
izq; der :	ArbolAVL	$\Rightarrow$	ArbolAVL
numnodos :	ArbolAVL	$\Rightarrow$	Natural
eshoja :	ArbolAVL	$\Rightarrow$	Boolean
alturaAVL :	ArbolAVL	$\Rightarrow$	Natural
balancear :	ArbolBusqueda	$\Rightarrow$	ArbolAVL

Semántica
$e \in Elemento; b \in ArbolAVL; b \neq nulo;$

- **Axioma de Comportamiento**
  $|altura(izq(b)) - altura(der(b))| \leq 1$

- $insertarAVL(nulo; e) = crearbin(e; nulo; nulo)$

- $e \prec raiz(b) \rightarrow insertarAVL(b; e) = balancear(crearbin(raiz(b);$
  $insertarAVL(izq(b); e); der(b)))$

- $raiz(b) \preceq e \rightarrow insertarAVL(b; e) = balancear(crearbin(raiz(b);$
  $izq(b); insertarAVL(der(b); e)))$

- $eliminarbol(nulo; e) = nulo$

- $raiz(b) \prec e \wedge \neg esnulo(der(b)) \rightarrow$
  $eliminarAVL(b; e) = balancear(crearbin(raiz(b); izq(b));$
  $eliminarAVL(der(b); e))))$

- $raiz(b) \prec e \wedge esnulo(der(b)) \rightarrow$
  $eliminarAVL(b; e) = balancear(b)$

- $e \prec raiz(b) \wedge \neg esnulo(izq(b)) \rightarrow$
  $eliminarAVL(b; e) = balancear(crearbin(raiz(b);$
  $eliminarAVL(izq(b); e); der(b))))$

- $e \prec raiz(b) \wedge esnulo(izq(b)) \rightarrow$
  $eliminarAVL(b; e) = balancear(b)$

- $e = raiz(b) \wedge eshoja(b) \rightarrow eliminarAVL(b; e) = nulo$

- $(e = raiz(b) \wedge \neg eshoja(b)) \rightarrow$

  $(\neg esnulo(izq(b)) \wedge \neg esnulo(der(b)) \rightarrow$
  $eliminarAVL(b; e) =$
  $balancear(crearbin(max(izq(b)); eliminarAVL(izq(b);$
  $max(izq(b))); der(b))))$

  $\wedge(esnulo(izq(b)) \wedge \neg esnulo(der(b)) \rightarrow$
  $eliminarAVL(b; e) = balancear(der(b)))$

  $\wedge(\neg esnulo(izq(b)) \wedge esnulo(der(b)) \rightarrow$
  $eliminarAVL(b; e) = balancear(izq(b)))$

- $buscarAVL(b; e) = buscarbol(b; e)$

- $eshoja(b) \rightarrow balancear(b) = b$

- **Axioma II**
  $\neg eshoja(b) \wedge (altura(izq(b)) - altura(der(b)) = 2) \wedge$
  $(altura(izq(izq(b))) > altura(der(izq(b)))) \rightarrow$
  $balancear(b) = crearbin(raiz(izq(b)); izq(izq(b)));$
  $crearbin(raiz(b); der(izq(b)); der(b)))$

² **Axioma ID**

: eshoja(b) ^ (altura(izq(b)) ¡ altura(der(b)) = 2)^
(altura(der(izq(b))) > altura(izq(izq(b)))) !
balancear(b) = crearbin(raiz(der(izq(b)));
crearbin(raiz(izq(b)); izq(izq(b)); izq(der(izq(b))));
crearbin(raiz(b); der(der(izq(b))); der(b)))

Fin-ArbolAVL;

La operaci¶n balancear se aplica sobre ¶arboles que eventualmente ya no veri¯can el axioma:

Axioma de Comportamiento

jaltura(izq(b)) ¡ altura(der(b))j · 1

Cuando se realiza una inserci¶n (eliminaci¶n) en un ¶arbol AVL, no necesariamente este pierde su condici¶n de AVL como se muestra en las ¯guras 8.5 y 8.6, por lo que balancear debe dejar el ¶arbol intacto cuando su argumento (¶arbol de b¶usqueda) sea un ¶arbol AVL. A continuaci¶n mostraremos una serie de inserciones, que no alteran la propiedad de AVL hasta llegar a la situaci¶n en que balancear debe actuar para reordenar el ¶arbol y recuperar la condici¶n de AVL.

Analizando los casos de deformaci¶n de un ¶arbol AVL cuando se inserta un elemento

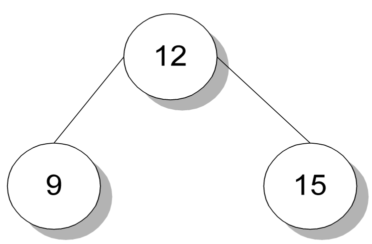


Figura 8.5: ¶Arbol AVL

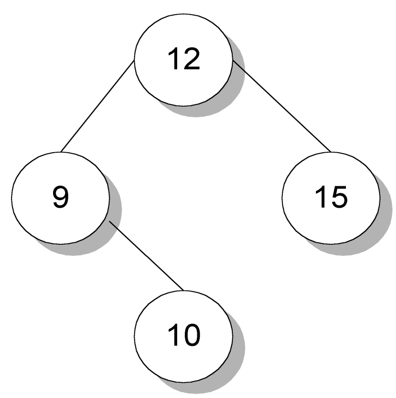


Figura 8.6: ¶Arbol de la ¯g. 8.5 luego de la inserci¶n

Usaremos unos diagramas para representar el desbalanceo en los axiomas II e ID.

Caso II

El árbol de la figura 8.7 presenta un desbalanceo producido por el árbol izquierdo del árbol izquierdo

Cuando se encuentra un árbol resultado de una inserción que no verifica el axioma de AVL con las características de la figura 8.9, aplicando el axioma II resultará un árbol transformado como el de la figura 8.10.

Este árbol lo podemos representar como el diagrama de la figura 8.9 donde los árboles representados por rectángulos corresponden a árboles AVL y los marcados por X son los que realizan la transgresión de la propiedad de ser AVL.

Como vemos en la figura 8.8 es el resultado de aplicar el axioma II al árbol de la figura 8.7.

Caso ID

El árbol de la figura 8.11 presenta un desbalanceo producido por el árbol derecho del árbol izquierdo

Cuando se encuentra un árbol resultado de una inserción que no verifica el axioma de AVL con las características de la figura 8.13, aplicando el axioma ID resultará un árbol transformado como el de la figura 8.14.

Este árbol lo podemos representar como el diagrama de la figura 8.13 donde los árboles representados por rectángulos corresponden a árboles AVL y los marcados por X son los que realizan la transgresión de la propiedad de ser AVL.

Como vemos en la figura 8.12 es el resultado de aplicar el axioma ID al árbol de la figura 8.11.

Se deja como ejercicio al lector escribir los axiomas correspondientes a los casos simétricos (DD y DI).

