LIBRO

Algoritmos y Estructura de Datos

Ingeniería en sistemas computacionales

Quinto

Enero - Abril

Marco Estratégico de Referencia

Antecedentes históricos

Nuestra Universidad tiene sus antecedentes de formación en el año de 1979 con el inicio de actividades de la normal de educadoras "Edgar Robledo Santiago", que en su momento marcó un nuevo rumbo para la educación de Comitán y del estado de Chiapas. Nuestra escuela fue fundada por el Profesor Manuel Albores Salazar con la idea de traer educación a Comitán, ya que esto representaba una forma de apoyar a muchas familias de la región para que siguieran estudiando.

En el año 1984 inicia actividades el CBTiS Moctezuma Ilhuicamina, que fue el primer bachillerato tecnológico particular del estado de Chiapas, manteniendo con esto la visión en grande de traer educación a nuestro municipio, esta institución fue creada para que la gente que trabajaba por la mañana tuviera la opción de estudiar por las tardes.

La Maestra Martha Ruth Alcázar Mellanes es la madre de los tres integrantes de la familia Albores Alcázar que se fueron integrando poco a poco a la escuela formada por su padre, el Profesor Manuel Albores Salazar; Víctor Manuel Albores Alcázar en julio de 1996 como chofer de transporte escolar, Karla Fabiola Albores Alcázar se integró en la docencia en 1998, Martha Patricia Albores Alcázar en el departamento de cobranza en 1999.

En el año 2002, Víctor Manuel Albores Alcázar formó el Grupo Educativo Albores Alcázar S.C. para darle un nuevo rumbo y sentido empresarial al negocio familiar y en el año 2004 funda la Universidad Del Sureste.

La formación de nuestra Universidad se da principalmente porque en Comitán y en toda la región no existía una verdadera oferta Educativa, por lo que se veía urgente la creación de una institución de Educación superior, pero que estuviera a la altura de las exigencias de los jóvenes que tenían intención de seguir estudiando o de los profesionistas para seguir preparándose a través de estudios de posgrado.

Nuestra Universidad inició sus actividades el 18 de agosto del 2004 en las instalaciones de la 4ª avenida oriente sur no. 24, con la licenciatura en Puericultura, contando con dos grupos de cuarenta alumnos cada uno. En el año 2005 nos trasladamos a nuestras propias instalaciones en la carretera Comitán – Tzimol km. 57 donde actualmente se encuentra el campus Comitán y el corporativo UDS, este último, es el encargado de estandarizar y controlar todos los procesos operativos y educativos de los diferentes campus, así como de crear los diferentes planes estratégicos de expansión de la marca.

Misión

Satisfacer la necesidad de Educación que promueva el espíritu emprendedor, aplicando altos estándares de calidad académica, que propicien el desarrollo de nuestros alumnos, Profesores, colaboradores y la sociedad, a través de la incorporación de tecnologías en el proceso de enseñanza-aprendizaje.

Visión

Ser la mejor oferta académica en cada región de influencia, y a través de nuestra plataforma virtual tener una cobertura global, con un crecimiento sostenible y las ofertas académicas innovadoras con pertinencia para la sociedad.

Valores

- Disciplina
- Honestidad
- Equidad
- Libertad

Escudo

El escudo del Grupo Educativo Albores Alcázar S.C. está constituido por tres líneas curvas que nacen de izquierda a derecha formando los escalones al éxito. En la parte superior está situado un cuadro motivo de la abstracción de la forma de un libro abierto.

Eslogan

"Mi Universidad"

ALBORES

Es nuestra mascota, un Jaguar. Su piel es negra y se distingue por ser líder, trabaja en equipo y obtiene lo que desea. El ímpetu, extremo valor y fortaleza son los rasgos que distinguen.

Algoritmos y Estructura de Datos

Objetivo de la materia:

Independientemente del sistema de evaluación elegido para la convocatoria ordinaria y en concordancia con lo que se ha venido haciendo en el plan de estudios de 1996, los alumnos con el proyecto o la práctica de Estructuras

CRITERIOS DE EVALUACIÓN:

No	Concepto	Porcentaje
1	Trabajos Escritos	10%
2	Actividades web escolar	20%
3	Actividades Áulicas	20%
4	Examen	50%
	Total de Criterios de evaluación	100%

UNIDAD I

DISEÑO Y ANÁLISIS DE ALGORITMOS

1.1 Conceptos básicos de algoritmos

1.1.1 Introducción básica a grafos

1.1.2. Planteo del problema mediante grafos

1.1.3. Algoritmo de búsqueda exhaustiva

1.1.4. Generación de las coloraciones

1.2. Tipos abstractos de datos

1.2.1. Operaciones abstractas y características del tad conjunto

1.2.2. Interfaz del tad conjunto

1.2.3. Implementación del tad conjunto

1.3. Tiempo de ejecución de un programa

1.4. Conteo de operaciones para el cálculo del tiempo de ejecución

1.4.1. Bloques if

1.4.2. Lazos

1.4.3. Suma de potencias

1.4.4. Llamadas a rutinas

UNIDAD II

TIPOS DE DATOS ABSTRACTOS FUNDAMENTALES

2.1. El tad lista

2.1.1. Descripción matemática de las listas

2.1.2. Operaciones abstractas sobre listas

2.2. El tad pila

2.2.1. Una calculadora rpn con una pila

2.3. El tad cola

2.3.1. Intercalación de vectores ordenados

2.3.1.1. Ordenamiento por inserción

2.3.1.2. Tiempo de ejecución

2.3.1.3. Particularidades al estar las secuencias pares e impares ordenadas

2.3.1.4. Algoritmo de intercalación con una cola auxiliar

2.4. El tad correspondencia

2.4.1. Interfaz simple para correspondencias

2.4.2. Implementación de correspondencias mediante contenedores lineales

UNIDAD III

ÁRBOLES

3.1. Nomenclatura básica de árboles

3.1.0.0.2. Profundidad de un nodo. nivel

3.1.0.0.3. Nodos hermanos

3.2. Orden de los nodos

3.2.1. Particionamiento del conjunto de nodos

3.2.2. Listado de los nodos de un árbol

3.2.2.1. Orden previo

3.2.2.2. 0rden posterior

3.2.2.3. Orden posterior y la notación polaca invertida

3.3. Operaciones con árboles

3.3.1. Algoritmos para listar nodos

3.3.2. Inserción en árboles

3.3.2.1. Algoritmo para copiar árboles

3.5. Implementación de la interfaz básica por punteros

3.5.1. El tipo iterator

3.5.2. Las clases cell e iterator_t

Mi Universidad

UNIDAD IV

ORDENAMIENTO

4.1 Introducción

4.1.1 Relaciones de orden débiles

4.1.2 Signatura de las relaciones de orden. predicados binarios

4.2 Métodos de ordenamiento lentos

4.2.1 El método de la burbuja

4.2.2 El método de inserción

4.2.3 El método de selección

4.3 Ordenamiento indirecto

4.3.1 Minimizar la llamada a funciones

4.4. El método de ordenamiento rápido, quick-sort

4.4.1 Tiempo de ejecución. casos extremos

4.4.2 Elección del pivote

INDICE

Contenido

Misión....4
Visión....4
Valores....5
Escudo....5
Eslogan....6
ALBORES....6
UNIDAD I....14
DISEÑO Y ANÁLISIS DE ALGORITMOS....14
1.1 CONCEPTOS BÁSICOS DE ALGORITMOS....14
1.1.1 INTRODUCCIÓN BÁSICA A GRAFOS....15
1.1.2. PLANTEO DEL PROBLEMA MEDIANTE GRAFOS....17
1.1.3. ALGORITMO DE BÚSQUEDA EXHAUSTIVA....19
1.1.4. GENERACIÓN DE LAS COLORACIONES....19
1.2. TIPOS ABSTRACTOS DE DATOS....21
1.2.1. OPERACIONES ABSTRACTAS Y CARACTERÍSTICAS DEL TAD CONJUNTO....23
1.2.2. INTERFAZ DEL TAD CONJUNTO....23
1.2.3. IMPLEMENTACIÓN DEL TAD CONJUNTO....26
1.3. TIEMPO DE EJECUCIÓN DE UN PROGRAMA....26
1.4. CONTEO DE OPERACIONES PARA EL CÁLCULO DEL TIEMPO DE EJECUCIÓN....29
1.4.1. BLOQUES IF....30
1.4.2. LAZOS....31
1.4.3. SUMA DE POTENCIAS....33
1.4.4. LLAMADAS A RUTINAS....34
1.4.5. LLAMADAS RECURSIVAS....34
UNIDAD II....36
TIPOS DE DATOS ABSTRACTOS FUNDAMENTALES....36

2.1. EL TAD LISTA 39
2.1.1. DESCRIPCIÓN MATEMÁTICA DE LAS LISTAS 40
2.1.2. OPERACIONES ABSTRACTAS SOBRE LISTAS 41
2.2. El TAD pila 42
2.2.1. UNA CALCULADORA RPN CON UNA PILA 44
2.3. EL TAD COLA 45
2.3.1. INTERCALACIÓN DE VECTORES ORDENADOS 45
2.3.1.1. ORDENAMIENTO POR INSERCIÓN 46
2.3.1.2. TIEMPO DE EJECUCIÓN 48
2.3.1.3. PARTICULARIDADES AL ESTAR LAS SECUENCIAS PARES E IMPARES ORDENADAS 49
2.3.1.4. ALGORITMO DE INTERCALACIÓN CON UNA COLA AUXILIAR 50
2.4. EL TAD CORRESPONDENCIA 51
2.4.1. INTERFAZ SIMPLE PARA CORRESPONDENCIAS 55
2.4.2. IMPLEMENTACIÓN DE CORRESPONDENCIAS MEDIANTE CONTENEDORES LINEALES 59
UNIDAD III ÁRBOLES 62
3.1. NOMENCLATURA BÁSICA DE ÁRBOLES 62
3.1.0.0.2. Profundidad de un nodo. Nivel. 65
3.1.0.0.3. NODOS HERMANOS 66
3.2. ORDEN DE LOS NODOS 66
3.2.1. PARTICIONAMIENTO DEL CONJUNTO DE NODOS 68
3.2.2. LISTADO DE LOS NODOS DE UN ÁRBOL 70
3.2.2.1. ORDEN PREVIO 70
3.2.2.2. ORDEN POSTERIOR 71
3.2.2.3. ORDEN POSTERIOR Y LA NOTACIÓN POLACA INVERTIDA 72
3.3. OPERACIONES CON ÁRBOLES 74
3.3.1. ALGORITMOS PARA LISTAR NODOS 74
3.3.2. INSERCIÓN EN ÁRBOLES 76
3.3.2.1. ALGORITMO PARA COPIAR ÁRBOLES 77
3.5. IMPLEMENTACIÓN DE LA INTERFAZ BÁSICA POR PUNTEROS 79
3.5.1. EL TIPO ITERATOR 80
3.5.2. LAS CLASES CELL E ITERATOR_T 82

UNIDAD IV ORDENAMIENTO 89
4.1 INTRODUCCIÓN 89
4.1.1 RELACIONES DE ORDEN DÉBILES 89
4.1.2 SIGNATURA DE LAS RELACIONES DE ORDEN. PREDICADOS BINARIOS 92
4.2 MÉTODOS DE ORDENAMIENTO LENTOS 95
4.2.1 EL MÉTODO DE LA BURBUJA 96
4.2.2 EL MÉTODO DE INSERCIÓN 97
4.2.3 EL MÉTODO DE SELECCIÓN 98
4.3 ORDENAMIENTO INDIRECTO 99
4.3.1 MINIMIZAR LA LLAMADA A FUNCIONES 102
4.4. EL MÉTODO DE ORDENAMIENTO RÁPIDO, QUICK-SORT 102
4.4.1 TIEMPO DE EJECUCIÓN. CASOS EXTREMOS 105
4.4.2 ELECCIÓN DEL PIVOTE 107
CRITERIOS DE EVALUACIÓN: 7
BIBLIOGRAFIA 112

UNIDAD I

DISEÑO Y ANÁLISIS DE ALGORITMOS

Objetivo: El alumno conocerá los métodos para el diseño de algoritmos estructurados iniciando desde el concepto hasta la forma de análisis y estructuración con diseño.

1.1 CONCEPTOS BÁSICOS DE ALGORITMOS

No existe una regla precisa para escribir un programa que resuelva un dado problema práctico. Al menos por ahora escribir programas es en gran medida un arte. Sin embargo, con el tiempo se han desarrollado una variedad de conceptos que ayudan a desarrollar estrategias para resolver problemas y comparar a priori la eficiencia de las mismas.

Por ejemplo, supongamos que queremos resolver el "Problema del Agente Viajero" (TSP, por "Traveling Salesman Problem") el cual consiste en encontrar el orden en que se debe recorrer un cierto número de ciudades (esto es, una serie de puntos en el plano) en forma de tener un recorrido mínimo. Este problema surge en una variedad de aplicaciones prácticas, por ejemplo, encontrar caminos mínimos para recorridos de distribución de productos o resolver el problema de "la vuelta del caballo en el tablero de ajedrez", es decir, encontrar un camino para el caballo que recorra toda la casilla del tablero pasando una sola vez por cada casilla. Existe una estrategia (trivial) que consiste en evaluar todos los caminos posibles. Pero esta estrategia de "búsqueda exhaustiva" tiene un gran defecto, el costo computacional crece de tal manera con el número de ciudades que deja de ser aplicable a partir de una cantidad relativamente pequeña. Otra estrategia "heurística" se basa en buscar un camino que, si bien no es el óptimo (el de menor recorrido sobre todos los posibles) puede ser relativamente bueno en la mayoría de los casos prácticos. Por ejemplo, empezar en una ciudad e ir a la más cercana que no haya sido aún visitada hasta recorrerlas todas.

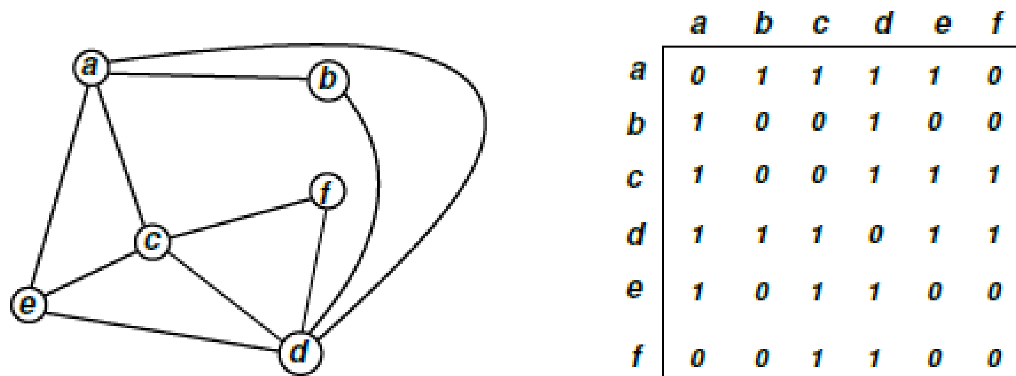
Una forma abstracta de plantear una estrategia es en la forma de un "algoritmo", es decir una secuencia de instrucciones cada una de las cuales representa una tarea bien definida y puede ser llevada a cabo en una cantidad finita de tiempo y con un número finito de recursos computacionales. Un requerimiento fundamental es que el algoritmo debe terminar en un número finito de pasos, de esta manera el mismo puede ser usado como una instrucción en otro algoritmo más complejo. Entonces, comparando diferentes algoritmos para el TSP entre sí, podemos plantear las siguientes preguntas:

- ¿Da el algoritmo la solución óptima?
- Si el algoritmo es iterativo, ¿converge?
- ¿cómo crece el esfuerzo computacional a medida que el número de ciudades crece?

1.1.1 INTRODUCCIÓN BÁSICA A GRAFOS

El problema se puede plantear usando una estructura matemática conocida como "grafo". La base del grafo es un conjunto finito V de puntos llamados "vértices". La estructura del grafo está dada por las conexiones entre los vértices. Si dos vértices están conectados se dibuja una línea que va desde un vértice al otro. Estas conexiones se llaman "aristas" ("edges") del grafo. Los vértices pueden identificarse con un número de 0 am☐1 donde m es el número total de vértices. También es usual representarlos gráficamente con un letra a; b; c; ::: encerrada en un círculo o usar cualquier etiqueta única relativa al problema.