Para implementar el TAD arbolAVL es conveniente extender el TAD ArbolBusqueda con el siguiente operador:

$$\mathrm{crearbusq} : \mathrm{Elemento} \times \mathrm{ArbolBusqueda} \times \mathrm{ArbolBusqueda} \rightarrow \mathrm{ArbolBusqueda}$$

cuya semántica damos a continuación:

- $\mathrm{esnulo}(b_1) \wedge \mathrm{esnulo}(b_2) \rightarrow \mathrm{crearbusq}(e; b_1; b_2) = \mathrm{crearbol}(e; b_1; b_2)$

- $\neg\mathrm{esnulo}(b_1) \wedge \neg\mathrm{esnulo}(b_2) \wedge \mathrm{raiz}(b_1) < e < \mathrm{raiz}(b_2) \rightarrow$
  $\mathrm{crearbusq}(e; b_1; b_2) = \mathrm{crearbol}(e; b_1; b_2)$

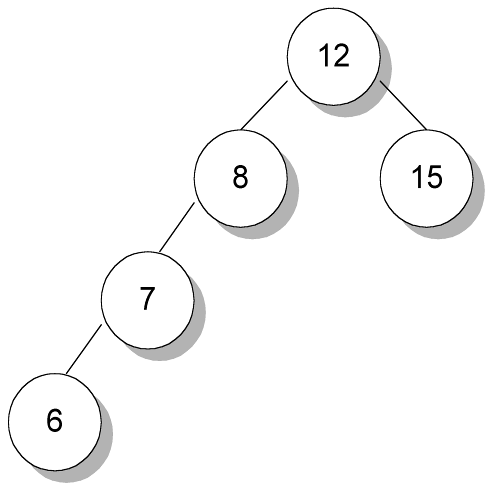


Figura 8.7: Desbalanceo en ¶rbol izquierdo del ¶rbol izquierdo

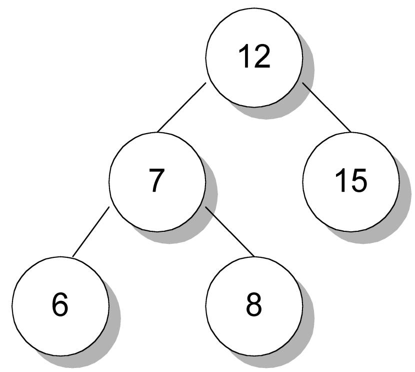


Figura 8.8: Resultado de balancear el ¶rbol de la ¯gura 8.7

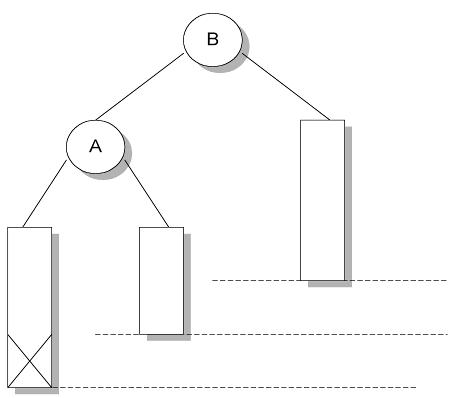


Figura 8.9: Caso II

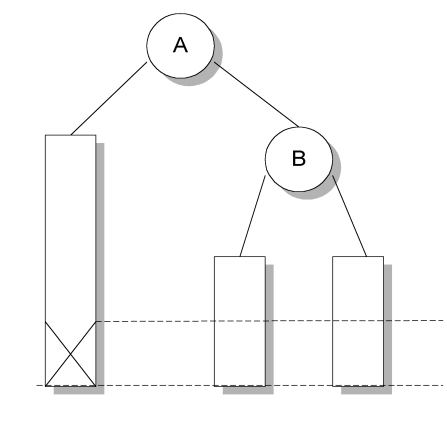


Figura 8.10: Balanceo Usando el axioma II

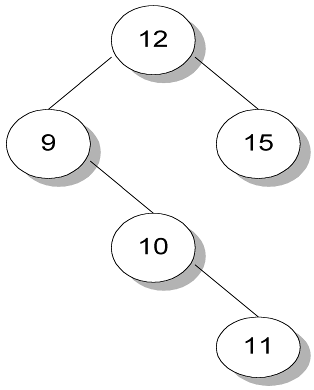


Figura 8.11: Desbalanceo en ¶arbol derecho del ¶arbol izquierdo

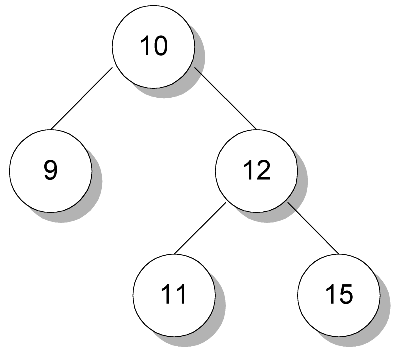


Figura 8.12: Resultado de balancear el ¶arbol de la ¯gura 8.11

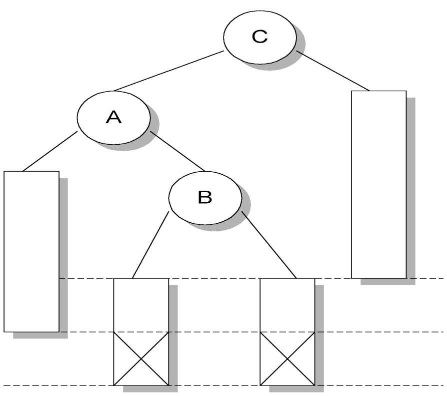


Figura 8.13: Caso ID

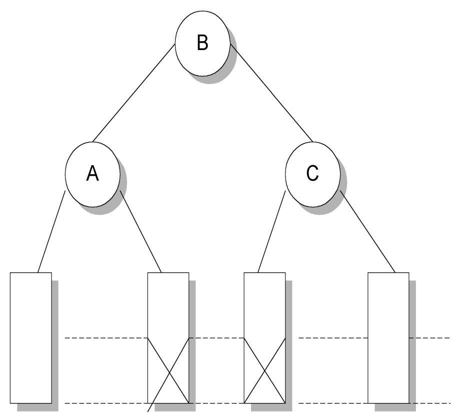


Figura 8.14: Balanceo Usando el axioma ID

² :esnulo(b_1) ^ esnulo(b_2) ^ raiz(b_1) Á e !
crearbusq(e; b_1; b_2) = crearbol(e; b_1; b_2)

² esnulo(b_1) ^ :esnulo(b_2) ^ e Á raiz(b_2) !
crearbusq(e; b_1; b_2) = crearbol(e; b_1; b_2)

Implementación del TAD $ArbolAVL[Elemento]$**:** La implementación del TAD $ArbolAVL$ se encuentra el el apéndice A

8.5 Otros árboles

En esta sección nos ocuparemos de revisar otras estructuras de árboles que permiten al almacenamiento de un conjunto de elementos, una de cuyas componentes es la clave por la que se recupera al elemento.

Árboles Perfectamente Balanceados Un árbol perfectamente balanceado es un árbol de búsqueda donde el número de elementos de los subárboles izquierdo y derecho difieren a lo sumo en 1.

TAD $ArbolPB < ArbolBusqueda[Elemento]$
Sintaxis
  ⋮
Semántica
$\ldots b \in ArbolPB, b \neq nulo;$

- **Axioma de Comportamiento**
  $|\#elem(izq(b)) - \#elem(der(b))| \leq 1$

Fin-ArbolPB;

8.6 Resúmen del capítulo 8

En este capítulo hemos presentado una clase de datos que pueden caracterizarse como un conjunto (o multiconjunto) que posee una relación jerárquica entre los elementos (relación de orden parcial).

En contraposición a los mecanismos presentados en el capítulo 7, la organización de árboles es más estable respecto al tiempo requerido para la operación de búsqueda, admitiendo crecimiento (y decrecimiento) del conjunto sobre el cuál se realiza la búqueda, pagando por ello que las operaciones se realicen en un tiempo que depende del tamaño del conjunto.

Hemos visto que una primera idea de organización en árboles puede dar lugar a árboles degenerados (donde la operación de búsqueda tiene O(N)), por lo que se investiga la introducción de la propiedad de ser balanceado. Vimos que el mantenimiento de esta propiedad puede ser costosa (degradando el tiempo total de operación de la aplicación, si bien conserva el tiempo de búsqueda en O(log (N))), por lo que se analizan métodos para mantener un criterio menos estricto de balance, por ejemplo la propiedad AVL, que permite que las operaciones requeridas para el mantenimiento del balance en las inserciones y eliminaciones se mantenga en el mismo orden que las operaciones de búsqueda, haciendo que todas las operaciones de este TAD resulten de O(log (N))), que resulta aceptable para conjuntos de gran tamaño. En particular los manejadores de bases de datos usan esta organización para aligerar las búsquedas en una tabla de la base de datos.

Debemos hacer notar que tal como lo mencionamos en el capítulo 7, el uso de esta organización de datos en una aplicación puede hacer que inicialmenete la aplicación tenga un comportamiento admisible pero con el tiempo (debido al crecimiento de los conjuntos que se manejan) este comportamiento pase de admisible a no admisible. A diferencia del mecanismo de hashing, la degradación en estos casos no puede resolverse con un mantenimiento y de llegar a ser totalmente inadmisible requiere un rediseño de la aplicación en lo que concierne al almacenamiento del conjunto.

8.7 Ejercicios

1. Implemente el TAD Arbol[Elemento] con estructuras dinámicas.

2. Considere la siguiente operación sobre árboles n-arios:

   $$\text{elem\_por\_niveles} : \text{Arbol} \Rightarrow \text{Cola[Cola[Elemento]]}$$

   Con la siguiente semántica:

   elem_por_niveles(a) da como resultado una cola de colas, donde cada una de estas últimas contiene los elementos que se encuentran a la misma altura (igual nivel). Los primeros elementos en arribar a cada cola son las raices de los subárboles que se encuentran en las primeras posiciones de los sucesores de la raíz del árbol a.

   Implemente la operación elem_por_niveles.

3. Implementa el TAD ArbolBinario[Elemento]:

(a) Con estructuras din¶micas.

(b) Con estructuras est¶ticas.

4. Determinar el m¶ximo n¶mero de elementos de un ¶rbol binario de altura h.

5. Considere la siguiente operaci¶n sobre expresiones en preorden:

$$arbol\_exp : ExpPre \quad) \quad ArbolBinario$$

Con la siguiente sem¶ntica:

arbol_exp(exp) da como resultado el ¶rbol binario asociado a la expresi¶n exp.

Implemente la operaci¶n arbol_exp.

6. Dado los recorridos en preorden y en inorden de un ¶rbol binario, implemente un programa para reconstruir el ¶rbol.

7. **Def.** Un ¶rbol es perfectamente equilibrado si el n¶mero de elementos de sus sub¶rboles di¯eren a lo sumo en uno.

(a) > Cu¶l es la relaci¶n entre ¶rboles perfectamente equilibrados y ¶rboles completos?

(b) > Cu¶l es la altura de un ¶rbol perfectamente equilibrado de N elementos?

(c) > Cu¶l es la ventaja de tener ¶rboles perfectamente equilibrados?

8. Construya, mediante inserciones, los ¶rboles de b¶squeda asociados a cada una de las siguientes secuencias:

(a) $< 100; 50; 170; 25; 55; 115; 193 >$

(b) $< 50; 25; 170; 193; 100; 55; 115 >$

(c) $< 25; 50; 55; 100; 115; 170; 193 >$

(d) $< 193; 170; 115; 100; 55; 50; 25 >$

Para cada caso, determine el costo de buscar el elemento 80. Generalice el costo para buscar cualquier elemento en un arbol de b¶squeda de N elementos. > Cu¶l es su conclusi¶n?

9. Sugiera un algoritmo para crear un árbol binario perfectamente equilibrado a partir de una secuencia.Puede usar este algoritmo si el árbol es de búsqueda. Explique.

10. Escriba un programa que retorne una lista con la frontera de un árbol.

11. Cuántos vértices pueden ser eliminados de un árbol perfectamente balanceado de quince elementos sin que deje de serlo y sin disminuir su altura.

12. Árboles AVL.

    (a) Construya un árbol AVL a partir de la siguiente secuencia de números:
    $$< 5; 10; 15; 6; 3; 2; 7; 12; 20; 70; 51; 36 >$$

    (b) Elimine los siguientes elementos del árbol generado en (a):
    $$20; 10; 15; 7; 70$$

    En cada caso muestre la evolución de los procesos de inserción y eliminación del árbol.

8.8 Bibliografía

1. AHO,Alfred, HOPCROFT,John & ULLMAN,Jeffrey. “Data Structure and Algorithms. Addison Wesley Series in Computer Science.1985

2. HILLE,R.F. “Data Abstraction and Program Development using Modula–2”. Advance in Computer Science. Prentice Hall. 1989.

3. ORTEGA, Maruja y MEZA, Oscar. “Grafos y Algoritmos”. Rep. # IC–1991–002. Dpto. de Computación y Tecn. de la Información. USB.

4. WIRTH, Nicklaus. “Algorithms+Data Structures=Programs”. Prentice Hall.

Apéndice A

Implementación

A.1 Conjunto Estático

Implementación Estática de los operadores del TAD Conjunto

Arreglo de booleanos Sea A un arreglo booleano de dimensión finita y D el universo sobre el cual se define el conjunto.

$$\text{Type CONJUNTO} = A : \text{array}[1..card(D)] \text{ of Boolean}$$

$\{vacio : \Rightarrow Conjunto\}$

```
Function vacio( A: CONJUNTO) CONJUNTO;
begin
    for i=1 to card(D)
      A[i] := FALSE;
    vacio := A
end;
```

Usando la función convert que permite asociar a cada elemento de D un elemento del dominio [1..card(D)]

$\{convert : Elemento \Rightarrow [1..card(D)]\}$

$\{insertar : Conjunto \times Elemento \Rightarrow Conjunto\}$

$\{pre : convert(e) \in [1..card(D)]\}$

```
Procedure insertar( var A: CONJUNTO, e: Elemento);
begin
      A[convert(e)] := TRUE;
end
```

feliminar : Conjunto £ Elemento) Conjuntog
f pre : convert(e) 2 [1::card(D)] g

```
Procedure eliminar( var A: CONJUNTO, e: Elemento);
begin
      A[convert(e)] := FALSE;
end
```

fesvacio : Conjunto) Booleang

```
Function esvacio(A: CONJUNTO): Boolean;
var esta: boolean
begin
      esta := TRUE; i:=1;
      while (i · card(D) and esta)
      begin
         if (A[i] = TRUE) then esta:= FALSE;
         i := i+1;
      end
   esvacio := esta
end
```

fpertenece : Conjunto £ Elemento) Booleang
f pre : convert(e) 2 [1::card(D)] g

```
Function pertenece( A:CONJUNTO, e: Elemento): Boolean)
begin
      return ( A[convert(e)] == TRUE) ;
end
```