Desde el punto de vista de la teoría de conjuntos un grafo es un subconjunto del conjunto G de pares de vértices. Un par de vértices está en el grafo si existe una arista que los conecta. También puede representarse como una matriz A simétrica de tamaño m _ m con 0's y 1's. Si hay una arista entre el vértice i y el j entonces el elemento Aij es uno, y si no es cero. Además, si existe una arista entre dos vértices i y j entonces decimos que i es "adyacente" a j.



	a	b	c	d	e	f
a	0	1	1	1	1	0
b	1	0	0	1	0	0
c	1	0	0	1	1	1
d	1	1	1	0	1	1
e	1	0	1	1	0	0
f	0	0	1	1	0	0

Figura 1.1: Representación gráfica y matricial del grafo G

En la figura 1.1 vemos un grafo con 6 vértices etiquetados de a a f, representado gráficamente y como una matriz de 0's y 1's. El mismo grafo representado como pares de elementos es

G = ffa; bg; fa; cg; fa; dg; fa; eg; fb; dg; fc; dg; fc; eg; fc; fg; fd; eg; fd; fg; g (1.1)

Para este ejemplo usaremos "grafos no orientados", es decir que si el vértice i está conectado con el j entonces el j está conectado con el i. También existen "grafos orientados" donde las aristas se representan por flechas.

Se puede también agregar un peso (un número real) a los vértices o aristas del grafo. Este peso puede representar, por ejemplo, un costo computacional.

1.1.2. PLANTEO DEL PROBLEMA MEDIANTE GRAFOS

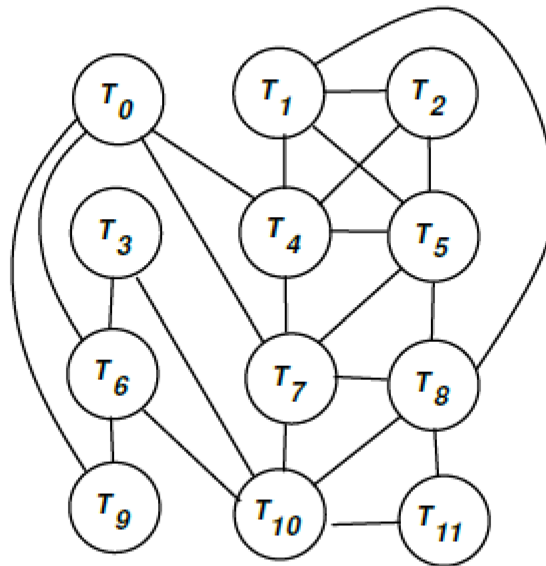


Figura 1.2: Representación del problema mediante un grafo.

Podemos plantear el problema dibujando un grafo donde los vértices corresponden a las tareas y dibujaremos una arista entre dos tareas si son incompatibles entre sí (modifican el mismo objeto). En este caso el grafo resulta ser como muestra la figura 1.2.

La buena noticia es que nuestro problema de particionar el grafo ha sido muy estudiado en la teoría de grafos y se llama el problema de "colorear" el grafo, es decir se representan gráficamente las etapas asignándole colores a los vértices del grafo. La mala noticia es que se ha encontrado que obtener el coloreado ´optimo (es decir el coloreado admisible con la menor cantidad de colores posibles) resulta ser un problema extremadamente costoso en cuanto a tiempo de cálculo. (Se dice que es "NP". Explicaremos esto en la sección §1.3.12.)

El término “colorear grafos” viene de un problema que también se puede poner en términos de colorear grafos y es el de colorear países en un mapa. Consideremos un mapa como el de la figura 1.3. Debemos asignar a cada país un color, de manera que países limítrofes (esto es, que comparten una porción de frontera de medida no nula) tengan diferentes colores y, por supuesto, debemos tratar de usar el mínimo número de colores posibles.

El problema puede ponerse en términos de grafos, poniendo vértices en los países (Cj , j = 1::10) y uniendo con aristas aquellos países que son limítrofes (ver figura 1.4).

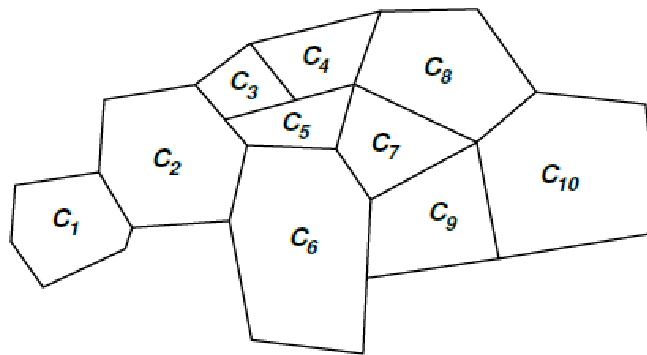


Figura 1.3: Coloración de mapas.

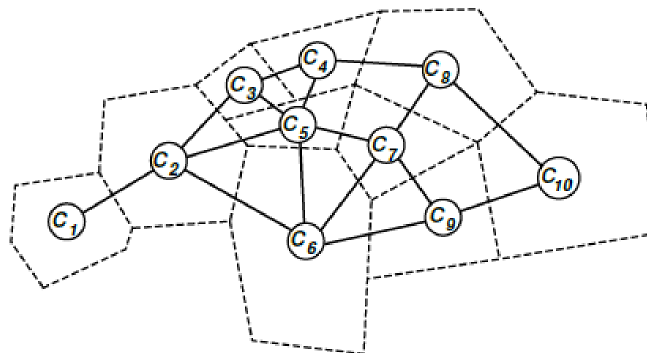


Figura 1.4: Grafo correspondiente al mapa de la figura 1.3.

UDS Mi Universidad

1.1.3. ALGORITMO DE BÚSQUEDA EXHAUSTIVA

Consideremos primero un algoritmo de “búsqueda exhaustiva” es decir, probar si el grafo se puede colorear con 1 solo color (esto solo es posible si no hay ninguna arista en el grafo). Si esto es posible el problema está resuelto (no puede haber coloraciones con menos de un color). Si no es posible entonces generamos todas las coloraciones con 2 colores, para cada una de ellas verificamos si satisface las restricciones o no, es decir si es admisible.

Si lo es, el problema está resuelto: encontramos una coloración admisible con dos colores y ya verificamos que con 1 solo color no es posible. Si no encontramos ninguna coloración admisible de 2 colores entonces probamos con las de 3 colores y así sucesivamente. Si encontramos una coloración de nc colores entonces será optima, ya que previamente verificamos para cada número de colores entre 1 y nc □ 1 que no había ninguna coloración admisible.

Ahora tratando de resolver las respuestas planteadas en la sección §1.1, vemos que el algoritmo propuesto si da la solución óptima. Por otra parte, podemos ver fácilmente que sí termina en un n´umero finito de pasos ya que a lo sumo puede haber nc = m colores, es decir la coloración que consiste en asignar a cada vértice un color diferente es siempre admisible.

1.1.4. GENERACIÓN DE LAS COLORACIONES

En realidad, todavía falta resolver un punto del algoritmo y es como generar todas las coloraciones posibles de nc colores. Además esta parte del algoritmo debe ser ejecutable en un número finito de pasos así que trataremos de evaluar cuantas coloraciones N(nc;m) hay

para m vértices con nc colores. Notemos primero que el procedimiento para generar las coloraciones es independiente de la estructura del grafo (es decir de las aristas), sólo depende de cuantos vértices hay en el grafo y del número de colores que pueden tener las coloraciones.

Figura 1.5: Posibles coloraciones de dos vértices con dos colores

Para nc = 1 es trivial, hay una sola coloración donde todos los vértices tienen el mismo color, es decir N(nc = 1;m) = 1 para cualquier m.

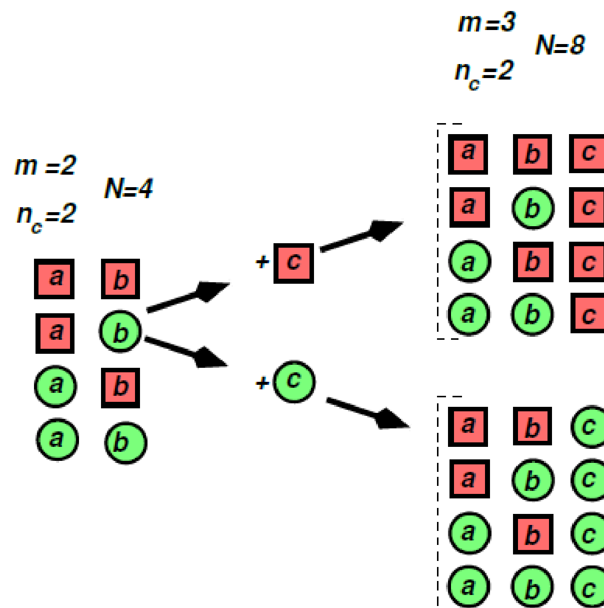


Figura 1.6: Las coloraciones de 3 vértices y dos colores se pueden obtener de las de 2 vértices.

Consideremos ahora las coloraciones de nc = 2 colores, digamos rojo y verde. Si hay un sólo vértice en el grafo, entonces hay solo dos coloraciones posibles: que el vértice sea rojo o verde. Si hay dos vértices, entonces podemos tener 4 coloraciones rojo-rojo, rojo-verde, verde-rojo y verde-verde, es decir N(2; 2) = 4 (ver figura 1.5. Nota: Para que los gráficos con colores sean entendibles en impresión blanco y negro hemos agregado una pequeña letra arriba del vértice indicando el color). Las coloraciones de 3 vértices a; b; c y dos colores las podemos generar a partir de las de 2 vértices, combinando cada una de las 4 coloraciones para los vértices a y b con un posible color para c, ver figura 1.6, de manera que tenemos:

$$N(2,3) = 2\,N(2,2) \tag{1.2}$$

Recursivamente, para cualquier $n_c, m \geq 1$, tenemos que

$$\begin{aligned} N(n_c, m) &= n_c\,N(n_c, m-1) \\ &= n_c^2\,N(n_c, m-2) \\ &\vdots \\ &= n_c^{m-1}\,N(n_c, 1) \end{aligned} \tag{1.3}$$

Pero el número de coloraciones para un sólo vértice con n_c colores es n_c,

de manera que

$$N(n_c, m) = n_c^m \tag{1.4}$$

1.2. TIPOS ABSTRACTOS DE DATOS

Una vez que se ha elegido el algoritmo, la implementación puede hacerse usando las estructuras más simples, comunes en casi todos los lenguajes de programación: escalares, arreglos y matrices. Sin embargo, algunos problemas se pueden plantear en forma más simple

o eficiente en términos de estructuras informáticas más complejas, como listas, pilas, colas, ´arboles, grafos, conjuntos. Por ejemplo, el TSP se plantea naturalmente en términos de un grafo donde los vértices son las ciudades y las aristas los caminos que van de una ciudad a otra. Estas estructuras están incorporadas en muchos lenguajes de programación o bien pueden obtenerse de librerías. El uso de estas estructuras tiene una serie de ventajas

- Se ahorra tiempo de programación ya que no es necesario codificar.
- Estas implementaciones suelen ser eficientes y robustas.
- Se separan dos capas de código bien diferentes, por una parte, el algoritmo que escribe el programador, y por otro las rutinas de acceso a las diferentes estructuras.
- Existen estimaciones bastante uniformes de los tiempos de ejecución de las diferentes operaciones.
- Las funciones asociadas a cada estructura son relativamente independientes del lenguaje o la implementación en particular. Así, una vez que se plantea un algoritmo en términos de operaciones sobre una tal estructura es fácil implementarlo en una variedad de lenguajes con una performance similar.

Un "Tipo Abstracto de Datos" (TAD) es la descripción matemática de un objeto abstracto, definido por las operaciones que actúan sobre el mismo. Cuando usamos una estructura compleja como un conjunto, lista o pila podemos separar tres niveles de abstracción diferente, ejemplificados en la figura 1.12, a saber, las "operaciones abstractas" sobre el TAD, la "interfaz" concreta de una implementación y finalmente la "implementación" de esa interfaz.

Tomemos por ejemplo el TAD CONJUNTO utilizado en el ejemplo de la sección §1.1.1 . Las siguientes son las operaciones abstractas que podemos querer realizar sobre un conjunto

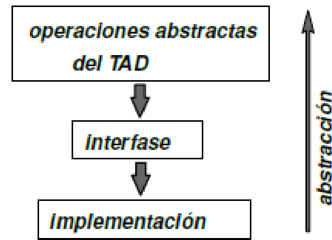


Figura 1.12: Descripción de lo diferentes niveles de abstracción en la definición de un TAD

1.2.1. OPERACIONES ABSTRACTAS Y CARACTERÍSTICAS DEL TAD CONJUNTO

- Contiene elementos, los cuales deben ser diferentes entre sí.
- No existe un orden particular entre los elementos del conjunto.
- Se pueden insertar o eliminar elementos del mismo.
- Dado un elemento se puede preguntar si está dentro del conjunto o no.
- Se pueden hacer las operaciones binarias bien conocidas entre conjuntos a saber, unión, intersección y diferencia.

1.2.2. INTERFAZ DEL TAD CONJUNTO

La "interfaz" es el conjunto de operaciones (con una sintaxis definida) que producen las operaciones del TAD. Por supuesto depende del lenguaje a utilizar, si bien algunas veces es también común que una librería pueda ser usada desde diferentes lenguajes y trate de mantener la interfaz entre esos diferentes lenguajes.

Por ejemplo la implementación del TAD CONJUNTO en la librería STL es (en forma muy simplificada) la siguiente,

```
1. template<class T>
2. class set {
3. public:
4.   class iterator { /* . . . */ };
5.   void insert(T x);
6.   void erase(iterator p);
7.   void erase(T x);
8.   iterator find(T x);
9.   iterator begin();
10.  iterator end();
11. };
```

Código 1.7: *Interfaz de la clase set<> [Archivo: stl-set.cpp]*

Esta interfaz tiene la desventaja de que no provee directamente las operaciones mencionadas previamente, pero encaja perfectamente dentro de la interfaz general de los otros "contenedores" de STL. Recordemos que en STL los elementos de los contenedores, como en este caso **set**, se acceden a través de "iteradores" ("literatos"). En la siguiente descripción **s** es un conjunto, **x** un elemento y **p** un iterador.

- `s.insert(x)` inserta un elemento en el conjunto. Si el elemento ya estaba en el conjunto `s` queda inalterado.
- `p=s.find(x)` devuelve el iterador para el elemento `x`. Si `x` no está en `s` entonces devuelve un iterador especial `end()`. En consecuencia, la expresión lógica para saber si un elemento está en `s` es

```
if(s.find(x)==s.end()) {
  // `x' no esta en `s'
  // ...
}
```

- `s.erase(p)` elimina el elemento que está en el iterador `p` en `s`. `s.erase(x)` elimina el elemento `x` (si está en el conjunto).
- La unión de dos conjuntos, por ejemplo $C = A \cup B$ podemos lograrla insertando los elementos de A y B en C:

```
set A,B,C;
// Pone elementos en A y B
// ...
C.insert(A.begin(),A.end());
C.insert(B.begin(),B.end());
```

Normalmente en C/C++ la interfaz está definida en los headers de las respectivas clases.

- Todas las operaciones binarias con conjuntos se pueden realizar con algoritmos genéricos definidos en el header `algorithm`, usando el adaptador `inserter`:

```
template<class Container, class Iter)
insert_iterator<Container>
inserter(Container& C, Iter i);
```

Típicamente:

- $C = A \cup B$

```
set_union(a.begin(),a.end(),b.begin(),b.end(),
          inserter(c,c.begin()));
```

- $C = A - B$

```
set_difference(a.begin(),a.end(),b.begin(),b.end(),
               inserter(c,c.begin()));
```

- $C = A \cap B$

```
set_intersection(a.begin(),a.end(),b.begin(),b.end(),
                 inserter(c,c.begin()));
```

1.2.3. IMPLEMENTACIÓN DEL TAD CONJUNTO

Finalmente, la "implementación" de estas funciones, es decir el código específico que implementa cada una de las funciones declaradas en la interfaz.

Como regla general podemos decir que un programador que quiere usar una interfaz abstracta como el TAD CONJUNTO, debería tratar de elaborar primero un algoritmo abstracto basándose en las operaciones abstractas sobre el mismo. Luego, al momento de escribir su código debe usar la interfaz específica para traducir su algoritmo abstracto en un código compilable. En el caso del TAD CONJUNTO veremos más adelante que internamente ´este
puede estar implementado de varias formas, a saber, con listas o ´arboles, por ejemplo. En general, el código que escribe no debería depender nunca de los detalles de la implementación particular que está usando.

1.3. TIEMPO DE EJECUCIÓN DE UN PROGRAMA

La eficiencia de un código va en forma inversa con la cantidad de recursos que consume, principalmente tiempo de CPU y memoria. A veces en programación la eficiencia se contrapone con la sencillez y legibilidad de un código. Sin embargo, en ciertas aplicaciones la eficiencia es un factor importante que no podemos dejar de tener en cuenta. Por ejemplo, si escribimos un programa para buscar un nombre en una agenda personal de 200 registros, entonces probablemente la eficiencia no es la mayor preocupación. Pero si escribimos un algoritmo para un motor de búsqueda en un número de entradas > 109, como es común en las aplicaciones para buscadores en Internet hoy en día, entonces la eficiencia probablemente pase a ser un concepto fundamental. Para tal volumen de datos, pasar de un algoritmo O (n log n) a uno O(n1:3) puede ser fatal.