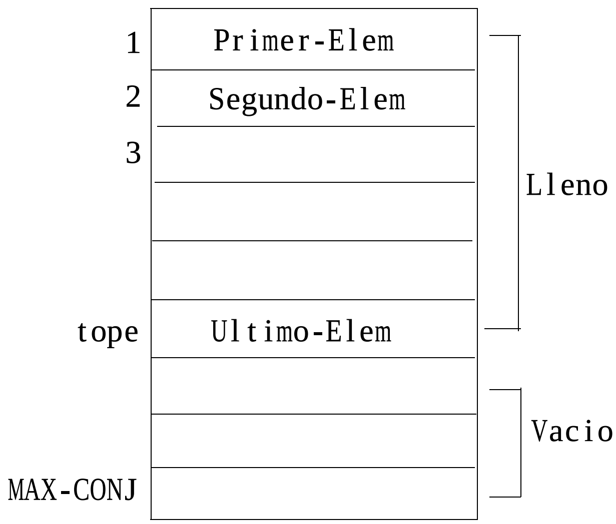


Figura A.1: Arreglo de elementos con repetici¶on

Arreglo de elementos con repetici¶on En esta parte se muestra operaciones admitiendo que el arreglo almacene elementos repetidos, haciendo que la eliminaci¶on requiera revisar la totalidad del arreglo y que la inserci¶on sea mas sencilla.

```
Type CONJUNTO =record
                    tope : 0..MAX-CONJ
                    entrada : array[1..MAX-CONJ] of Tipo-Elem
                  end
```

Algunas operaciones sobre Conjunto son implementadas como sigue
    fvacio : Conjunto) Booleang

```
Procedure vacio(var c: CONJUNTO);
begin
    c.tope := 0
end;
```

$\{insertar : Conjunto \times Elemento \Rightarrow Conjunto\}$

```
Procedure insertar(var c: CONJUNTO; elem:Tipo-Elem);
begin
      if c.tope = MAX-CONJ then ABORT
      else
         begin
            c.tope := c.tope + 1
            c.entrada[c.tope] := elem
         end;
end
```

$\{eliminar : Conjunto \times Elemento \Rightarrow Conjunto\}$

```
Procedure eliminar(var c: CONJUNTO; elem:Tipo-Elem);
var i : 1.. MAX-CONJ
begin
      i := 1
      while (i ≤ c.tope)
         if c.entrada[i] ≠ elem then i := i + 1
         else
           begin
              if(i ≠ c.tope) then
              c.entrada[i] := c.entrada[c.tope];
              c.tope := c.tope - 1;
      end;
end;
```

Arreglo de elementos sin repetición En esta parte se muestra operaciones no admitiendo que el arreglo almacene elementos repetidos, haciendo que la inserción requiera revisar la totalidad del arreglo y que la eliminación sea mas sencilla.

```
Type CONJUNTO = record
                    tope : 0..MAX-CONJ
                    entrada : array[1..MAX-CONJ] of Tipo-Elem
end
```

Algunas operaciones sobre Conjunto son implementadas como sigue

fvacio : Conjunto) Booleang

```
Procedure vacio(var c: CONJUNTO);
begin
    c.tope := 0
end;
```

finsertar : Conjunto £ Elemento) Conjuntog

```
Procedure insertar(var c: CONJUNTO; elem:Tipo-Elem);
begin
    if c.tope = MAX-CONJ then ABORT
    else
     begin
       i := 1;
       while ( c.entrada[i] 6= elem and i · c.tope) i := i + 1
          if( i > c.tope) then
           begin
             c.tope := c.tope + 1
             c.entrada[c.tope] := elem
           end;
     end;
end
```

feliminar : Conjunto £ Elemento) Conjuntog

```
Procedure eliminar(var c: CONJUNTO; elem:Tipo-Elem);
var i : 1.. MAX-CONJ
begin
    i := 1
    while(c.entrada[i] 6= elem and i · c.tope) i := i + 1
    if( i · c.tope) then
     begin
       c.entrada[i] := c.entrada[c.tope];
       c.tope := c.tope - 1;
     end;
end;
```

A.2 Conjunto Dinámico

Implementación Dinámica de los Operadores del TAD Conjunto

La estructura puede ser declarada como

```
Type
      Apunt-Nodo = ^ Nodo
      Nodo = record
         entrada: Tipo-Elem
         Sig-Nodo:Apunt-Nodo
      end
```

La implementación de las operaciones usando una representación dinámica pueden hacerse como se indica a continuación.

$\{insertar : Conjunto \times Elemento \Rightarrow Conjunto\}$

```
Type Conjunto = pointer of Nodo
Function insertar(C:Conjunto;elem:Tipo_Elem):Conjunto
      var p,q: Apunt-Nodo;
      begin
        new(p);
        p^ .entrada := elem;
        p^ .Sig-Nodo := C;
        C := p ;
        insertar := C;
      end
```

La complejidad de este algoritmo es de orden O(1).

$\{eliminar : Conjunto \times Elemento \Rightarrow Conjunto\}$

```
Function eliminar(C:Conjunto; e: Tipo_Elem):Conjunto
var p,q: Apunt-Nodo;
begin
     q := C;
     if C ≠ NIL then
     begin
        if (C^ .entrada ≠ e) then
        begin
```

```
          p := q;
          q := q^ .Sig-Nodo;
          while q^ 6= NIL and q^ .entrada 6= e do
             begin p := q;
                q := q^ .Sig-Nodo;
             end
          if (q^ 6= NIL ) then p^ .Sig-Nodo = q
       end
    else
       C := C^ .Sig-Pos;
end;
```

La complejidad de este algoritmo es £(n) , dado que en el peor de los casos el elemento buscado est¶ en la ¶ultima posici¶n.

f long : Conjunto) Enterog

```
Function long(C:Conjunto):Integer
var p: Apunt-Nodo;
    i: Integer;
begin
if C = NIL then long := 0;
else
    p := C;
    i := 1;
    while p^ .Sig-Nodo 6= NIL do
       begin
         p := p^ .Sig-Nodo;
         i := i + 1;
       end
    od
long:= i;
end
```

La complejidad de esta algoritmo es £(n), dado que hay que recorrer todo el Conjunto

Otra versi¶n de esta misma funci¶n

```
Function long(C:Conjunto):Integer
var p: Apunt-Nodo;
     i: Integer;
     Begin
        if C = NIL then long := 0;
        else
        long:= 1 + long (C^ .Sig-Nodo);
     end
```

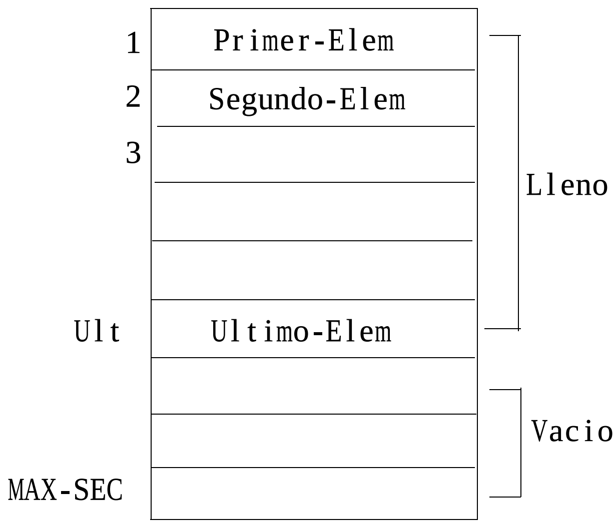


Figura A.2: Secuencia de elementos

A.3 Secuencia Estática

Implementación Estática en PASCAL de los Operadores del TAD Secuencia La representación estática de secuencia utilizará un arreglo (de dimensión N) y un entero (Ult) que registra la última posición del arreglo ocupada con valores de la secuencia. Vemos un diagrama en la figura A.2

$\{vacia : \Rightarrow Secuencia\}$

```
function vacia: Secuencia;
var s: Secuencia;
begin
    s.Ult := 0;
    return(s)
end;
```