Más importante que saber escribir programas eficientemente es saber cuándo y donde preocuparse por la eficiencia. Antes que nada, un programa está compuesto en general por varios componentes o módulos. No tiene sentido preocuparse por la eficiencia de un dado modulo si este representa un 5% del tiempo total de cálculo. En un tal modulo tal vez sea mejor preocuparse por la robustez y sencillez de programación que por la eficiencia.

El tiempo de ejecución de un programa (para fijar ideas, pensemos por ejemplo en un programa que ordena de menor a mayor una serie de números enteros) depende de una variedad de factores, entre los cuales

- La eficiencia del compilador y las opciones de optimización que pasamos al mismo en tiempo de compilación.
- El tipo de instrucciones y la velocidad del procesador donde se ejecutan las instrucciones compiladas.
- Los datos del programa. En el ejemplo, la cantidad de números y su distribución estadística: ¿son todos iguales?, ¿están ya ordenados o casi ordenados?
- La "complejidad algorítmica" del algoritmo subyacente. En el ejemplo de ordenamiento, el lector ya sabrá que hay algoritmos para los cuales el número de instrucciones crece como n2, donde n es la longitud de la lista a ordenar, mientras que algoritmos como el de "ordenamiento rápido" ("Quicksort") crece como n log n. En el problema de coloración estudiado en §1.1.1 el algoritmo de búsqueda exhaustiva crece como mm, donde m es el número de vértices del grafo contra m3 para el algoritmo ávido descripto en §1.1.8.

En este libro nos concentraremos en los dos últimos puntos de esta lista.

```
1. int search(int l,int *a,int n) {
2.   int j;
3.   for (j=0; j<n; j++)
4.     if (a[j]==l) break;
5.   return j;
6. }
```

Código 1.8: *Rutina simple para buscar un elemento* l *en un arreglo* a[] *de longitud* n *[Archivo: search.cpp]*

En muchos casos, el tiempo de ejecución depende no tanto del conjunto de datos específicos, sino de alguna "medida" del tamaño de los datos. Por ejemplo, sumar un arreglo de n números no depende de los números en sí mismos sino de la longitud n del arreglo. Denotando por T(n) el tiempo de ejecución

$$T(n) = cn \quad (1.16)$$

donde c es una constante que representa el tiempo necesario para sumar un elemento. En otros muchos casos, si bien el tiempo de ejecución sí depende de los datos específicos, en promedio sólo depende del tamaño de los datos. Por ejemplo, si buscamos la ubicación de un elemento l en un arreglo a, simplemente recorriendo el arreglo desde el comienzo hasta el fin (ver código 1.8) hasta encontrar el elemento, entonces el tiempo de ejecución depender 'a fuertemente de la ubicación del elemento dentro del arreglo. El tiempo de ejecución es proporcional a la posición j del elemento dentro del vector tal que aj = l, tomando j = n si el elemento no está. El mejor caso es cuando el elemento esta al principio del arreglo, mientras que el peor es cuando esta al final o cuando no está. Pero "en promedio" (asumiendo que la distribución de elementos es aleatoria) el elemento buscado estará en la zona media del arreglo, de manera que una ecuación como la (1.16) será válida (en promedio). Cuando sea necesario llamaremos Tprom(n) al promedio de los tiempos de ejecución de un dado

algoritmo sobre un “ensamble” de posibles entradas y por Tpeor(n) el peor de todos sobre el ensamble. Entonces, para el caso de buscar la numeración de un arreglo tenemos

$$\begin{aligned} T(n) &= cj \\ T_{\text{peor}}(n) &= cn \\ T_{\text{prom}}(n) &= c\frac{n}{2} \end{aligned} \qquad (1.17)$$

En estas expresiones c puede tomarse como el tiempo necesario para ejecutar una vez el cuerpo del lazo en la rutina **search (...)**. Notar que esta constante c, si la medimos en segundos, puede depender fuertemente de los ítems considerados en los dos primeros puntos de la lista anterior. Por eso, preferimos dejar la constante sin especificar en forma absoluta, es decir que
de alguna forma estamos evaluando el tiempo de ejecución en términos de “unidades de trabajo”, donde una unidad de trabajo c es el tiempo necesario para ejecutar una vez el lazo.

En general, determinar analíticamente el tiempo de ejecución de un algoritmo puede ser una tarea intelectual ardua. Muchas veces, encontrar el Tpeor(n) es una tarea relativamente más fácil. Determinar el Tprom(n) puede a veces ser más fácil y otras veces más difícil.

1.4. CONTEO DE OPERACIONES PARA EL CÁLCULO DEL TIEMPO DE EJECUCIÓN

Comenzaremos por asumir que no hay llamadas recursivas en el programa. (Ni cadenas recursivas, es decir, **sub1()** llama a **sub()** y **sub2()** llama a **sub1()**). Entonces, debe haber rutinas que no llaman a otras rutinas. Comenzaremos por calcular el tiempo de ejecución de ´estas. La regla básica para calcular el tiempo de ejecución de un programa es ir desde los

lazos o construcciones más internas hacia las más externas. Se comienza asignando un costo computacional a las sentencias básicas. A partir de esto se puede calcular el costo de un bloque, sumando los tiempos de cada sentencia. Lo mismo se aplica para funciones y otras construcciones sintácticas que iremos analizando a continuación.

1.4.1. BLOQUES IF

Para evaluar el tiempo de un bloque `if`

```
if(<cond>) {
   <body>
}
```

podemos o bien considerar el peor caso, asumiendo que <*body*> se ejecuta siempre

$$T_{\text{peor}} = T_{\text{cond}} + T_{\text{body}} \tag{1.47}$$

o, en el caso promedio, calcular la probabilidad P de que <*cond*> de verdadero. En ese caso

$$T_{\text{prom}} = T_{\text{cond}} + PT_{\text{body}} \tag{1.48}$$

Notar que T_{cond} no está afectado por P ya que la condición se evalúa siempre. En el caso de que tenga un bloque `else`, entonces

```
if(<cond>) {
   <body-true>
} else {
   <body-false>
}
```

podemos considerar,

$$\begin{aligned} T_{\text{peor}} &= T_{\text{cond}} + \max\left(T_{\text{body-true}}, T_{\text{body-false}}\right) \\ &\leq T_{\text{cond}} + T_{\text{body-true}} + T_{\text{body-false}} \\ T_{\text{prom}} &= T_{\text{cond}} + P\,T_{\text{body-true}} + (1-P)\,T_{\text{body-false}} \end{aligned} \quad (1.49)$$

Las dos cotas para T_{peor} son válidas, la que usa "$\max$" es más precisa.

1.4.2. LAZOS

El caso más simple es cuando el lazo se ejecuta un número fijo de veces, y el cuerpo del lazo tiene un tiempo de ejecución constante,

```
for (i=0; i<N; i++) {
    <body>
}
```

donde $T_{\text{body}} =$ constante. Entonces

$$T = T_{\text{ini}} + N(T_{\text{body}} + T_{\text{inc}} + T_{\text{stop}}) \quad (1.50)$$

donde

- Tini es el tiempo de ejecución de la parte de "inicialización" del lazo, en este caso **i=0**,
- Tinc es el tiempo de ejecución de la parte de "incremento" del contador del lazo, en este caso, **i++** y
- Tstop es el tiempo de ejecución de la parte de "detención" del contador del lazo, en este caso, **i<N**.

En el caso más general, cuando Tbody no es constante, debemos evaluar explícitamente la suma de todas las contribuciones,

$$T = T_{\text{ini}} + \sum_{i=0}^{N-1} (T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}}). \quad (1.51)$$

Algunas veces es difícil calcular una expresión analítica para tales sumas. Si podemos determinar una cierta tasa de crecimiento para todos los términos

$$T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}} = O(f(n)), \quad \text{para todo } i \quad (1.52)$$

entonces,

$$T \leq N \max_{i=1}^{N-1} (T_{\text{body},i} + T_{\text{inc}} + T_{\text{stop}} = O(N f(n)) \quad (1.53)$$

Más difícil aún es el caso en que el número de veces que se ejecuta el lazo no se conoce a priori, por ejemplo un lazo `while` como el siguiente

```
while (<cond>) {
  <body>
}
```

En este caso debemos determinar también el número de veces que se ejecutara el lazo.

Ejemplo 1.5: Calcularemos el tiempo de ejecución del algoritmo de ordenamiento por el "método de la burbuja" ("bubble-sort"). Si bien a esta altura no es necesario saber exactamente cómo funciona el método, daremos una breve descripción del mismo. La función **bubble_sort(...)** toma como argumento un vector de enteros y los ordena de menor a mayor. En la ejecución del lazo de las líneas 6–16 para **j=0** el menor elemento de todos es insertado en **a[0]** mediante una serie de intercambios. A partir de ahí **a[0]** no es tocado más. Para **j=1** el mismo procedimiento es aplicado al rango de índices que va desde **j=1** hasta **j=n-1**, donde **n** es el número de elementos en el vector, de manera que después de la ejecución del lazo para **j=1** el segundo elemento menor es insertado en **a[1]** y así

siguiendo hasta que todos los elementos terminan en la posición que les corresponde en el elemento
ordenado.

```
1.  void bubble_sort(vector<int> &a) {
2.    int n = a.size();
3.    // Lazo externo. En cada ejecucion de este lazo
4.    // el elemento j-esimo menor elemento llega a la
5.    // posicion 'a[j]'
6.    for (int j=0; j<n-1; j++) {
7.      // Lazo interno. Los elementos consecutivos se
8.      // van comparando y eventualmente son intercambiados.
9.      for (int k=n-1; k>j; k--) {
10.       if (a[k-1] > a[k]) {
11.         int tmp = a[k-1];
12.         a[k-1] = a[k];
13.         a[k]=tmp;
14.       }
15.     }
16.   }
```

Código 1.9: Algoritmo de clasificación por el método de la burbuja. [Archivo: bubble.cpp]

1.4.3. SUMA DE POTENCIAS

Sumas como (1.60) ocurren frecuentemente en los algoritmos con lazos anidados. Una forma alternativa de entender esta expresión es aproximar la suma por una integral (pensando a j como una variable continua)

$$\sum_{j=1}^{n} j \approx \int_0^n j \, \mathrm{d}j = \frac{n^2}{2} = O(n^2). \qquad (1.65)$$

De esta forma se puede llegar a expresiones asintóticas para potencias más elevadas

$$\sum_{j=1}^{n} j^2 \approx \int_0^n j^2 \, \mathrm{d}j = \frac{n^3}{3} = O(n^3) \qquad (1.66)$$

y, en general

$$\sum_{j=1}^{n} j^p \approx \int_0^n j^p \, \mathrm{d}j = \frac{n^{p+1}}{p+1} = O(n^{p+1}) \qquad (1.67)$$

1.4.4. LLAMADAS A RUTINAS

Una vez calculados los tiempos de ejecución de rutinas que no llaman a otras rutinas (llamemos al conjunto de tales rutinas S0), podemos calcular el tiempo de ejecución de aquellas rutinas que solo llaman a las rutinas de S0 (llamemos a este conjunto S1), asignando a las línea con llamadas a rutinas de S0 de acuerdo con el tiempo de ejecución previamente calculado, como si fuera una instrucción más del lenguaje.

Por ejemplo, si un programa tiene un árbol de llamadas como el de la figura 1.17, entonces podemos empezar por calcular los tiempos de ejecución de las rutinas **sub4()** y **sub5()**, luego los de **sub1()** y **sub2()**, luego el de **sub3()** y finalmente el de **main()**.

1.4.5. LLAMADAS RECURSIVAS

Si hay llamadas recursivas, entonces el principio anterior no puede aplicarse. Una forma de evaluar el tiempo de ejecución en tales casos es llegar a una expresión recursiva para el tiempo de ejecución mismo.

```
1. int bsearch2(vector<int> &a,int k,int j1, int j2) {
2.   if (j1==j2-1) {
3.     if (a[j1]==k) return j1;
```

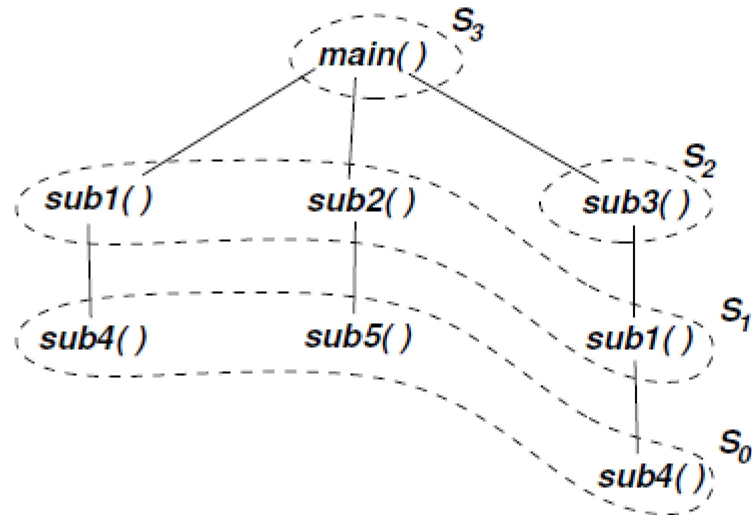


Figura 1.17: Ejemplo de árbol de llamadas de un programa y como calcular los tiempos de ejecución

```
4.     else return j2;
5.   } else {
6.     int p = (j1+j2)/2;
7.     if (k<a[p]) return bsearch2(a,k,j1,p);
8.     else return bsearch2(a,k,p,j2);
9.   }
10. }
11.
12. int bsearch(vector<int> &a,int k) {
13.   int n = a.size();
14.   if (k<a[0]) return 0;
15.   else return bsearch2(a,k,0,n);
16. }
```

Código 1.10: *Algoritmo de búsqueda binaria. [Archivo: bsearch.cpp]*

UNIDAD II

TIPOS DE DATOS ABSTRACTOS FUNDAMENTALES

Objetivo: El alumno conocerá los distintos tipos de datos estructurados de abstracción para realizar las operaciones de pilas, colas, y retrocesos.

La abstracción es un proceso cognitivo humano esencial para la comprensión de fenómenos o situaciones complejas que consiste en la categorización de elementos en grupos o clases de características similares. Cada una de las clases o grupo representa una abstracción, en virtud de la cual se destacan o ignoran determinadas características del grupo. La abstracción, por lo tanto, presenta dos caras complementarias:

- En primer lugar, considera o resalta algunos de los aspectos de los elementos en estudio, en concreto, los aspectos relevantes para el problema o situación que se desea resolver.
- En segundo lugar, ignora el resto de los detalles –no relevantes para la tarea en curso– de los elementos que se abstraen.

La abstracción permite estudiar un sistema complejo a diferentes niveles de detalle, es decir, la abstracción sigue un método jerárquico. El objetivo es poder representar y manejar sistemas complejos de manera más sencilla; para conseguirlo se suele realizar un proceso de abstracción en sentido descendente, lo que implica ir abstrayendo desde niveles más generales a niveles más detallados. El propio ser humano es un ejemplo de este tipo de abstracción: desde el nivel de los seres humanos en sociedades (sociología) al nivel de los constituyentes últimos de nuestro organismo (biología, química), pasando por el estudio de nuestros subsistemas (medicina).

Podemos encontrar otros muchos ejemplos en los que se utiliza la abstracción para tratar de disminuir la complejidad. Algunos de ellos pueden ser:

- Los sistemas de información en una empresa, desde su estructura departamental o geográfica hasta los procesos individuales que se realizan en cada punto.
- Un programa de ordenador es también un sistema complejo, ya que está compuesto de miles de instrucciones y cualquiera de ellas puede producir un error. Para disminuir la complejidad de los programas informáticos, se han llevado a cabo en toda la historia de la programación abstracciones como los lenguajes de alto nivel, que son abstracciones del lenguaje ensamblador o binario, y que permiten al desarrollador trabajar de un modo independiente de una máquina concreta.

El paso de un nivel a otro puede hacerse de un nivel de menor detalle (general) o mayor granularidad a otro de mayor detalle o granularidad más fina y viceversa, según el enfoque sea descendente o ascendente.
En resumen, podemos decir que la abstracción es un arma para la comprensión y resolución de sistemas o problemas complejos. Así pues, la evolución en la informática está basada en caminar hacia un grado creciente de abstracción. De hecho, los paradigmas de programación más modernos abstraen al programador de la secuencia concreta de instrucciones.

Abstracción funcional La abstracción funcional surge de la simple idea de crear procedimientos y funciones e invocarlos en diferentes partes del programa mediante un nombre. La parte que se ignora ("la información que se oculta") en este proceso es la de cómo el procedimiento o función realiza su tarea. La parte con la que nos quedamos es la

signatura o interfaz: los parámetros de entrada y salida (y sus tipos), y la descripción de la tarea que realiza. Las ventajas de la abstracción funcional son las siguientes:

- Generalización del concepto de operador: Gracias a esta abstracción podemos definir operadores adicionales a los del lenguaje (por ejemplo, una función factorial), incluyendo la posibilidad de que dichos operadores no trabajen sobre tipos básicos del lenguaje (por ejemplo, determinantes o multiplicaciones de matrices).
- Encapsulación y ocultamiento: Haciendo que una secuencia de acciones esté oculta y sólo se haga visible desde el exterior su resultado global que se entrega a través de parámetros de salida bien definidos.