$\{insertar : Secuencia \times Entero \times Elemento \Rightarrow Secuencia\}$
$\{pre : 1 \le long(s) + 1\}$

```
function insertar(s: Secuencia; p: Entero; e: Elemento): Secuencia;
var i: Entero;
be gin
   if (s.Ult < N) then
     for i:= s.Ult+1 downto p+1 do
       s.Casilla[i] := s.Casilla[i-1]
     od;
     s.Casilla[p] := e;
     s.Ult := s.Ult +1
   ¯;
   return(s)
end;
```

feliminar : Secuencia £ Entero) Secuenciag f pre : 1 · p · long(s)+ 1 g

```
function eliminar(s: Secuencia; p: Entero);
var i: Entero;
begin
   s.Ult := s.Ult -1;
   for i:= p to s.Ult do
     s.Casilla[i] := s.Casilla[i+1]
   od
   return(s)
end;
```

fesvacia : Secuencia) Booleang

```
function esvacia(s: Secuencia): Boolean;
begin
   return(s.Ult == 0)
end;
```

festa : Secuencia £ Elemento) Booleang

```
function esta(s: Secuencia; e: Elemento): Boolean;
var i: Entero;
begin
   if(s.Ult = 0) then
```

```
      return(false)
    else
      i := 1;
      while (i < s.Ult) and (s.Casilla[i] =6 e) do
         i := i + 1
      od;
      return(s.Casilla[i] = e)
    ¯ end;
```

flong : Secuencia) Enterog f pre : 1 · long(s) + 1 g

```
function long(s: Secuencia): Entero;
begin
    return(s.Ult)
end;
```

fproyectar : Secuencia £ Entero) Elementog f pre : 1 · p · long(s) g

```
function proyectar(s: Secuencia; p: Entero): Elemento;
begin
    return(s.Casilla[p])
end;
```

A.4 Secuencia Din¶mica

Implementaci¶n Din¶mica en PASCAL de los Operadores del TAD Secuencia Para esta implementaci¶n se usar¶n las funciones de manejo de memoria de PASCAL (new y dispose para la obtenci¶n y liberaci¶n de memoria).

fvacia :) Secuenciag

```
function vacia: Secuencia;
begin
  return(nil)
end;
```

finsertar : Secuencia £ Entero £ Elemento) Secuenciag
f pre : (1 <= p <= long(s) + 1) g

```
function insertar(s: Secuencia; p: Entero; e: Elemento): Secuencia;
var s1,s2: Secuencia;
begin
    new(s1);
    s1^.Elem := e;
    if ( p = 1 ) then
      s1^.Sig := s;
      s := s1
    else
      s2 := s;
      for i:=2 to p-1 do
          s2 := s2^.Sig
      od;
      s1^.Sig := s2^.Sig;
      s2^.Sig := s1
    _;
return(s)
end;
```

feliminar : Secuencia £ Entero) Secuenciag
f pre : (1 <= p <= long(s)) g

```
function eliminar(s: Secuencia; p: Entero);
var s1,s2: Secuencia;
begin
    if ( p = 1 ) then
      s1 := s;
      s := s^.Sig;
      dispose(s1)
    else
      s1 := s;
      for i:=2 to p-1 do
        s1 := s1^.Sig
      od;
      s2 := s1^.Sig;  La casilla a eliminar
      s1^.Sig := s2^.Sig;
      dispose(s2)
    _;
```

```
    return(s)
end;

    fesvacia : Secuencia ) Booleang

function esvacia(s: Secuencia): Boolean;
begin
    return(s = nil)
end;

    festa : secuencia £ Elemento ) Booleang

function esta(s: Secuencia; e: Elemento): Boolean;
var Ocur: Boolean;
begin
    Ocur := false;
    while(s =6 nil) and not(Ocur) do
        Ocur : = (s^.Elem = e);
        s := s^.Sig
    do;
    return(Ocur)
end;

    flong : secuencia ) Enterog

function long(s: Secuencia): Entero;
var i: Entero;
begin
    i = 0;
    while(s =6 nil) do
        i:= i + 1;
        s := s^.Sig
    od;
    return(i)
end;

    fproyectar : secuencia £ Entero ) Elementog
    f pre : (1 <= p <= long(s)) g
```

```
function proyectar(s: Secuencia; p: Entero): Elemento;
begin
   for i := 1 to p-1 do
      s := s^.Sig
   od;
   return(s.Elem)
end;
```

A.5 Pila Est¶tica

Implementaci¶n Est¶tica de los Operadores de TAD Pila Para implementar una pila en forma est¶tica se usa un arreglo donde se almacena los elementos y un contador que indica cuantas entradas hay en el arreglo. Por ser est¶tico es necesario declarar el tamaño m¶ximo permitido del arreglo (MAX-Pila) as¶ como el tipo de elementos a almacenar en la estructura o arreglo (Tipo-Elem).

```
Type Pila =record
           top :            0..MAX-Pila
           entrada :        array[1..MAX-Pila] of Tipo-Elem
           end
```

Las operaciones de empilar y desempilar sobre la Pila son implementadas como sigue
fempilar : P ila £ Elemento) P ilag

```
procedure empilar(var P:Pila; e:Tipo-Elem);
 begin
  with P do
    if top = MAX-Pila
      then
        writeln('Error:intenta almacenar un elemento en una pila llena')
      else begin
          top := top +1;
          entrada[top] := e
          end
  end;
```

fdesempilar : P ila) P ilag

```
procedure desempilar(var P:Pila);
  begin
   with P do
   if top = 0
     then
           writeln('Error: desempilando de una pila vacia')
     else  begin
           top := top -1
           end
  end;
```

fvacia$_P$: P ila) P ilag

```
function vaciaP( ):Pila
  begin
  P.top := 0
  return P
  end;
```

fesvacia$_P$: P ila) Booleang

```
function esvaciaP( P:Pila):Boolean
  begin
   with P do
   if top = 0
     then
           return TRUE
   else
     begin
           return FALSE
     end
  end;
```

ftope : P ila) Elementog

f pre : esvacia$_P$ (P) = F ALSE g

```
function tope( P:Pila ):Tipo-Elem
  begin
   with P do
   if top = 0
     then
          writeln('Error: intenta sacar un elemento de una pila vacia')
     else  begin
          return P.entrada[top]
          end
  end;
```

Con esta implementaci¶n la operaci¶n de empilar es de orden £(n).

A.6 Pila Din¶mica

Implementaci¶n Din¶mica de los Operadores del TAD Pila

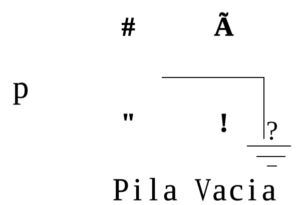


Figura A.3: Representaci¶n de la pila vac¶a

```
Type
      Apunt-Pila = ^NodoPila
      NodoPila =   record
                   entrada:   Tipo-Elem
                   Sig-Nodo: Apunt-Pila
                   end

Type Pila =        Apunt-pila
```

f esvacia$_p$: P ila) Booleang

```
function esvaciap( p:Pila):boolean
  begin
   if p = NIL
     then
          return TRUE
     else
          return FALSE
  end
```

f tope : P ila) Elementog

```
function tope( p:Pila):Tipo-Elem
  begin
   return p^.entrada
  end;
```

f empilar : P ilaX Elemento) P ilag

```
procedure empilar(var p:Pila; e:Tipo-Elem);
  var q: Apunt-Pila;
   begin
    new(q);
    if q = NIL
      then
           writeln('Error:intento de a~nadir un nodo inexistente')
      else
           begin
           q^.entrada := e;
           q^.Sig-Nodo := p;
           p := q;
           end
   end;
```