Abstracción de datos: La primera aplicación de la abstracción a los datos está en los propios tipos de datos básicos de los lenguajes de programación. Un tipo boolean en un lenguaje como Pascal tiene una semántica bien definida, que abstrae la representación concreta en una máquina o sistema operativo determinado (que puede ser un bit o un valor entero). Así, un programa en un lenguaje de alto nivel puede ejecutarse en máquinas de arquitecturas diferentes con sólo recompilarlo. Las operaciones que pueden realizarse sobre un tipo boolean son las operaciones lógicas habituales, y, aunque puede que internamente sea un entero, el lenguaje no nos permite manipularlo como tal (por ejemplo, manipular un bit o sumar). En este sentido, lenguajes como C permiten "violar" la abstracción (y por tanto la semántica) de los tipos de datos.

Un paso más adelante en la abstracción de datos es el de los tipos definidos por el programador, como los enumerados, que permiten definir explícitamente el dominio de valores y dar un nombre a cada uno de ellos, aunque internamente el compilador los maneje como un subrango de enteros. El problema es que se puede restringir el dominio de valores, pero no definir nuevas operaciones ni restringir las existentes, ya que estas vienen predefinidas por el lenguaje de programación.

Los tipos estructurados son un paso adicional que ayuda a conseguir la genericidad, es decir, se define un tipo como un agregado de instancias de otro tipo. Este tipo subordinado se suele denominar tipo parámetro, y al agregado, tipo parametrizado:

```
t = vector [limInferior..limSuperior] de t2
```

donde `t2` es el tipo parámetro y `t` el tipo parametrizado.

El problema de los tipos estructurados1 es que el lenguaje no permite que el programador establezca cuáles son las operaciones válidas sobre el tipo. Por ejemplo, dado un tipo estructurado registro "fecha", nada nos impide realizar asignaciones incorrectas desde el punto de vista de la lógica de una fecha, ni realizar operaciones sin sentido sobre este tipo, como las que se muestran a continuación sobre f1 y f2, ambas variables del tipo "fecha":

```
f1.Dia= 30
f1.Mes= 2
f2.Dia= 5 * f1.Mes
```

2.1. EL TAD LISTA

Las listas constituyen una de las estructuras lineales más flexibles, porque pueden crecer y acortarse según se requiera, insertando o suprimiendo elementos tanto en los extremos como en cualquier otra posición de la lista. Por supuesto esto también puede hacerse con vectores, pero en las implementaciones más comunes estas operaciones son O(n) para los vectores, mientras que son O(1) para las listas. El poder de las listas es tal que la familia de lenguajes derivados del Lisp, que hoy en día cuenta con el Lisp, Common Lisp y Scheme, entre otros, el lenguaje mismo está basado en la lista ("Lisp" viene de "list processing").

2.1.1. DESCRIPCIÓN MATEMÁTICA DE LAS LISTAS

Desde el punto de vista abstracto, una lista es una secuencia de cero o más elementos de un tipo determinado, que en general llamaremos elem_t, por ejemplo, int o double. A menudo representamos una lista en forma impresa como una sucesión de elementos entre paréntesis, separados por comas

$$L = (a_0, a_1, \ldots, a_{n-1}) \quad (2.1)$$

donde $n \geq 0$ es el número de elementos de la lista y cada a_i es de tipo elem_t. Si $n \geq 1$ entonces decimos que a_0 es el primer elemento y a_{n-1} es el último elemento de la lista. Si $n = 0$ decimos que la lista "está vacía". Se dice que n es la "longitud" de la lista.

Una propiedad importante de la lista es que sus elementos están ordenados en forma lineal, es decir, para cada elemento a_i existe un sucesor a_{i+1} (si $i < n - 1$) y un predecesor a_{i-1} (si $i > 0$). Este orden es parte de la lista, es decir, dos listas son iguales si tienen los mismos elementos y en el mismo orden. Por ejemplo, las siguientes listas son distintas

(1, 3, 7, 4) 6= (3, 7, 4, 1) (2.2)

Mientras que, en el caso de conjuntos, éstos serían iguales. Otra diferencia con los conjuntos es que puede haber elementos repetidos en una lista, mientras que en un conjunto no.

Decimos que el elemento a_i "está en la posición i". También introducimos la noción de una posición ficticia n que está fuera de la lista. A las posiciones en el rango $0 \leq i \leq n - 1$ las llamamos "de referenciales" ya que pertenecen a un objeto real, y por lo tanto podemos obtener una referencia a ese objeto. Notar que, a medida que se vayan insertando o

eliminando elementos de la lista la posición ficticia n va variando, de manera que convendrá tener un método de la clase end() que retorne esta posición.

2.1.2. OPERACIONES ABSTRACTAS SOBRE LISTAS

Consideremos una operación típica sobre las listas que consiste en eliminar todos los elementos duplicados de la misma. El algoritmo más simple consiste en un doble lazo, en el cual el lazo externo sobre i va desde el comienzo hasta el último elemento de la lista. Para cada elemento i el lazo interno recorre desde i + 1 hasta el último elemento, eliminado los elementos iguales a i. Notar que no hace falta revisar los elementos anteriores a i (es decir, los elementos j con j < i), ya que, por construcción, todos los elementos de 0 hasta i son distintos.

Este problema sugiere las siguientes operaciones abstractas

- Dada una posición i, *"insertar"* el elemento x en esa posición, por ejemplo

$$\begin{aligned} &L = (1, 3, 7) \\ &\text{inserta } 5 \text{ en la posición } 2 \\ &\rightarrow L = (1, 3, 5, 7) \end{aligned} \tag{2.3}$$

  Notar que el elemento 7, que estaba en la posición 2, se desplaza hacia el fondo, y termina en la posición 3. Notar que es válido insertar en cualquier posición dereferenciable, o en la posición ficticia `end()`

- Dada una posición i, *"suprimir"* el elemento que se encuentra en la misma. Por ejemplo,

$$\begin{aligned} &L = (1, 3, 5, 7) \\ &\text{suprime elemento en la posición } 2 \\ &\rightarrow L = (1, 3, 7) \end{aligned} \tag{2.4}$$

Notar que esta operación es, en cierta forma, la inversa de insertar. Si, como en el ejemplo anterior, insertamos un elemento en la posición i y después suprimimos en esa misma posición, entonces la lista queda inalterada. Notar que sólo es válido suprimir en las posiciones dereferenciables.

Si representáramos las posiciones como enteros, entonces avanzar la posición podría efectuarse con la sencilla operación de enteros i ← i + 1, pero es deseable pensar en las posiciones como entidades abstractas, no necesariamente enteros y por lo tanto para las cuales no necesariamente es válido hacer operaciones de enteros. Esto lo sugiere la experiencia previa de cursos básicos de programación donde se ha visto que las listas se representan por celdas encadenadas por punteros. En estos casos, puede ser deseable representar a las posiciones como punteros a las celdas. De manera que asumiremos que las posiciones son objetos abstractos. Consideramos entonces las operaciones abstractas:

- Acceder al elemento en la posición p, tanto para modificar el valor ($a_p \leftarrow x$) como para acceder al valor ($x \leftarrow a_p$).
- Avanzar una posición, es decir dada una posición p correspondiente al elemento a_i, retornar la posición q correspondiente al elemento a_{i+1}. (Como mencionamos previamente, no es necesariamente $q = p + 1$, o más aún, pueden no estar definidas estas operaciones aritméticas sobre las posiciones p y q.)
- Retornar la primera posición de la lista, es decir la correspondiente al elemento a_0.
- Retornar la posición ficticia al final de la lista, es decir la correspondiente a n.

2.2. El TAD pila

Básicamente es una lista en la cual todas las operaciones de inserción y borrado se producen en uno de los extremos de la lista. Un ejemplo gráfico es una pila de libros en un cajón. A medida que vamos recibiendo más libros los ubicamos en la parte superior. En todo momento tenemos acceso sólo al libro que se encuentra sobre el "tope" de la pila. Si queremos acceder a algún libro que se encuentra más abajo (digamos en la quinta posición

desde el tope) debemos sacar los primeros cuatro libros y ponerlos en algún lugar para poder acceder al mismo. La Pila es el típico ejemplo de la estructura tipo "LIFO" (por "Last In First Out", es decir "el último en entrar es el primero en salir").

La pila es un subtipo de la lista, es decir podemos definir todas las operaciones abstractas sobre pila en función de las operaciones sobre lista. Esto motiva la idea de usar "adaptadores" es decir capas de código (en la forma de templates de C++) para adaptar cualquiera de las posibles clases de lista (por arreglo, punteros o cursores) a una pila.

Ejemplo 2.3: Consigna: Escribir un programa para calcular expresiones aritméticas complejas con números en doble precisión usando "notación polaca invertida" (RPN, por "reverse polish notation"). Solución: En las calculadoras con RPN una operación como 2+3 se introduce en la forma 2 3 +. Esto lo denotamos así

$$\mathrm{rpn}[2+3] = 2, 3, + \tag{2.7}$$

donde hemos separados los elementos a ingresar en la calculadora por comas. En general, para cualquier operador binario (como +, -, * o /) tenemos

$$\mathrm{rpn}[(a)\,\theta\,(b)] = \mathrm{rpn}(a),\ \mathrm{rpn}(b),\ \theta \tag{2.8}$$

donde a, b son los operandos y θ el operador. Hemos introducido paréntesis alrededor de los operandos a y b ya que (eventualmente) estos pueden ser también expresiones, de manera que, por ejemplo la expresión (2 + 3) ∗ (4 − 5) puede escribirse como

$$\begin{aligned} \mathrm{rpn}[(2+3)*(4-5)] &= \mathrm{rpn}[2+3]\,,\mathrm{rpn}[4-5]\,,* \\ &= 2\,,3\,,+\,,4\,,5\,,-\,,* \end{aligned} \tag{2.9}$$

La ventaja de una tal calculadora es que no hace falta ingresar paréntesis, con el inconveniente de que el usuario debe convertir mentalmente la expresión a RPN.

2.2.1. UNA CALCULADORA RPN CON UNA PILA

La forma de implementar una calculadora RPN es usando una pila. A medida que el usuario entra operandos y operadores se aplican las siguientes reglas

- Si el usuario ingresó un operando, entonces simplemente se almacena en la pila.
- Si ingresó un operador θ se extraen dos operandos del tope de la pila, digamos t el tope de la pila y u el elemento siguiente, se aplica el operador a los dos operandos (en forma invertida) es decir u θ t y se almacena el resultado en el tope de la pila.

Por ejemplo, para la expresión (2.9) tenemos un seguimiento como el mostrado en la tabla 2.3. que es el resultado correcto. El algoritmo puede extenderse fácilmente a funciones con un número arbitrario de variables (como exp(), cos() ...), sólo que en ese caso se extrae el número de elementos apropiados de la pila, se le aplica el valor y el resultado es introducido en la misma. De nuevo, notar que en general debe invertirse el orden de los argumentos al sacarlos de la pila. Por ejemplo la función rem(a,b) (resto) retorna el resto de dividir b en a. Si analizamos la expresión mod (5,3) (que debe retornar 2) notamos que al momento de aplicar la función mod, tenemos en la pila los elementos 3,5 (el top primero), de manera que la función debe aplicarse a los elementos en orden invertido.

Ingresa	Tipo	Pila
2	operando	2
3	operando	3,2
+	operador	5
4	operando	4,5
5	operando	5,4,5
-	operador	-1,5
*	operador	-5

Tabla 2.3: Seguimiento del funcionamiento de la calculadora RPN usando una pila. (Los elementos en la pila son enumerados empezando por el tope.)

2.3. EL TAD COLA

Por contraposición con la pila, la cola es un contenedor de tipo "FIFO" (por "First In First Out", el primero en entrar es el primero en salir). El ejemplo clásico es la cola de la caja en el supermercado. La cola es un objeto muchas veces usado como buffer o pulmón, es decir un contenedor donde almacenar una serie de objetos que deben ser procesados, manteniendo el orden en el que ingresaron. La cola es también, como la pila, un subtipo de la lista llama también a ser implementado como un adaptador.

2.3.1. INTERCALACIÓN DE VECTORES ORDENADOS

Ejemplo 2.4: Un problema que normalmente surge dentro de los algoritmos de ordenamiento es el intercalamiento de contenedores ordenados. Por ejemplo, si tenemos dos listas ordenadas L1 y L2, el proceso de intercalamiento consiste en generar una nueva lista L ordenada que contiene los elementos en L1 y L2 de tal forma que L está ordenada, en la forma lo más eficiente posible. Con listas es fácil llegar a un algoritmo O(n) simplemente tomando de las primeras posiciones de ambas listas el menor de los elementos e insertándolo en L (ver secciones §4.3.0.2 y §5.6). Además, este algoritmo no requiere memoria adicional, es decir, no necesita alocar nuevas celdas ya que los elementos son agregados a L a medida que se eliminan de L1 y L2 (Cuando manipula contenedores sin requerir memoria adicional se dice que es "in place" ("en el lugar")). Para vectores el problema podría plantearse así

Consigna: Sea a un arreglo de longitud par, tal que las posiciones pares como las impares están ordenadas entre sí, es decir

$$a_0 \leq a_2 \leq \cdots \leq a_{n-2}$$
$$a_1 \leq a_3 \leq \cdots \leq a_{n-1} \qquad (2.10)$$

Escribir un algoritmo que ordena los elementos de a.

2.3.1.1. ORDENAMIENTO POR INSERCIÓN

Solución: Consideremos por ejemplo que el arreglo contiene los siguientes elementos.

$$a = 10 \ 1 \ 12 \ 3 \ 14 \ 5 \ 16 \ 7 \ 18 \ 9 \ 20 \ 51 \ 22 \ 53 \ 24 \ 55 \ 26 \ 57 \ 28 \ 59 \qquad (2.11)$$

Verificamos que los elementos en las posiciones pares (10 12 14 16 . . .) están ordenados entre sí, como también los que están en las posiciones impares (1 3 5 7 . . .). Consideremos primero el algoritmo de "ordenamiento por inserción" (ver código 2.19, los algoritmos de ordenamiento serán estudiados en más detalle en un capítulo posterior).

```
1 void inssort(vector<int> &a) {
2   int n=a.size();
3   for (int j=1; j<n; j++) {
4     int x = a[j];
5     int k = j;
6     while (--k>=0 && x<a[k]) a[k+1] = a[k];
7     a[k+1] = x;
8   }
9 }
```

Código 2.19: *Algoritmo de ordenamiento por inserción. [Archivo: inssort.cpp]*

El cursor j va avanzando desde el comienzo del vector hasta el final. Después de ejecutar el cuerpo del lazo sobre j el rango de elementos [0, j] queda ordenado. (Recordar que [a, b) significa los elementos que están en las posiciones entre a y b incluyendo a a y excluyendo a

b). Al ejecutar el lazo para un dado j, los elementos a0, . . . , aj−1 están ordenados, de manera que basta con “insertar” (de ahí el nombre del método) el elemento aj en su posición correspondiente. En el lazo sobre k, todos los elementos mayores que aj son desplazados una posición hacia el fondo y el elemento es insertado en alguna posición p <= j, donde corresponde.

$$
\begin{array}{cccccccccccccccccccc}
10 & \boxed{1}^{j} & 12 & 3 & 14 & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
\boxed{1}^{p} & 10 & 12 & 3 & 14 & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 10 & \boxed{12}^{j} & 3 & 14 & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 10 & \boxed{12}^{p} & 3 & 14 & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 10 & 12 & \boxed{3}^{j} & 14 & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & \boxed{3}^{p} & 10 & 12 & 14 & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 10 & 12 & \boxed{14}^{j} & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 10 & 12 & \boxed{14}^{p} & 5 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 10 & 12 & 14 & \boxed{5}^{j} & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & \boxed{5}^{p} & 10 & 12 & 14 & 16 & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 5 & 10 & 12 & 14 & \boxed{16}^{j} & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 5 & 10 & 12 & 14 & \boxed{16}^{p} & 7 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 5 & 10 & 12 & 14 & 16 & \boxed{7}^{j} & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 5 & \boxed{7}^{p} & 10 & 12 & 14 & 16 & 18 & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 5 & 7 & 10 & 12 & 14 & 16 & \boxed{18}^{j} & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 5 & 7 & 10 & 12 & 14 & 16 & \boxed{18}^{p} & 9 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 5 & 7 & 10 & 12 & 14 & 16 & 18 & \boxed{9}^{j} & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59 \\
1 & 3 & 5 & 7 & \boxed{9}^{p} & 10 & 12 & 14 & 16 & 18 & 20 & 51 & 22 & 53 & 24 & 55 & 26 & 57 & 28 & 59
\end{array}
\tag{2.12}
$$

En (2.12) vemos un seguimiento de algunas de las operaciones de inserción. Cada par de líneas corresponde a la ejecución para un j dado, la primera línea muestra el estado del vector antes de ejecutar el cuerpo del lazo y la segunda línea muestra el resultado de la operación. En ambos casos se indica con una caja el elemento que es movido desde la posición j a la p. Por ejemplo, para j = 9 el elemento aj = 9 debe viajar hasta la posición p = 4,

lo cual involucra desplazar todos los elementos que previamente estaban en el rango [4, 9) una posición hacia arriba en el vector. Este algoritmo funciona, por supuesto, para cualquier vector, independientemente de si las posiciones pares e impares están ordenadas entre si, como estamos asumiendo en este ejemplo.

2.3.1.2. TIEMPO DE EJECUCIÓN

Consideremos ahora el tiempo de ejecución de este algoritmo. El lazo sobre j se ejecuta n − 1 veces, y el lazo interno se ejecuta, en el peor caso j − 1 veces, con lo cual el costo del algoritmo es, en el peor caso

$$T_{\text{peor}}(n) = \sum_{j=1}^{n-1} (j-1) = O(n^2) \tag{2.13}$$

En el mejor caso, el lazo interno no se ejecuta ninguna vez, de manera que sólo cuenta el lazo externo que es O(n). En general, el tiempo de ejecución del algoritmo dependerá de cuantas posiciones deben ser desplazadas (es decir p − j) para cada j

$$T(n) = \sum_{j=1}^{n-1} (p-j) \tag{2.14}$$

o, tomando promedios

$$T_{\text{prom}}(n) = \sum_{j=1}^{n-1} d_j \tag{2.15}$$

donde d_j es el tamaño promedio del rango que se desplaza.

El promedio debe ser tomado sobre un cierto conjunto de posibles vectores. Cuando tomamos vectores completamente desordenados, se puede ver fácilmente que dj = j/2 ya que el elemento aj no guarda ninguna relación con respecto a los elementos precedentes y en promedio irá a parar a la mitad del rango ordenado, es decir p = j/2 y entonces j − p = j/2, de manera que

$$T_{\text{prom}}(n) = \sum_{j=1}^{n-1} d_j = \sum_{j=1}^{n-1} \frac{j}{2} = O(n^2) \tag{2.16}$$

2.3.1.3. PARTICULARIDADES AL ESTAR LAS SECUENCIAS PARES E IMPARES ORDENADAS

Como la intercalación de listas ordenadas es O(n) surge la incógnita de si el algoritmo para arreglos puede ser mejorado. Al estar las posiciones pares e impares ordenadas entre sí puede ocurrir que en promedio el desplazamiento sea menor, de hecho, generando vectores en forma aleatoria, pero tales que sus posiciones pares e impares estén ordenada se llega a la conclusión que el desplazamiento promedio es O(√ n), de manera que el algoritmo resulta ser O(n 3/2). Esto representa una gran ventaja contra el O(n 2) del algoritmo de ordenamiento original.

De todas formas, podemos mejorar más aún esto si tenemos en cuenta que las subsecuencias pares e impares están ordenadas. Por ejemplo consideremos lo que ocurre en el seguimiento (2.12) al mover los elementos 18 y 9 que originalmente estaban en las posiciones q = 8 y q+1 = 9. Como vemos, los elementos en las posiciones 0 a q −1 = 7 están ordenados. Notar que el máximo del rango ya ordenado [0, q) es menor que el máximo de estos dos nuevos elementos aq y aq+a, ya que todos los elementos en [0, q) provienen de elementos en las subsecuencias que estaban antes de aq y aq+1.

$$\max_{j=0}^{q-1} a_j < \max(a_q, a_{q+1}) \tag{2.17}$$

por lo tanto después de insertar los dos nuevos elementos, el mayor (que en este caso es 18) quedará en la posición q + 1. El menor (min(aq, aq+1) = 9) viaja una cierta distancia, hasta la posición p = 4. Notar que, por un razonamiento similar, todos los elementos en las posiciones [q + 2, n) deben ser mayores que min(aq, aq+1), de manera que los elementos en [0, p) no se moverán a partir de esta inserción.

2.3.1.4. ALGORITMO DE INTERCALACIÓN CON UNA COLA AUXILIAR

```
1 void merge(vector<int> &a) {
2   int n = a.size();
3   // C = cola vacia . . .
4   int p=0, q=0, minr, maxr;
5   while (q<n) {
6     // minr = min(a_q,a_{q+1}, maxr = max(a_q,a_{q+1}
7     if (a[q]<=a[q+1]) {
8       minr = a[q];
9       maxr = a[q+1];
10    } else {
11      maxr = a[q];
12      minr = a[q+1];
13    }
14    // Apendizar todos los elementos del frente de la cola menores que
15    // min(a_q,a_{q+1}) al rango [0,p), actualizando eventualmente
16    while ( /* C no esta vacia. . . */) {
17      x = /* primer elemento de C . . . */;
18      if (x>minr) break;
19      a[p++] = x;
20      // Saca primer elemento de C . . .
21    }
22    a[p++] = minr;
23    a[p++] = minr;
24    // Apendizar 'maxr' al rango [0,p) . . .
25    q += 2;
26  }
27  // Apendizar todos los elementos en C menores que
28  // min(a_q,a_{q+1}) al rango [0,p)
29  // . . .
30 }
```

Código 2.20: *Algoritmo de intercalación con una cola auxiliar [Archivo: mrgarray1.cpp]*

El algoritmo se muestra en el código 2.20, manteniendo el rango [p, q) en una cola auxiliar C. En (2.18) vemos el seguimiento correspondiente. Los elementos en la cola C son mostrados

encerrados en una caja, en el rango [p, q). Notar que si bien, el número de elementos en la cola entra exactamente en ese rango, en la implementación el estado los elementos en ese rango es irrelevante y los elementos están en una cola auxiliar.

```
1 [10]^C 12  3 14  5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
1  3 [10 12]^C 14  5 16  7 18  9 20 51 22 53 24 55 26 57 28 59
1  3  5 [10 12 14]^C 16  7 18  9 20 51 22 53 24 55 26 57 28 59
1  3  5  7 [10 12 14 16]^C 18  9 20 51 22 53 24 55 26 57 28 59
1  3  5  7  9 [10 12 14 16 18]^C 20 51 22 53 24 55 26 57 28 59      (2.18)
1  3  5  7  9 10 12 14 16 18 20 [51]^C 22 53 24 55 26 57 28 59
1  3  5  7  9 10 12 14 16 18 20 22 [51 53]^C 24 55 26 57 28 59
1  3  5  7  9 10 12 14 16 18 20 22 24 [51 53 55]^C 26 57 28 59
1  3  5  7  9 10 12 14 16 18 20 22 24 26 [51 53 55 57]^C 28 59
1  3  5  7  9 10 12 14 16 18 20 22 24 26 28 [51 53 55 57 59]^C
```

Consideremos por ejemplo el paso en el cual se procesan los elementos aq = 20 y aq+1 = 51 en las posiciones q = 10 y q + 1 = 11. En ese momento la cola contiene los elementos 10,12,14,16 y 18. Como son todos menores que min(aq, aq+1) = 20 apendizamos todos al rango [0, p = 5) de manera que queda p = 10. Se apendiza también el 20, con lo cual queda p = 11 y finalmente se apendiza max(aq, aq+1) = 51 a la cola. Como en ese momento la cola esta vacía, depués de la inserción queda en la cola solo el 51.

2.4. EL TAD CORRESPONDENCIA

La "correspondencia" o "memoria asociativa" es un contenedor que almacena la relación entre elementos de un cierto conjunto universal D llamado el "dominio" con elementos de otro conjunto universal llamado el "contradominio" o "rango". Por ejemplo, la

correspondencia M que va del dominio de los números enteros en sí mismo y transforma un número j en su cuadrado j 2 puede representarse como se muestra en la figura 2.15. Una restricción es que un dado elemento del dominio o bien no debe tener asignado ningún

	insort	merge
$T_{\mathrm{peor}}(n)$	$O(n^2)$	$O(n)$
$T_{\mathrm{prom}}(n)$	$O(n^{3/2})$	$O(n)$
$T_{\mathrm{mejor}}(n)$	$O(n)$	$O(n)$
$M_{\mathrm{peor}}(n)$	$O(n)$	$O(n)$
$M_{\mathrm{prom}}(n)$	$O(n)$	$O(\sqrt{n})$
$M_{\mathrm{mejor}}(n)$	$O(n)$	$O(1)$

Tabla 2.4: Tiempo de ejecución para la intercalación de vectores ordenados. T es tiempo de ejecución, M memoria *adicional* requerida.

elemento del contradominio o bien debe tener asignado uno solo. Por otra parte, puede ocurrir que a varios elementos del dominio se les asigne un solo elemento del contradominio. En el ejemplo de la figura a los elementos 3 y -3 del dominio les es asignado el mismo elemento 9 del contradominio. A veces también se usa el término "clave" ("key") (un poco por analogía con las bases de datos) para referirse a un valor del dominio y "valor" para referirse a los elementos del contradominio.

Las correspondencias son representadas en general guardando internamente los pares de valores y poseen algún algoritmo para asignar valores a claves, en forma análoga a como funcionan las bases de datos. Por eso, en el caso del ejemplo previo j → j 2 , es mucho más eficiente representar la correspondencia como una función, ya que es mucho más rápido y no es necesario almacenar todos los valores. Notar que, para las representaciones más usuales de enteros, por ejemplo, con 32 bits, harían falta varios gigabytes de RAM. El uso de un contenedor tipo correspondencia es útil justamente cuando no es posible calcular el elemento del contradominio a partir del elemento del dominio. Por ejemplo, un tal caso es

una correspondencia entre el número de documento de una persona y su nombre. Es imposible de “calcular” el nombre a partir del número de documento, necesariamente hay que almacenarlo internamente en forma de pares de valores.

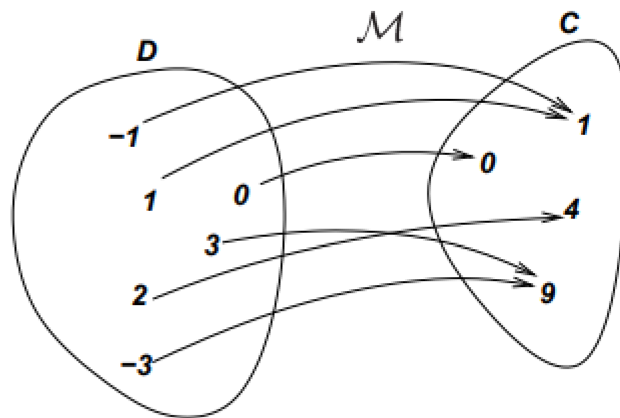


Figura 2.15: Correspodencia entre números enteros $j \to j^2$

Ejemplo 2.5: Consigna: Escribir un programa que memoriza para cada documento de identidad el sueldo de un empleado. Se van ingresando números de documento, si el documento ya tiene un sueldo asignado, entonces esto se reporta por consola, sino el usuario debe entrar un sueldo el cual es asignado a ese número de documento en la tabla. Una posible interacción con el programa puede ser como sigue

```
1 [mstorti@spider aedsrc]$ payroll
2 Ingrese nro. documento > 14203323
3 Ingrese salario mensual: 2000
4 Ingrese nro. documento > 13324435
5 Ingrese salario mensual: 3000
6 Ingrese nro. documento > 13323421
7 Ingrese salario mensual: 2500
8 Ingrese nro. documento > 14203323
9 Doc: 14203323, salario: 2000
10 Ingrese nro. documento > 13323421
11 Doc: 13323421, salario: 2500
12 Ingrese nro. documento > 13242323
13 Ingrese salario mensual: 5000
14 Ingrese nro. documento > 0
15 No se ingresan mas sueldos...
16 [mstorti@spider aedsrc]$
```

Solución: Un posible seudocódigo puede observarse en el código 2.23. El programa entra en un lazo infinito en el cual se ingresa el número de documento y se detiene cuando se ingresa un documento nulo. Reconocemos las siguientes operaciones abstractas necesarias para manipular correspondencias

- Consultar la correspondencia para saber si una dada clave tiene un valor asignado.
- Asignar un valor a una clave.
- Recuperar el valor asignado a una clave

```
// declarar 'tabla_sueldos' como 'map' . . .
while(1) {
  cout << "Ingrese nro. documento > ";
  int doc;
  double sueldo;
  cin >> doc;
  if (!doc) break;
  if (/* No tiene 'doc' sueldo asignado?. . . */) {
    cout << "Ingrese sueldo mensual: ";
    cin >> sueldo;
    // Asignar 'doc -> sueldo'
    // . . .
  } else {
    // Reportar el valor almacenado
    // en 'tabla_sueldos'
    // . . .
    cout << "Doc: " << doc << ", sueldo: "
         << sueldo << endl;
  }
}
cout << "No se ingresan mas sueldos. . ." << endl;
```

Código 2.23: Seudocódigo para construir una tabla que representa la correspondencia número de documento → sueldo. [Archivo: payroll4.cpp]

2.4.1. INTERFAZ SIMPLE PARA CORRESPONDENCIAS

```
1 class iterator_t {/* . . . */};
2
3 class map {
4  private:
5   // . . .
6  public:
7   iterator_t find(domain_t key);
8   iterator_t insert(domain_t key,range_t val);
9   range_t& retrieve(domain_t key);
10  void erase(iterator_t p);
11  int erase(domain_t key);
12  domain_t key(iterator_t p);
13  range_t& value(iterator_t p);
14  iterator_t begin();
15  iterator_t next(iterator_t p);
16  iterator_t end();
17  void clear();
18  void print();
19 };
```

Código 2.24: *Interfaz básica para correspondencias. [Archivo: mapbas.h]*

En el código 2.24 vemos una interfaz básica posible para correspondencias. Está basada en la interfaz STL pero, por simplicidad evitamos el uso de clases anidadas para el correspondiente iterator y también evitamos el uso de templates y sobrecarga de operadores. Primero se deben definir (probablemente via typedef's) los tipos que corresponden al dominINIO (domain_t) y al contradominio (range_t). Una clase iterator (cuyos detalles son irrelevantes para la interfaz) representa las posiciones en la correspondencia. Sin embargo, considerar que, en contraposición con las listas y a semejanza de los conjuntos, no hay un orden definido entre los pares de la correspondencia. Por otra parte, en la correspondencia el iterator itera sobre los pares de valores que representan la correspondencia.

En lo que sigue M es una correspondencia, p es un iterator, k una clave, val un elemento del contradominio (tipo range_t). Los métodos de la clase son

- p = find(k): Dada una clave k devuelve un iterator al par correspondiente (si existe debe ser único). Si k no tiene asignado ningún valor, entonces devuelve end().
- p = insert(k,val): asigna a k el valor val. Si k ya tenía asignado un valor, entonces este nuevo valor reemplaza a aquel en la asignación. Si k no tenía asignado ningún valor entonces la nueva asignación es definida. Retorna un iterator al par.
- val = retrieve(k): Recupera el valor asignado a k. Si k no tiene ningún valor asignado, entonces inserta una asignación de k al valor creado por defecto para el tipo range_t (es decir el que retorna el constructor range_t()). Esto es muy importante y muchas veces es fuente de error. (Muchos esperan que retrieve() de un error en ese caso.) Si queremos recuperar en val el valor asignado a k sin insertar accidentalmente una asignación en el caso que k no tenga asignado ningún valor entonces debemos hacer

```
1  if (M.find(k)!=M.end()) val = M.retrieve(k);
```

val=M.retrieve(k) retorna una *referencia* al valor asignado a k, de manera que también puede ser usado como miembro izquierdo, es decir, es valido hacer (ver §2.1.4)

```
1  M.retrieve(k) = val;
```

- k = key(p) retorna el valor correspondiente a la asignación apuntada por p
- val = value(p) retorna el valor correspondiente a la asignación apuntada por p. El valor retornado es una referencia, de manera que también podemos usar value(p) como miembro izquierdo (es decir, asignarle un valor como en value(p)=val). Notar que, por el contrario, key(p) no retorna una referencia.
- erase(p): Elimina la asignación apuntada por p. Si queremos eliminar una eventual asignación a la clave k entonces debemos hacer

```
1  p = M.find(k);
2  if (p!=M.end()) M.erase(p);
```

- p = begin(): Retorna un iterator a la primera asignación (en un orden no especificado).