f desempilar : P ila) P ilag

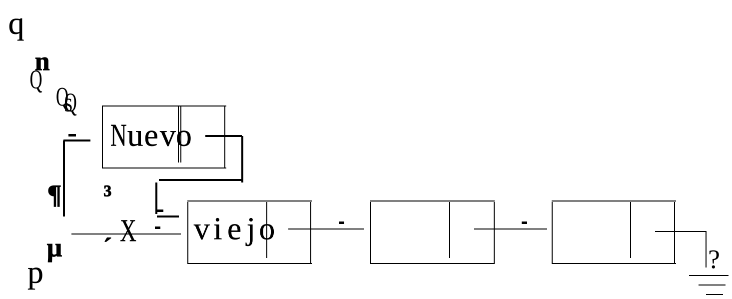


Figura A.4: Agregando un NODO a la PILA

```
procedure desempilar(var p:Pila);
  var q:  Apunt-Pila;
   begin
    if esvacia_p(p)
       then
              writeln('Intento de eliminar un elemento de una pila vacia');
       else
              begin
              q := p;
              p := q^.Sig-Nodo
              dispose(q);
              end
   end;
```

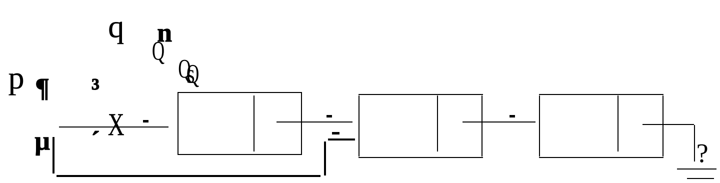


Figura A.5: Eliminando el elemento TOPE

A.7 Cola Est¶tica

Implementaci¶n Est¶tica de los Operadores del TAD Cola .

```
Type  Cola record
          cont : 0..MAX-Cola;
          Primero :  0..MAX-Cola;
          Ultimo :   0..MAX-Cola;
          entrada :  array[1..MAX-Cola] of Tipo-Elem;
         end


var C : Cola;
```

f $vacia_C$:) Colag

```
procedure vacia_C();
  begin
        with C do begin
         cont := 0;
         Primero := 1;
         Ultimo := 0;
        end
  end;
```

f $esvacia_C$: Cola) Booleang

```
function esvacia_C(C:Cola):boolean;
 begin
       with C do begin
        return (cont == 0)
  end;
```

f encolar : ColaeElemento) Colag

```
Procedure encolar(var C:Cola, e:Tipo-Elem );
 begin
       with C do
         if (cont == MAX-Cola) then
         Writeln('ERROR: Cola Llena')
       else
       begin
       cont := cont +1;
       Ultimo := ( Ultimo mod MAX-Cola) + 1;
       entrada[Ultimo] := e;
end;
end
```

fdesencolar : Cola) Colag

```
Procedure desencolar(var C:Cola);
begin
   with C do
        if (count == 0) then
          writeln('ERROR: intento de eliminar de una cola vacia');
        else
          begin
          cont := cont - 1;
          Primero := (Primero mod MAX-Cola)+1;
        end;
end
```

ff rente : Cola) Elementog

```
Procedure f rente( C:Cola, var e:Tipo-Elem);
 begin
     with C do
          if (cont == 0) then
            writeln('ERROR: intento de ver en una cola vacia');
          else
            begin
                e := entrada[Primero];
```

```
            end
  end
```

A.8 Cola Din¶mica

Implementaci¶n Din¶mica de los Operadores del TAD Cola

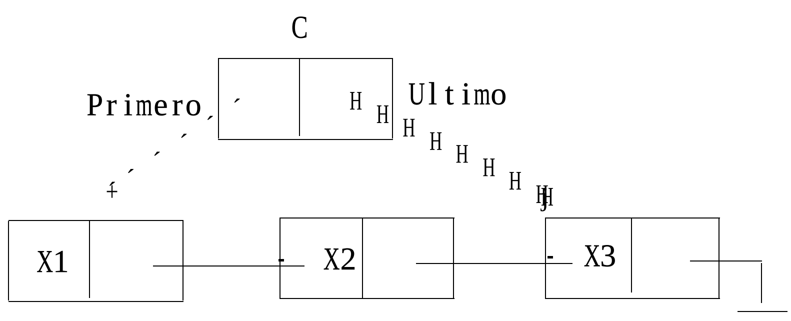


Figura A.6: Cola din¶mica

```
Type
      Apunt-Cola : ^NodoCola
      NodoCola =   record
                   entrada: Tipo-Elem
                   Sig-Nodo:Apunt-Cola
                   end

      Cola =       record
                   Primero : Apunt-Cola;
                   Ultimo :  Apunt-Cola
                   end
```

fvacia$_C$:) Colag

```
var C : Cola

Procedure vaciaC();
  begin
   C.Primero := NIL;
   C.Ultimo := NIL
  end;
```

fencolar : ColaeElemento) Colag

```
Procedure encolar(var C:Cola, e:Tipo-Elem);
var p: Apunt-Cola;
  begin
   new(p );
   p^.entrada := e;
  p^.Sig-Nodo := NIL
  with C do
   if Primero == NIL
     then
          Primero := p;
          Ultimo := p;
    else
        Ultimo^.Sig-Nodo := p;
        Ultimo := p;
    _
  end
```

fdesencolar : Cola) Colag

```
Procedure desencolar(var C:Cola);
var p:Apunt-Cola;
  begin
   with C do
    if Primero == NIL
      then
```

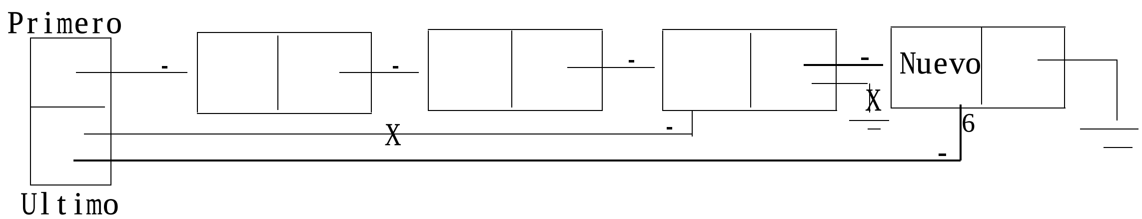


Figura A.7: Sumar un elemento a la Cola

```
          writeln('ERROR intento de eliminar de una cola vacia');
    else
        begin
              p := Primero;
              Primero := Primero^.Sig-Nodo;
              if Primero == NIL then
                Ultimo := NIL
              dispose(p);
        end
end
```

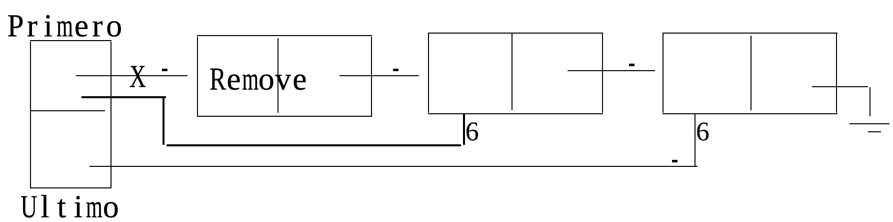


Figura A.8: Remueve un elemento del frente

A.9 Árbol Binario

Implementación Dinámica de los Operadores de TAD ArbolBinario Se describe a continuación la implementaci¶n din¶mica de ¶rboles.

```
type ArbolBinario = ^ NodoArbol;
      NodoArbol = record
                        elem: Elemento;
                        izq, der: ArbolBinario
      end;

    fcrearbin : ElementoXArbolBinarioXArbolBinario ) ArbolBinariog

function crearbin(e: Elemento; bi, bd: ArbolBinario): ArbolBinario;
var b : ArbolBinario;
begin
     new(b); b^.elem := e;
     b^.izq := bi; b^.der := bd;
     crearbin := b
end;

    fnulo : ) ArbolBinariog

function nulo: ArbolBinario;
begin
    nulo := nil
end;

    fesnulo : ArbolBinario  ) Boolean g

function esnulo(b: ArbolBinario): Boolean;
begin
     esnulo := (b = nil)
end;

    fraiz : ArbolBinario ) Elementog
    fpre : not(esnulo(b))g

function raiz(b: ArbolBinario) : Elemento;
begin
     raiz := b^.elem
end;
```

```
   fder : ArbolBinario ) ArbolBinariog
   fpre : not(esnulo(b))g

function der(b: ArbolBinario) :ArbolBinario;
begin
   der := b^.der
end;

   fizq : ArbolBinario ) ArbolBinariog
   fpre : not(esnulo(b))g

function izq(b: ArbolBinario) : ArbolBinario;
begin
   izq := b^.izq
end;

   feshoja : ArbolBinario ) Booleang

function eshoja(b: ArbolBinario): Boolean;
begin
   eshoja := not(esnulo(b)) and
              (b^.izq = nil and b^.der = nil)
end;

   f#elem : ArbolBinario ) Naturalg

function #elem(b: ArbolBinario): Natural;
begin
   if esnulo(b) then #elem:= 0
   else
     #elem := 1 + #elem(b^.izq) + #elem(b^.der)
end;

   faltura : ArbolBinario ) Naturalg
   fpre : not(esnulo(b))g

function altura(b: ArbolBinario): Natural;
begin
   if eshoja(b) then altura := 0
   else
     altura := 1 + max(altura(b^.izq),altura(b^.der))
end;
```

La funci¶n max calcula el m¶ximo entre dos n¶meros naturales.

A.10 ¶Arbol de B¶usqueda

Implementaci¶on del TAD ArbolBusqueda A continuaci¶on presentamos una implementaci¶on del TAD ArbolBusqueda, basada en el TAD ArbolBinario.

```
  type ArbolBusqueda = ArbolBinario;
  fbuscarbol : ArbolBusqueda £ Elemento ) ArbolBusquedag

function buscarbol(b: ArbolBusqueda; e: Elemento): ArbolBusqueda;
begin
   if esnulo(b) then return(nulo)
   elseif (raiz(b) = e) then return(b)
      elseif (raiz(b) < e) then
          return(buscarbol(der(b),e))
         else
          return(buscarbol(izq(b),e))
         –
      –
   –
end;

  finsertarbol : ArbolBusqueda £ Elemento ) ArbolBusquedag

function insertarbol(b:ArbolBusqueda; e: Elemento): ArbolBusqueda;
begin
   if esnulo(b) then return(crearbin(e, nulo, nulo))
   else if (raiz(b) = e) then return(b)
       elseif(raiz(b) < e) then
          b1 := insertarbol(der(b),e);
          return(crearbin(raiz(b), izq(b), b1))
         else
          b1:= insertarbol(izq(b),e);
          return(crearbin(raiz(b), b1, der(b)))
         –
       –
   –
end;
```

feliminarbol : ArbolBusqueda £ Elemento) ArbolBusqueda g
fpre : not(esnulo(b)) g

La funci¶on eliminararbol utiliza como auxiliar la funci¶on borrar_max que recibe como par¶ametro un ¶arbol y elimina el elemento m¶aximo del mismo, devolviendo en el segundo par¶ametro el valor de ese elemento m¶aximo.

```
function eliminarbol(b:ArbolBusqueda; e: Elemento): ArbolBusqueda;
var max: Elemento;
function borrar_max (b: ArbolBusqueda; var m: Elemento): ArbolBusqueda;
begin borrar_max
   ifnot(eshoja(b)) and not(esnulo(der(b)) then
    return(crearbin(raiz(b),izq(b),borrar_max(der(b),m)))
   else
    m := raiz(b);
    return(izq(b))
   _

end borrar_max
begin eliminarbol
   if esnulo(b) then return(b)
   else
    if(raiz(b) = e) then encontro el elemento
     if eshoja(b) then return(nulo)
     else debe eliminar un nodo interno
        if esnulo(der(b)) then
        return(izq(b))
        else
           if esnulo(izq(b)) then
           return(der(b))
           else
           b1 := borrar_max (izq(b), max);
           return(crearbin(max, b1, der(b)))
           _
        _
     _

    else continua la busqueda
     if (raiz(b) < e) then
        if (esnulo(der(b)) then return(b)
        else
```

```
            b1 := eliminarbol(der(b),e);
            return(crearbin(raiz(b), izq(b), b1))
         _

      else
         if (esnulo(izq(b)) then return(b)
         else
            b1:= eliminarbol(izq(b),e);
            return(crearbin(raiz(b), b1, der(b)))
         _
      _
    _
   _
end; eliminarbol
```

A.11 Árbol AVL

Implementación del TAD ArbolAVL[Elemento] type ArbolBusqueda = ArbolBusqueda; faltura : ArbolBusqueda) Naturalg

```
function alturaAVL(b: ArbolAVL): Integer;
begin
   if esnulo(b) then alturaAVL := 0
   else
   _return(1 + max(altura(izq(b),altura(der(b))))
   _

end;

   fbalancear : ArbolBusqueda ) ArbolAVLg

function balancear(b: ArbolAVL): ArbolAVL;
begin
   if (esnulo(b) or eshoja(b)) then balancear := nulo
   else if abs(alturaAVL(izq(b)) - alturaAVL(der(b)) = 2) then
     Cargado hacia la izquierda
     if (alturaAVL(izq(izq(b)) > alturaAVL(der(izq(b)))
     then  Axioma DD
     return(crearbusq(raiz(izq(b)), izq(izq(b)),
            crearbusq(raiz(b),der(izq(b)),
```

```
            der(b))))
        else  Axioma ID
            return(crearbusq(raiz(der(izq(b))),
            crearbusq(raiz(izq(b)),izq(izq(b)),izq(der(izq(b)))),
            der(b)))
        _

        else Cargado hacia la derecha
.......
_

end balancear
```

fbuscarAV L : ArbolAV L £ Elemento) ArbolAV Lg

```
function buscarAV L(b: ArbolAVL; e: Elemento): ArbolAVL;
begin
    if esnulo(b) then return(nulo)
    else
        if (raiz(b) = e) then return(b)
        else if (raiz(b) < e) then
            return(buscarAV L(der(b),e))
            else
            _  return(buscarAV L(izq(b),e))

        _
    _

end;
```