- p = end(): Retorna un iterator a una asignación ficticia después de la última (en un orden no especificado).
- clear(): Elimina todas las asignaciones.
- print(): Imprime toda la tabla de asignaciones

Con esta interfaz, el programa 2.23 puede completarse como se ve en código 2.25. Notar que el programa es cuidadoso en cuanto a no crear nuevas asignaciones. El retrieve de la línea 16 está garantizado que no generará ninguna asignación involuntaria ya que el test del if garantiza que doc ya tiene asignado un valor.

```
1  map sueldo;
2  while(1) {
3    cout << "Ingrese nro. documento > ";
4    int doc;
5    double salario;
6    cin >> doc;
7    if(!doc) break;
8    iterator_t q = sueldo.find(doc);
9    if (q==sueldo.end()) {
10     cout << "Ingrese salario mensual: ";
11     cin >> salario;
12     sueldo.insert(doc,salario);
13     cout << sueldo.size() << " salarios cargados" << endl;
14   } else {
15     cout << "Doc: " << doc << ", salario: "
16         << sueldo.retrieve(doc) << endl;
17   }
18 }
19 cout << "No se ingresan mas sueldos. . ." << endl;
```

Código 2.25: *Tabla de sueldos del personal implementado con la interfaz básica de código 2.24. [Archivo: payroll2.cpp]*

2.4.2. IMPLEMENTACIÓN DE CORRESPONDENCIAS MEDIANTE CONTENEDORES LINEALES

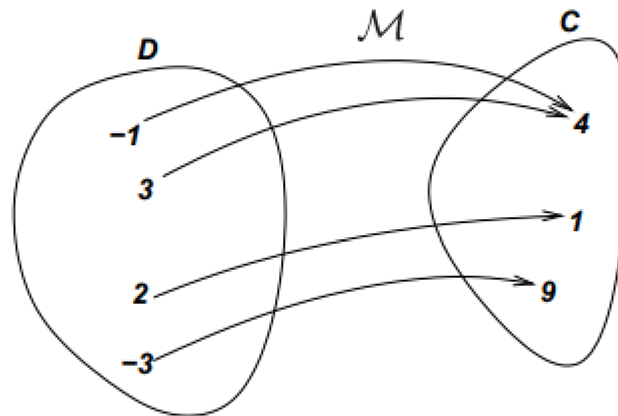


Figura 2.16: Ejemplo de correspondencia.

Tal vez la forma más simple de implementar una correspondencia es guardando en un contenedor todas las asignaciones. Para eso simplemente definimos una clase elem_t que simplemente contiene dos campos first y second con la clave y el valor de la asignación (los nombres de los campos vienen de la clase pair de las STL). Estos pares de elementos (asignaciones) se podrían guardar tanto en un vector como en una lista (list). Por ejemplo, la correspondecia de enteros a enteros que se muestra en la figura 2.15 se puede representar almacenando los siguientes pares en un contenedor:

$$(-1,4),(3,4),(2,1),(-3,9) \tag{2.19}$$

A las listas y vectores se les llama contenedores lineales, ya que en ellos existe un ordenamiento natural de las posiciones. En lo que sigue discutiremos la implementación del TAD correspondencia basadas en estos contenedores lineales. Más adelante, en otro capítulo, veremos otras implementaciones más eficientes. Cuando hablemos de la

implementación con listas asumiremos una implementación de listas basada en punteros o cursores, mientras que para vectores asumiremos arreglos estándar de C++ o el mismo vector de STL. En esta sección asumiremos que las asignaciones son insertadas en el contenedor ya sea al principio o en el final del mismo, de manera que el orden entre las diferentes asignaciones es en principio aleatorio. Más adelante discutiremos el caso en que las asignaciones se mantienen ordenadas por la clave. En ese caso las asignaciones aparecerían en el contenedor como en (2.20).

- p=find(k) debe recorrer todas las asignaciones y si encuentra una cuyo campo first coincida con k entonces debe devolver el iterator correspondiente. Si la clave no tiene ninguna asignación, entonces después de recorrer todas las asignaciones, debe devolver end(). El peor caso de find() es cuando la clave no está asignada, o la asignación está al final del contenedor, en cuyo caso es O(n) ya que debe recorrer todo el contenedor (n es el número de asignaciones en la correspondencia). Si la clave tiene una asignación, entonces el costo es proporcional a la distancia desde la asignación hasta el origen. En el peor caso esto es O(n), como ya mencionamos, mientras que en el mejor caso, que es cuando la asignación está al comienzo del contenedor, es O(1). La distancia media de la asignación es (si las asignaciones se han ingresado en forma aleatoria) la mitad del número de asociaciones y por lo tanto en promedio el costo será O(n/2).
- insert() debe llamar inicalmente a find(). Si la clave ya tiene un valor asignado, la inserción es O(1) tanto para listas como vectores. En el caso de vectores, notar que esto se debe a que no es necesario insertar una nueva asignación. Si la clave no está asignada entonces el elemento se puede insertar al final para vectores, lo cual también es O(1), y en cualquier lugar para listas.
- Un análisis similar indica que retrieve(k) también es equivalente a find().

- Para erase(p) sí hay diferencias, la implementación por listas es O(1) mientras que la implementación por vectores es O(n) ya que implica mover todos los elementos que están después de la posición eliminada
- clear() es O(1) para vectores, mientras que para listas es O(n).
- Para las restantes funciones el tiempo de ejecución en el peor caso es O(1).

Operación	lista	vector
find(key)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
insert(key,val)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
retrieve(key)	$O(1)/O(n)/O(n)$	$O(1)/O(n)/O(n)$
erase(p)	$O(1)$	$O(1)/O(n)/O(n)$
key, value, begin, end,	$O(1)$	$O(1)$
clear	$O(n)$	$O(1)$

Tabla 2.5: Tiempos de ejecución para operaciones sobre correspondencias con contenedores lineales no ordenadas. n es el número de asignaciones en la correspondencia. (La notación es *mejor/promedio/peor*. Si los tres son iguales se reporta uno sólo.)

Estos resultados se sumarizan en la Tabla 2.5. No mostraremos ninguna implementación de correspondencias con contenedores lineales no ordenados ya que discutiremos a continuación la implementación con contenedores ordenados, que es más eficiente.

UNIDAD III ÁRBOLES

Objetivo: El alumno aprenderá a realizar la secuecialización de arboles y toma de decisiones con la utilización de nodos padres, hijos y hermanos,

Los árboles son contenedores que permiten organizar un conjunto de objetos en forma jerárquica. Ejemplos típicos son los diagramas de organización de las empresas o instituciones y la estructura de un sistema de archivos en una computadora. Los árboles sirven para representar fórmulas, la descomposición de grandes sistemas en sistemas más pequeños en forma recursiva y aparecen en forma sistemática en muchísimas aplicaciones de la computación científica. Una de las propiedades más llamativas de los árboles es la capacidad de acceder a muchísimos objetos desde un punto de partida o raíz en unos pocos pasos. Por ejemplo, en mi cuenta poseo unos 61,000 archivos organizados en unos 3500 directorios a los cuales puedo acceder con un máximo de 10 cambios de directorio (en promedio unos 5).

Sorprendentemente no existe un contenedor STL de tipo árbol, si bien varios de los otros contenedores (como conjuntos y correspondencias) están implementados internamente en términos de árboles. Esto se debe a que en la filosofía de las STL el árbol es considerado o bien como un subtipo del grafo o bien como una entidad demasiado básica para ser utilizada directamente por los usuarios.

3.1. NOMENCLATURA BÁSICA DE ÁRBOLES

Un árbol es una colección de elementos llamados “nodos”, uno de los cuales es la “raíz”. Existe una relación de parentesco por la cual cada nodo tiene un y sólo un “padre”, salvo la raíz que no lo tiene. El nodo es el concepto análogo al de “posición” en la lista, es decir un objeto abstracto que representa una posición en el mismo, no directamente relacionado con el “elemento” o “etiqueta” del nodo. Formalmente, el árbol se puede definir recursivamente de la siguiente forma (ver figura 3.1)

- Un nodo sólo es un árbol
- Si n es un nodo y $T_1, T_2, \ldots, T_k$ son árboles con raíces $n_1, \ldots, n_k$ entonces podemos construir un nuevo árbol que tiene a n como raíz y donde $n_1, \ldots, n_k$ son *“hijos”* de n.

También es conveniente postular la existencia de un *“árbol vacío”* que llamaremos Λ.

Ejemplo 3.1: Consideremos el árbol que representa los archivos en un sistema de archivos. Los nodos del árbol pueden ser directorios o archivos. En el ejemplo de la figura 3.2, la cuenta anuser/ contiene 3 subdirectorios docs/, programas/ y juegos/, los cuales a su vez contienen una serie de archivos. En este caso la relación entre nodos hijos y padres corresponde a la de pertenencia: un nodo a es hijo de otro b, si

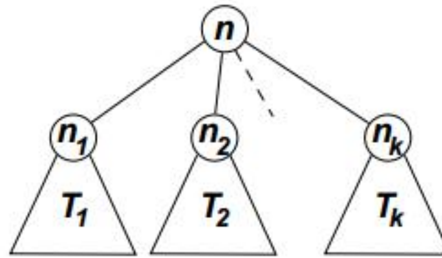


Figura 3.1: Construcción recursiva de un árbol

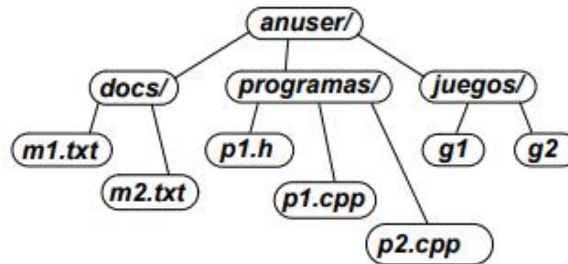


Figura 3.2: Arboles representando un sistema de archivos

el archivo a pertenecer al directorio b. En otras aplicaciones la relación padre/hijo puede representar querer significar otra cosa.

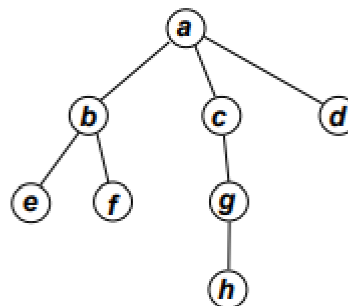


Figura 3.3: Ejemplo simple de árbol

Camino. Si n1, n2, . . . , nk es una secuencia de nodos tales que ni es padre de ni+1 para i = 1 . . . k − 1, entonces decimos que esta secuencia de nodos es un "camino" ("path"), de

manera que {anuser, docs, m2.txt} es un camino, mientras que {docs, anuser, programas} no. (Coincidentemente en Unix se llama camino a la especificación completa de directorios que va desde el directorio raíz hasta un archivo, por ejemplo, el camino correspondiente a m2.txt es /anuser/docs/m2.txt.) La “longitud” de un camino es igual al número de nodos en el camino menos uno, por ejemplo, la longitud del camino {anuser, docs, m2.txt} es

2. Notar que siempre existe un camino de longitud 0 de un nodo a sí mismo.

Descendientes y antecesores. Si existe un camino que va del nodo a al b entonces decimos que a esa antecesora de b y b es descendiente de a. Por ejemplo m1.txt es descendiente de anuser y juegos es antecesor de g2. Estrictamente hablando, un nodo es antecesor y descendiente de sí mismo ya que existe camino de longitud 0. Para diferenciar este caso trivial, decimos que a es descendiente (antecesor) propio de b si a es descendiente (antecesor) de b, pero a 6= b. En el ejemplo de la figura 3.3 a es antecesor propio de c, f y d,

Hojas. Un nodo que no tiene hijos es una “hoja” del árbol. (Recordemos que, por contraposición el nodo que no tiene padre es único y es la raíz.) En el ejemplo, los nodos e, f, h y d son hojas.

3.1.0.0.2. Profundidad de un nodo. Nivel.

La “profundidad” de un nodo es la longitud de único camino que va desde el nodo a la raíz. La profundidad del nodo g en el ejemplo es 2. Un “nivel” en el árbol es el conjunto de todos los nodos que están a una misma profundidad. El nivel de profundidad 2 en el ejemplo consta de los nodos e, f y g.

3.1.0.0.3. NODOS HERMANOS

Se dice que los nodos que tienen un mismo padre son "hermanos" entre sí. Notar que no basta con que dos nodos estén en el mismo nivel para que sean hermanos. Los nodos f y g en el árbol de la figura 3.3 están en el mismo nivel, pero no son hermanos entre sí.

3.2. ORDEN DE LOS NODOS

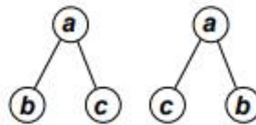


Figura 3.4: Arboles ordenados: el orden de los hijos es importante de manera que los árboles de la figura son diferentes.

En este capítulo, estudiamos árboles para los cuales el orden entre los hermanos es relevante. Es decir, los árboles de la figura 3.4 son diferentes ya que si bien a tiene los mismos hijos, están en diferente orden. Volviendo a la figura 3.3 decimos que el nodo c está a la derecha de b, o también que c es es el hermano derecho de b. También decimos que b es el "hijo más a la izquierda" de a. El orden entre los hermanos se propaga a los hijos, de manera que h está a la derecha de e ya que ambos son descendientes de c y b, respectivamente. A estos árboles se les llama "árboles ordenados orientados" (AOO).

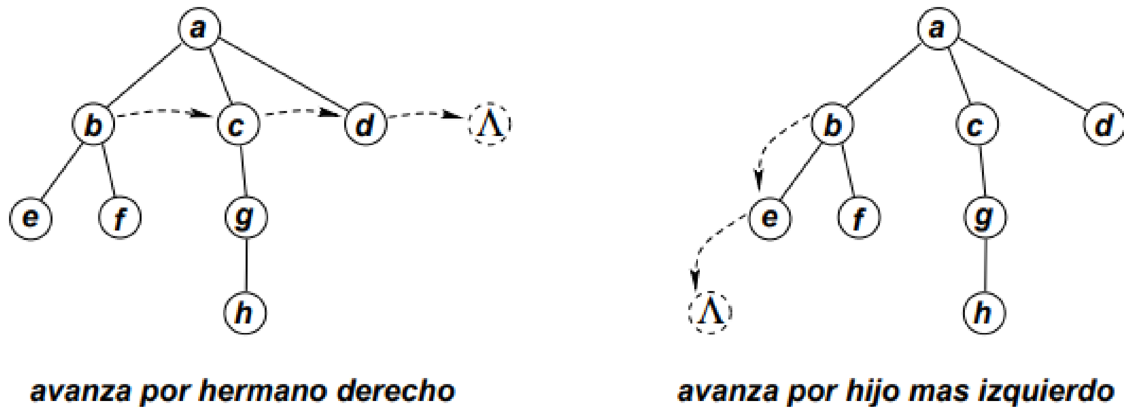


Figura 3.5: Direcciones posibles para avanzar en un árbol.

Podemos pensar al árbol como una lista bidimensional. Así como en las listas se puede avanzar linealmente desde el comienzo hacia el fin, en cada nodo del árbol podemos avanzar en dos direcciones (ver figura 3.5)

- Por el hermano derecho, de esta forma se recorre toda la lista de hermanos de izquierda a derecha.
- Por el hijo más izquierdo, tratando de descender lo más posible en profundidad.

En el primer caso el recorrido termina en el último hermano a la derecha. Por analogía con la posición end() en las listas, asumiremos que después del último hermano existe un nodo ficticio no dereferenciable. Igualmente, cuando avanzamos por el hijo más izquierdo, el recorrido termina cuando nos encontramos con una hoja. También asumiremos que el hijo más izquierdo de una hoja es un nodo ficticio no dereferenciable. Notar que, a diferencia de la lista donde hay una sola posición no dereferenciable (la posición end()), en el caso de los árboles puede haber más de una posiciones ficticias no dereferenciables, las cuales simbolizaremos con Λ cuando dibujamos el árbol. En la figura 3.6 vemos todas las posibles posiciones ficticias Λ1, . . . ,Λ8 para el árbol de la figura 3.5. Por ejemplo, el nodo f no tiene hijos, de manera que genera la posición ficticia Λ2. Tampoco tiene hermano derecho, de manera que genera la posición ficticia Λ3.

3.2.1. PARTICIONAMIENTO DEL CONJUNTO DE NODOS

Ahora bien, dados dos nodos cualquiera m y n consideremos sus caminos a la raíz. Si m es descendiente de n entonces el camino de n está incluido en el de m o viceversa. Por ejemplo, el camino de c, que es a, c, está incluido en el de h, a, c, g, h, ya que c es antecesor de h. Si entre m y n no hay relación de descendiente o antecesor, entonces los caminos se deben bifurcar necesariamente en un cierto nivel. El orden entre m y n es el orden entre los antecesores a ese nivel. Esto demuestra que, dados dos nodos cualquiera m y n sólo una de las siguientes afirmaciones puede ser cierta

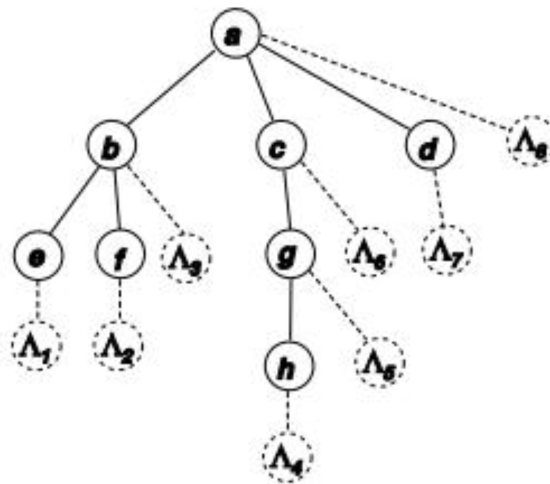


Figura 3.6: Todas las posiciones no dereferenciables de un árbol.

- $m = n$
- m es antecesor propio de n
- n es antecesor propio de m
- m está a la derecha de n
- n está a la derecha de m

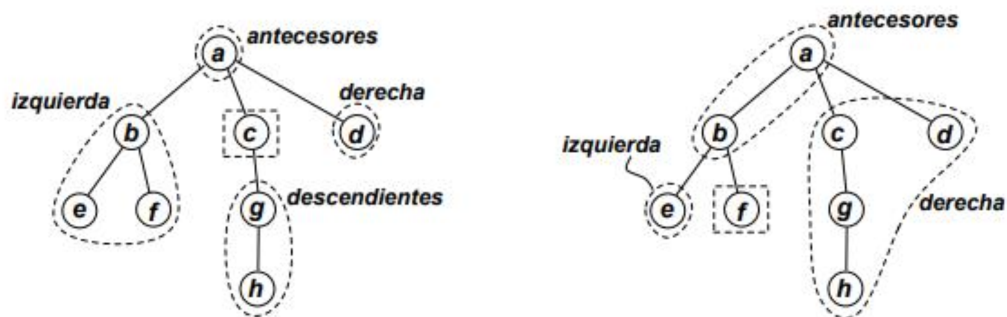


Figura 3.7: Clasificación de los nodos de un árbol con respecto a un nodo. Izquierda: con respecto al nodo c. Derecha: con respecto al nodo f

Dicho de otra manera, dado un nodo n el conjunto N de todos los nodos del árbol se puede dividir en 5 conjuntos disjuntos a saber

$$N = \{n\} \cup \{\text{descendientes}(n)\} \cup \{\text{antecesores}(n)\} \cup \{\text{derecha}(n)\} \cup \{\text{izquierda}(n)\} \qquad (3.2)$$

En la figura 3.7 vemos la partición inducida para los nodos c y f. Notar que en el caso del nodo f el conjunto de los descendientes es vacío (∅).

3.2.2. LISTADO DE LOS NODOS DE UN ÁRBOL

3.2.2.1. ORDEN PREVIO

Existen varias formas de recorrer un árbol listando los nodos del mismo, generando una lista de nodos. Dado un nodo n con hijos n1, n2, . . . nm, el "listado en orden previo" ("preorder") del nodo n que denotaremos como oprev(n) se puede definir recursivamente como sigue

$$\text{oprev}(n) = (n, \text{oprev}(n_1), \text{oprev}(n_2), \ldots, \text{oprev}(n_m)) \qquad (3.3)$$

Además el orden previo del árbol vacío es la lista vacía: $\text{oprev}(\Lambda) = ()$.

Consideremos por ejemplo el árbol de la figura 3.3. Aplicando recursivamente (3.3) tenemos

$$\begin{aligned} \text{oprev}(a) &= a, \text{oprev}(b), \text{oprev}(c), \text{oprev}(d) \\ &= a, b, \text{oprev}(e), \text{oprev}(f), c, \text{oprev}(g), d \\ &= a, b, \text{oprev}(e), \text{oprev}(f), c, \text{oprev}(g), d \\ &= a, b, e, f, c, g, \text{oprev}(h), d \\ &= a, b, e, f, c, g, h, d \end{aligned} \qquad (3.4)$$

Una forma más visual de obtener el listado en orden previo es como se muestra en la figura 3.8. Recorremos el borde del árbol en el sentido contrario a las agujas del reloj, partiendo de

un punto imaginario a la izquierda del nodo raíz y terminando en otro a la derecha del mismo, como muestra la línea de puntos. Dado un nodo como el b el camino pasa cerca de él en varios puntos (3 en el caso de b, marcados con pequeños números en el camino). El orden previo consiste en listar los nodos una sola vez, la primera vez que el camino pasa cerca del árbol. Así en el caso del nodo b, este se lista al pasar por 1. Queda como ejercicio para el lector verificar el orden resultante coincide con el dado en (3.4).

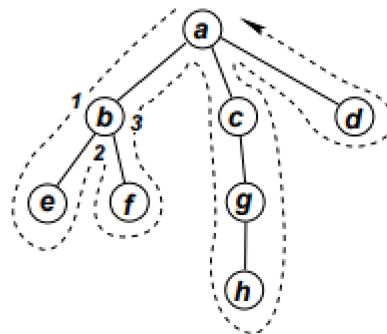


Figura 3.8: Recorrido de los nodos de un árbol en orden previo.

3.2.2.2. ORDEN POSTERIOR

El *"orden posterior"* (*"postorder"*) se puede definir en forma análoga al orden previo pero reemplazando (3.3) por

$$\operatorname{opost}(n) = (\operatorname{opost}(n_1), \operatorname{opost}(n_2), \dots, \operatorname{opost}(n_m), n) \tag{3.5}$$

y para el árbol del ejemplo resulta ser

$$\begin{aligned} \operatorname{opost}(a) &= \operatorname{opost}(b), \operatorname{opost}(c), \operatorname{opost}(d), a \\ &= \operatorname{opost}(e), \operatorname{opost}(f), b, \operatorname{opost}(g), c, d, a \\ &= e, f, b, \operatorname{opost}(h), g, c, d, a \\ &= e, f, b, h, g, c, d, a \end{aligned} \tag{3.6}$$

Visualmente se puede realizar de dos maneras.

- Recorriendo el borde del árbol igual que antes (esto es en sentido contrario a las agujas del reloj), listando el nodo la última vez que el recorrido pasa por al lado del mismo. Por ejemplo, el nodo b sería listado al pasar por el punto 3.
- Recorriendo el borde en el sentido opuesto (es decir en el mismo sentido que las agujas del reloj), y listando los nodos la primera vez que el camino pasa cera de ellos. Una vez que la lista es obtenida, invertimos la lista. En el caso de la figura el recorrido en sentido contrario daría (a, d, c, g, h, b, f, e). Al invertirlo queda como en (3.6).

Existe otro orden que se llama "simétrico", pero este sólo tiene sentido en el caso de árboles binarios, así que no será explicado aquí.

3.2.2.3. ORDEN POSTERIOR Y LA NOTACIÓN POLACA INVERTIDA

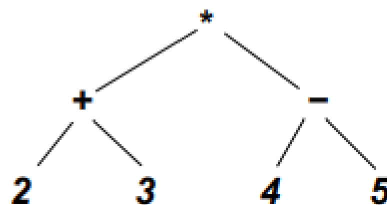


Figura 3.9: Arbol correspondiente a la expresión matemática $(2 + 3) * (4 - 5)$

Las expresiones matemáticas como $(2+ 4)*(4-5)$ se pueden poner en forma de árbol como se muestra en la figura 3.9. La regla es

- Para operadores binarios de la forma a + b se pone el operador (+) como padre de los dos operandos (a y b). Los operandos pueden ser a su vez expresiones. Funciones binarias como rem(10, 5) (rem es la función resto) se tratan de esta misma forma.
- Operadores unarios (como −3) y funciones (como sin(20)) se escriben poniendo el operando como hijo del operador o función.
- Operadores asociativos con más de dos operandos (como 1+ 3+ 4+ 9) deben asociarse de a 2 (como en (((1 + 3) + 4) + 9)).

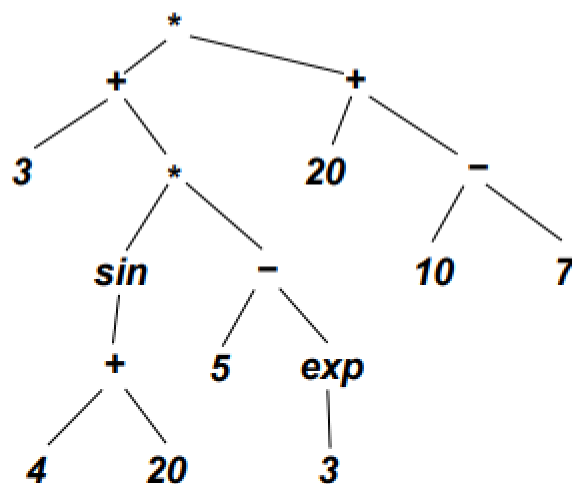


Figura 3.10: Arbol correspondiente a la expresión matemática (3.7)

De esta forma, expresiones complejas como

$$(3 + \sin(4 + 20) * (5 - e^3)) * (20 + 10 - 7) \tag{3.7}$$

pueden ponerse en forma de árbol, como en la figura 3.10.

El listado en orden posterior de este árbol coincide con la notación polaca invertida (RPN) discutida en la sección §2.2.1.

3.3. OPERACIONES CON ÁRBOLES

3.3.1. ALGORITMOS PARA LISTAR NODOS

Implementar un algoritmo para recorrer los nodos de un árbol es relativamente simple debido a su naturaleza intrínsecamente recursiva, expresada en (3.3). Un posible algoritmo puede observarse en el código 3.1. Si bien el algoritmo es genérico hemos usado ya algunos conceptos familiares de las STL, por ejemplo, las posiciones se representan con una clase iterator. Recordar que para árboles se puede llegar al “fin del contenedor”, es decir los nodos Λ, en más de un punto del contenedor. El código genera una lista de elementos L con los elementos de T en orden previo.

```
1 void preorder(tree &T,iterator n,list &L) {
2   L.insert(L.end(),/* valor en el nodo ‘n’. . . */);
3   iterator c = /* hijo mas izquierdo de n. . . */;
```

```
4  while (/* ‘c’ no es ‘Lambda’. . . */) {
5    preorder(T,c,L);
6    c = /* hermano a la derecha de c. . . */;
7  }
8 }
```

Código 3.1: *Algoritmo para recorrer un árbol en orden previo. [Archivo: preorder.cpp]*

```
1 void postorder(tree &T,iterator n,list &L) {
2  iterator c = /* hijo mas izquierdo de n . . . */;
3  while (c != T.end()) {
4    postorder(T,c,L);
5    c = /* hermano a la derecha de c . . . */;
6  }
7  L.insert(L.end(),/* valor en el nodo ‘n’. . . */);
8 }
```

Código 3.2: *Algoritmo para recorrer un árbol en orden posterior. [Archivo: postorder.cpp]*

```
1 void lisp_print(tree &T,iterator n) {
2  iterator c = /* hijo mas izquierdo de n . . . */;
3  if (/* ‘c’ es ‘Lambda’. . . */) {
4    cout << /* valor en el nodo ‘n’. . . */;
5  } else {
6    cout << "(" << /* valor de ‘n’ . . . */;
7    while (/* ‘c’ no es ‘Lambda’. . . */) {
8      cout << " ";
9      lisp_print(T,c);
10     c = /* hermano derecho de c . . . */;
11   }
12   cout << ")";
13 }
14 }
```

Código 3.3: *Algoritmo para imprimir los datos de un árbol en notación Lisp. [Archivo: lispprint.cpp]*

En el código 3.2 se puede ver un código similar para generar la lista con el orden posterior, basada en (3.5). Similarmente, en código 3.3 puede verse la implementación de una rutina que imprime la notación Lisp de un árbol.

3.3.2. INSERCIÓN EN ÁRBOLES

Para construir árboles necesitaremos rutinas de inserción supresión de nodos. Como en las listas, las operaciones de inserción toman un elemento y una posición e insertan el elemento en esa posición en el árbol.

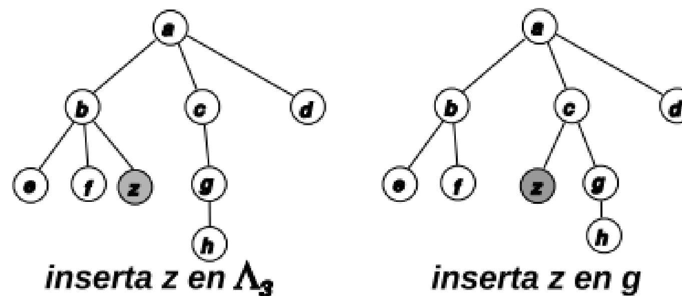


Figura 3.16: Resultado de insertar el elemento z en el árbol de la figura 3.6. *Izquierda:* Inserta z en la posición Λ_3. *Derecha:* Inserta z en la posición g.

- Cuando insertamos un nodo en una posición Λ entonces simplemente el elemento pasa a generar un nuevo nodo en donde estaba el nodo ficticio Λ. Por ejemplo, el resultado de insertar el elemento z en el nodo Λ3 de la figura 3.6 se puede observar en la figura 3.16 (izquierda). (Observación: En un abuso de notación estamos usando las mismas letras para denotar el contenido del nodo que el nodo en sí.)
- Cuando insertamos un nodo en una posición dereferenciable, entonces simplemente el elemento pasa a generar un nuevo nodo hoja en el lugar en esa posición, tal como operaría la operación de inserción del TAD lista en la lista de hijos. Por ejemplo, consideremos el resultado de insertar el elemento z en la posición g. El padre de g es c y su lista de hijos es (g). Al insertar z en la posición de g la lista de hijos pasa a ser (z, g), de manera que z pasa a ser el hijo más izquierdo de c (ver figura 3.16 derecha).
- Así como en listas insert(p,x) invalida las posiciones después de p (inclusive), en el caso de árboles, una inserción en el nodo n invalida las posiciones que son descendientes de n y que están a la derecha de n

3.3.2.1. ALGORITMO PARA COPIAR ÁRBOLES

```
1  iterator tree_copy(tree &T,iterator nt,
2                     tree &Q,iterator nq) {
3    nq = /* nodo resultante de insertar el
4          elemento de 'nt' en 'nq' . . . */;
5    iterator
6      ct = /* hijo mas izquierdo de 'nt' . . .*/,
7      cq = /* hijo mas izquierdo de 'nq' . . .*/;
8    while (/* 'ct' no es 'Lambda'. . . */) {
9      cq = tree_copy(T,ct,Q,cq);
10     ct = /* hermano derecho de 'ct'. . . */;
11     cq = /* hermano derecho de 'cq'. . . */;
12   }
13   return nq;
14 }
```

Código 3.4: *Seudocódigo para copiar un árbol. [Archivo: treecpy.cpp]*

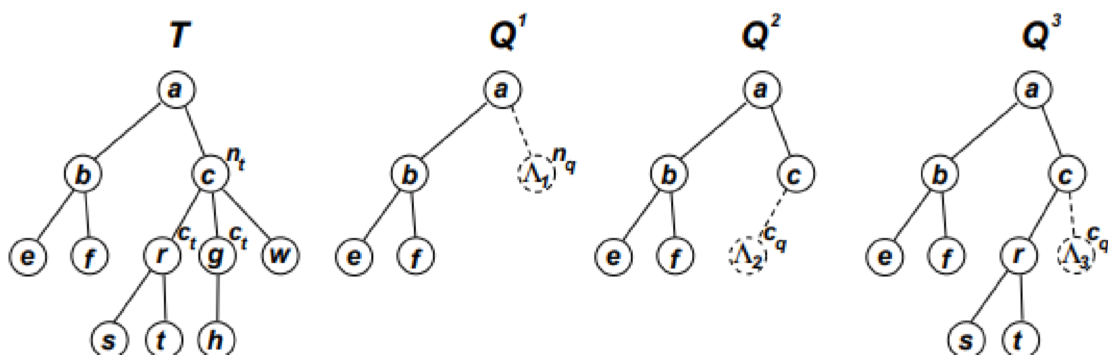


Figura 3.17: Algoritmo para copiar árboles.

Con estas operaciones podemos escribir el seudocódigo para una función que copia un árbol (ver código 3.4). Esta función copia el subárbol del nodo nt en el árbol T en la posición nq en el árbol Q y devuelve la posición de la raíz del subárbol insertado en Q (actualiza el nodo nq ya que después de la inserción es inválido). La función es recursiva, como lo son la mayoría de las operaciones no triviales sobre árboles. Consideremos el algoritmo aplicado al árbol de

la figura 3.17 a la izquierda. Primero inserta el elemento que esta en nt en la posición nq. Luego va copiando cada uno de los subárboles de los hijos de nq como hijos del nodo nt. El nodo ct itera sobre los hijos de nt mientras que cq lo hace sobre los hijos de nq. Por ejemplo, si consideramos la aplicación del algoritmo a la copia del árbol de la figura 3.17 a la izquierda, concentrémonos en la copia del subárbol del nodo c del árbol T al Q.