finsertarAV L : ArbolAV L £ Elemento) ArbolAV Lg

```
function insertarAV L(b:ArbolAVL; e: Elemento): ArbolAVL;
begin
    if esnulo(b) then
        return(crearbusq(e, nulo, nulo))
    else
        if (raiz(b) = e) then return(b)
        else
            if (raiz(b) < e) then
                b1 := insertarAV L(der(b),e);
```

```
            return(balancear(crearbusq(raiz(b),izq(b),b1)))
         else
            b1:= insertarAV L(izq(b),e);
            return(balancear(crearbusq(raiz(b),b1,der(b)))
         _
      _
   _
end;

   feliminarAV L : ArbolAV L £ Elemento ) ArbolAV Lg
   fpre : not(esnulo(b)) g

function eliminarAV L(b:ArbolBusqueda; e: Elemento): ArbolBusqueda;
var max: elemento;
function borrar_max (b: ArbolAVL; var m: Elemento): ArbolAVL;
begin borrar_max
   if not(eshoj a(b)) and not(esnulo(der(b)) then
      return(balancear(crearbusq(raiz(b),izq(b),borrar_max(der(b),m))))
   else
      m := raiz(b);
      return(izq(b))
   _
end borrar_max
begin eliminarAVL
   if esnulo(b) then return(nulo)
   else
      if (raiz(b) = e) then encontro el elemento
         if eshoj a(b) then return(nulo)
         else debe eliminar un nodo interno
               if esnulo(der(b)) then
               return(izq(b))
               else
                  if esnulo(izq(b)) then
                  return(der(b))
                  else
                     b1 := borrar_max (izq(b), max);
                     return(balancear(crearbusq(max, b1, der(b))))
                  _
```

```
                    _
            _

        else  continua la busqueda
    if (raiz(b) < e) then
            b1 := eliminarbol(der(b),e);
            return(balancear(crearbin(raiz(b), izq(b), b1)))
        else
            b1:= eliminarbol(izq(b),e);
            return(balancear(crearbin(raiz(b), b1, der(b))))
        _
    _
_

end;  eliminarAVL
```

Apéndice B

Estructuración de los Tipos Abstractos de Datos

En este apéndice se relacionan los tipos abstractos (TAD) cubiertos en este libro indicando las relaciones existentes entre ellos. Presentamos dos tipos de relaciones diferentes:

- Las relaciones que sobre el tipo Elemento.

- Las relaciones espaciales (o relaciones de accesibilidad) entre los elementos.

En ambos casos las relaciones que se establecen son relaciones de orden (parcial o total).

Los TAD estudiados se parametrizaron con el tipo Elemento. Como se indica en la figura B.1 los dos tipos abstractos base son el conjunto y el multiconjunto, dado que representan colecciones de objetos cuya única propiedad es la de pertenecer a un mismo conjunto. Los demás TAD se obtienes de conjuntos y multiconjuntos agregandoles relaciones espaciales o relaciones a los elementos.

Cuando se le agrega al conjunto o al multiconjunto un ORDEN TOTAL en el espacio como organización surge el tipo abstracto secuencia. La secuencia es un ordenamiento espacial de los elementos de un conjunto (multiconjunto). Surge así la noción de puesto de un elemento.

Los tipos COLA,PILA y DIPOLO son especializaciones del tipo secuencia poniendo restricciones a las políticas de ingreso y egreso de los elementos a una secuencia.

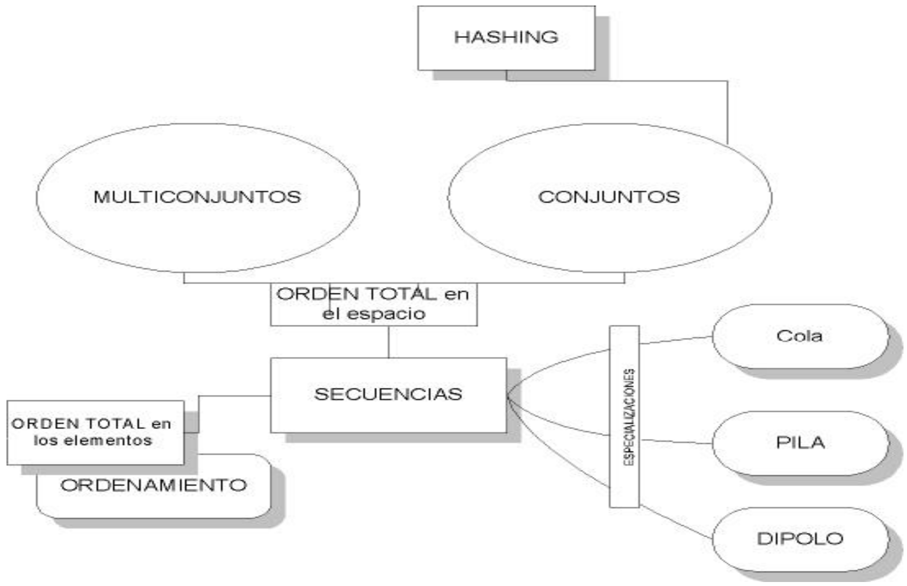


Figura B.1: Relaciones entre los TDA

Si adicionalmente el TAD Elemento posee un orden total sobre él, sea este $\leq$, sumado al orden espacial vemos que es interesante hacer coincidir el orden espacial con el orden total de los elementos. Esto lo expresamos como

$$i < j \Leftrightarrow proyectar(s; i) \leq proyectar(s; j)$$

.

Para lograr esto se revisan los métodos de ordenamiento de una secuencia que reciben una secuencia y la transforman de manera que el puesto de un elemento refleje el orden total de los elementos de la secuencia, lo que equivale a decir que el elemento del primer puesto de la secuencia resultado será el mínimo del conjunto y el máximo del conjunto ocupará el último puesto.

Si a los tipos Conjunto y Multiconjunto se le agrega un orden parcial como organización espacial surge el tipo abstracto árbol, en su forma más general de árbol n-ario a la especialización de árbol binario.

En la figura B.2 se muestra el TAD árboles de búsqueda que se obtienen organizando Conjunto y Multiconjunto de Elemento que posea un orden total, con un orden parcial espacial.

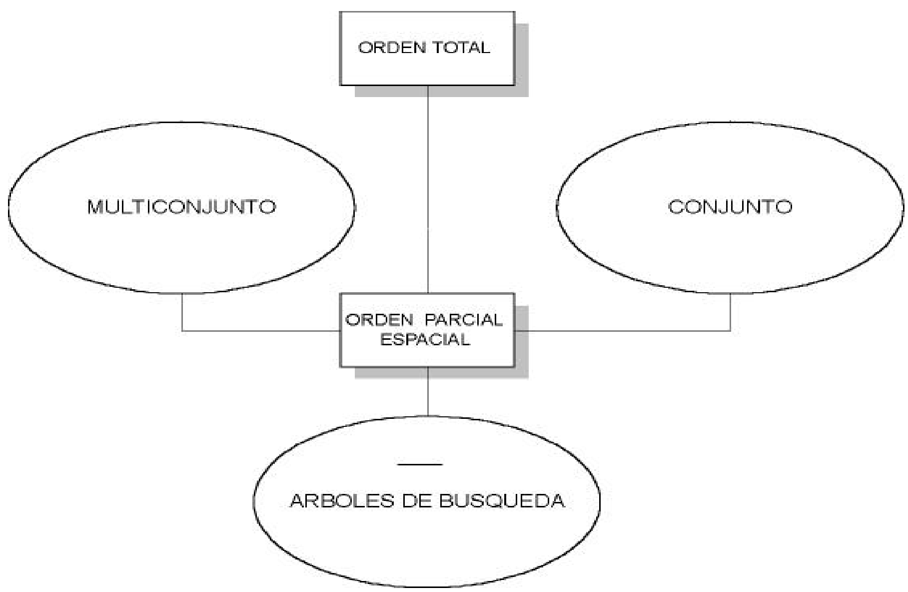


Figura B.2: ¶Arboles

Ap¶ndice C

¶Indice de materias

¶Indice de Materias