Cuando llamamos a tree_copy(T,nt,Q,nq), nt es c y nq es Λ1, (mostrado como Q1 en la figura). La línea 3 copia la raíz del subárbol que en este caso es el nodo c insertándolo en Λ1. Después de esta línea, el árbol queda como se muestra en la etapa Q2 . Como en la inserción en listas, la línea actualiza la posición nq, la cuál queda apuntando a la posición que contiene a c. Luego ct y nt toman los valores de los hijos más izquierdos, a saber r y Λ2. Como ct no es Λ entonces el algoritmo entra en el lazo y la línea 9 copia todo el subárbol de r en Λ2, quedando el árbol como en Q3 . De paso, la línea actualiza el iterator cq, de manera que ct y cq quedan apuntando a los dos nodos r en sus respectivos árboles. Notar que en este análisis no consideramos la llamada recursiva a tree_copy() sino que simplemente asumimos que estamos analizando la instancia específica de llamada a tree_copy donde nt es c y no aquéllas llamadas generadas por esta instancia. En las líneas 10–11, los iterators ct y cq son avanzados, de manera que quedan apuntando a g y Λ2. En la siguiente ejecución del lazo la línea 9 copiará todo el subárbol de g a Λ3. El proceso se detiene después de copiar el subárbol de w en cuyo caso ct obtendrá en la línea 10 un nodo Λ y la función termina, retornando el valor de nq actualizado

```
1 iterator mirror_copy(tree &T,iterator nt,
2                 tree &Q,iterator nq) {
3   nq = /* nodo resultante de insertar
4         el elemento de 'nt' en 'nq' */;
5   iterator
```

```
6     ct = /* hijo mas izquierdo de 'nt' . . .*/,
7     cq = /* hijo mas izquierdo de 'nq' . . .*/;
8   while (/* 'ct' no es 'Lambda'. . . */) {
9     cq = mirror_copy(T,ct,Q,cq);
10    ct = /* hermano derecho de 'ct' . . . */;
11  }
12  return nq;
13 }
```

Código 3.5: *Seudocódigo para copiar un árbol en espejo. [Archivo: mirrorcpy.cpp]*

Con menores modificaciones la función puede copiar un árbol en forma espejada, es decir, de manera que todos los nodos hermanos queden en orden inverso entre sí. Para eso basta con no avanzar el iterator cq donde se copian los subárboles, es decir eliminar la línea 11 . Recordar que si en una lista se van insertando valores en una posición sin avanzarla, entonces los elementos quedan ordenados en forma inversa a como fueron ingresados. El algoritmo de copia espejo mirror_copy() puede observarse en el código 3.5. Con algunas modificaciones el algoritmo puede ser usado para obtener la copia de un árbol reordenando las hojas en un orden arbitrario, por ejemplo, dejándolas ordenadas entre sí.

3.5. IMPLEMENTACIÓN DE LA INTERFAZ BÁSICA POR PUNTEROS

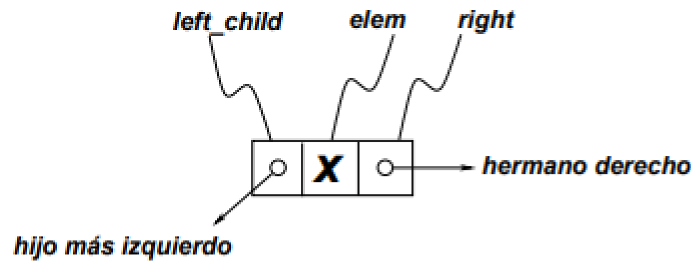


Figura 3.19: Celdas utilizadas en la representación de árboles por punteros.

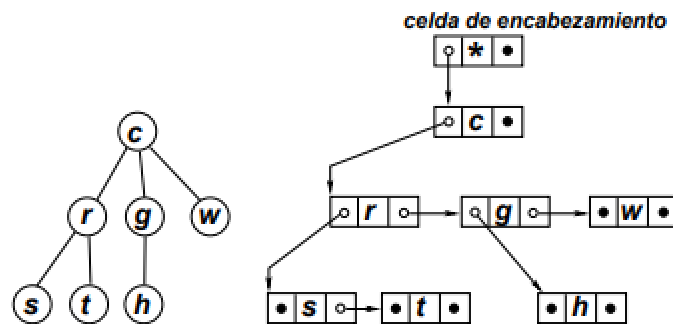


Figura 3.20: Representación de un árbol con celdas enlazadas por punteros.

Así como la implementación de listas por punteros mantiene los datos en celdas enlazadas por un campo next, es natural considerar una implementación de árboles en la cual los datos son almacenados en celdas que contienen, además del dato elem, un puntero right a la celda que corresponde al hermano derecho y otro left_child al hijo más izquierdo (ver figura 3.19). En la figura 3.20 vemos un árbol simple y su representación mediante celdas enlazadas.

3.5.1. EL TIPO ITERATOR

Por analogía con las listas podríamos definir el tipo iterator_t como un typedef a cell *. Esto bastaría para representar posiciones dereferenciables y posiciones no dereferenciables que provienen de haber aplicado right() al último hermano, como la posición Λ3 en la figura 3.21. Por supuesto habría que mantener el criterio de usar "posiciones adelantadas" con respecto al dato. Sin embargo no queda en claro como representar posiciones no dereferenciables como la Λ1 que provienen de aplicar lchild() a una hoja.

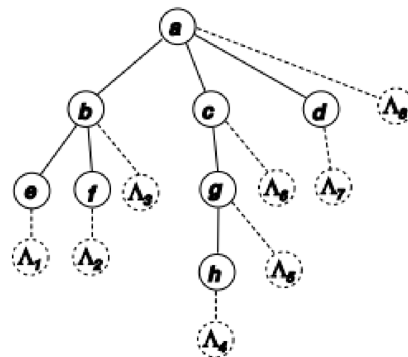


Figura 3.21: Todas las posiciones no dereferenciables de un árbol.

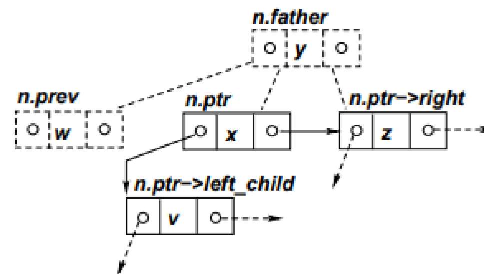


Figura 3.22: Entorno local de un iterator sobre árboles.

Una solución posible consiste en hacer que el tipo iterator_t contenga, además de un puntero a la celda que contiene el dato, punteros a celdas que de otra forma serían inaccesibles. Entonces el iterator consiste en tres punteros a celdas (ver figura 3.22) a saber,

- Un puntero ptr a la celda que contiene el dato. (Este puntero es nulo en el caso de posiciones no dereferenciables).
- Un puntero prev al hermano izquierdo.

- Un puntero father al padre.

Así, por ejemplo, a continuación mostramos los juegos de punteros correspondientes a varias posiciones dereferenciables y no dereferenciables en el árbol de la figura 3.6:

- nodo e: ptr=e, prev=NULL, father=b.
- nodo f: ptr=f, prev=e, father=b.
- nodo Λ1: ptr=NULL, prev=NULL, father=e.
- nodo Λ2: ptr=NULL, prev=NULL, father=f.
- nodo Λ3: ptr=NULL, prev=f, father=b.
- nodo g: ptr=g, prev=NULL, father=c.
- nodo Λ6: ptr=NULL, prev=g, father=c.

Estos tres punteros tienen la suficiente información como para ubicar a todas las posiciones (dereferenciables o no) del árbol. Notar que todas las posiciones tienen un puntero father no nulo, mientras que el puntero prev puede o no ser nulo. Para tener un código más uniforme se introduce una celda de encabezamiento, al igual que con las listas. La raíz del árbol, si existe, es una celda hija de la celda de encabezamiento. Si el árbol está vacío, entonces el iterator correspondiente a la raíz (y que se obtiene llamando a begin()) corresponde a ptr=NULL, prev=NULL, father=celda de encabezamiento.

3.5.2. LAS CLASES CELL E ITERATOR_T

```
class tree;
class iterator_t;

//---:---<*>---:---<*>---:---<*>---:---<*>
class cell {
  friend class tree;
  friend class iterator_t;
  elem_t elem;
  cell *right, *left_child;
  cell() : right(NULL), left_child(NULL) {}
};

//---:---<*>---:---<*>---:---<*>---:---<*>
class iterator_t {
private:
  friend class tree;
  cell *ptr,*prev,*father;
  iterator_t(cell *p,cell *prev_a, cell *f_a)
    : ptr(p), prev(prev_a), father(f_a) { }
public:
  iterator_t(const iterator_t &q) {
    ptr = q.ptr;
    prev = q.prev;
    father = q.father;
  }
  bool operator!=(iterator_t q) { return ptr!=q.ptr; }
  bool operator==(iterator_t q) { return ptr==q.ptr; }
  iterator_t()
    : ptr(NULL), prev(NULL), father(NULL) { }

  iterator_t lchild() {
    return iterator_t(ptr->left_child,NULL,ptr);
```

```
33   }
34   iterator_t right() {
35     return iterator_t(ptr->right,ptr,father);
36   }
37 };
38
39 //---:---<*>---:---<*>---:---<*>---:---<*>
40 class tree {
41 private:
42   cell *header;
43   tree(const tree &T) {}
44 public:
45
46   tree() {
47     header = new cell;
48     header->right = NULL;
49     header->left_child = NULL;
50   }
51   ~tree() { clear(); delete header; }
52
53   elem_t &retrieve(iterator_t p) {
54     return p.ptr->elem;
55   }
56
57   iterator_t insert(iterator_t p,elem_t elem) {
58     assert(!(p.father==header && p.ptr));
59     cell *c = new cell;
60     c->right = p.ptr;
61     c->elem = elem;
62     p.ptr = c;
63     if (p.prev) p.prev->right = c;
64     else p.father->left_child = c;
65     return p;
66   }
67   iterator_t erase(iterator_t p) {
68     if(p==end()) return p;
69     iterator_t c = p.lchild();

70    while (c!=end()) c = erase(c);
71    cell *q = p.ptr;
72    p.ptr = p.ptr->right;
73    if (p.prev) p.prev->right = p.ptr;
74    else p.father->left_child = p.ptr;
75    delete q;
76    return p;
77  }
78
79  iterator_t splice(iterator_t to,iterator_t from) {
80    assert(!(to.father==header && to.ptr));
81    if (from.ptr->right == to.ptr) return from;
82    cell *c = from.ptr;
83
84    if (from.prev) from.prev->right = c->right;
85    else from.father->left_child = c->right;
```

```
86
87      c->right = to.ptr;
88      to.ptr = c;
89      if (to.prev) to.prev->right = c;
90      else to.father->left_child = c;
91
92      return to;
93    }
94
95    iterator_t find(elem_t elem) {
96      return find(elem,begin());
97    }
98    iterator_t find(elem_t elem,iterator_t p) {
99      if(p==end() || retrieve(p) == elem) return p;
100     iterator_t q,c = p.lchild();
101     while (c!=end()) {
102       q = find(elem,c);
103       if (q!=end()) return q;
104       else c = c.right();
105     }
106     return iterator_t();
107   }
108   void clear() { erase(begin()); }
109   iterator_t begin() {
110     return iterator_t(header->left_child,NULL,header);
111   }
112   iterator_t end() { return iterator_t(); }
```

Código 3.9: *Implementación de la interfaz básica de árboles por punteros. [Archivo: treebas.h]*

Una implementación de la interfaz básica código 3.7 por punteros puede observarse en el código 3.9.

- Tenemos primero las declaraciones "hacia adelante" de tree e iterator_t. Esto nos habilita a declarar friend a las clases tree e iterator_t.
- La clase cell sólo declara los campos para contener al puntero al hijo más izquierdo y al hermano derecho. El constructor inicializa los punteros a NULL.
- La clase iterator_t declara friend a tree, pero no es necesario hacerlo con cell. iterator_t declara los campos punteros a celdas y por comodidad declaramos un constructor privado iterator_t(cell *p,cell *pv, cell *f) que simplemente asigna a los punteros internos los valores de los argumentos.

- Por otra parte existe un constructor público iterator_t(const iterator_t &q). Este es el "constructor por copia" y es utilizado cuando hacemos por ejemplo iterator_t p(q); con q un iterador previamente definido.
- El operador de asignación de iteradores (por ejemplo, p=q) es sintetizado por el compilador, y simplemente se reduce a una copia bit a bit de los datos miembros de la clase (en este caso los tres punteros prt, prev y father) lo cual es apropiado en este caso.
- Haber definido iterator_t como una clase (y no como un typedef) nos obliga a definir también los operadores != y ==. Esto nos permitirá comparar nodos por igualdad o desigualdad (p==q o p!=q). De la misma forma que con las posiciones en listas, los nodos no pueden compararse con los operadores de relación de orden (<=, > y >=). Notar que los operadores == y != sólo comparan el campo ptr de forma que todas las posiciones no dereferenciables (Λ) son "iguales" entre sí. Esto permite comparar en los lazos cualquier posición p con end(), entonces p==end() retornará true incluso si p no es exactamente igual a end() (es decir tiene campos father y prev diferentes).
- El constructor por defecto iterator_t() devuelve un iterator con los tres punteros nulos. Este iterator no debería ser normalmente usado en ninguna operación, pero es invocado automáticamente por el compilador cuando declaramos iterators como en: iterator p;

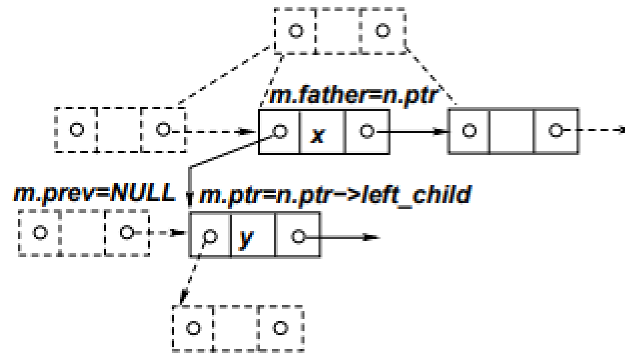


Figura 3.23: La función lchild().

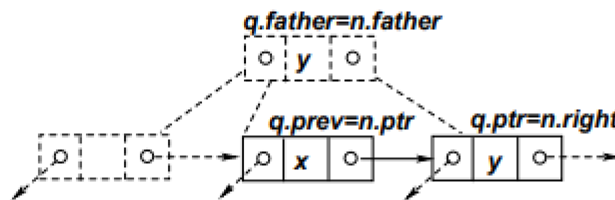


Figura 3.24: La función right().

Las operaciones lchild() y right() son las que nos permiten movernos dentro del árbol. Sólo pueden aplicarse a posiciones dereferenciables y pueden retornar posiciones dereferenciables o no. Para entender como funcionan consideremos la información contenida en un iterator. Si consideramos un iterator n (ver figura 3.23), entonces la posición de m=n.lchild() está definida por los siguientes punteros

- m.ptr= n.ptr->left_child
- m.prev= NULL (ya que m es un hijo más izquierdo)
- m.father= n.ptr

Por otra parte, si consideramos la función hermano derecho: q=n.right(), entonces q está definida por los siguientes punteros,

- q.ptr= n.ptr->right
- q.prev= n.ptr
- q.father= n.father

UDS Mi Universidad

UNIDAD IV ORDENAMIENTO

Objetivo: El alumno conocerá los métodos de ordenamiento programable para la obtención de autómatas programados.

El proceso de ordenar elementos (“sorting”) en base a alguna relación de orden (ver sección §2.4.4) es un problema tan frecuente y con tantos usos que merece un capítulo aparte. Nosotros nos concentraremos en el ordenamiento interna, es decir cuando todo el contenedor reside en la memoria principal de la computadora. El tema del ordenamiento externo, donde el volumen de datos a ordenar es de tal magnitud que requiere el uso de memoria auxiliar, tiene algunas características propias y por simplicidad no será considerado aquí.

4.1 INTRODUCCIÓN

Asumiremos normalmente que los elementos a ordenar pertenecen a algún tipo key_t con una relación de orden $<$ y que están almacenados en un contenedor lineal (vector o lista). El problema de ordenar un tal contenedor es realizar una serie de intercambios en el contenedor de manera de que los elementos queden ordenados, es decir $k_0 \leq k_1 \leq \ldots \leq k_{n-1}$, donde k_j es la clave del j-ésimo elemento. Si el ordenamiento se hace sin requerir ningún tipo de memoria adicional (que crezca con n), decimos que es “in-place”. Si no, se debe tener en cuenta también como crece la cantidad de memoria requerida.

4.1.1 RELACIONES DE ORDEN DÉBILES

Ya hemos discutido como se define una relación de orden en la sección §2.4.4. Igual que para definir conjuntos o correspondencias, la relación de ordenamiento puede a veces ser elegida por conveniencia. Así, podemos querer ordenar un conjunto de números por su valor absoluto. En ese caso al ordenar los números (1, 3, −2, −4) el contenedor ordenado

resultaría en (1, −2, 3, −4). En este caso la relación de orden la podríamos denotar por y la definiríamos como

$$a \lhd u, \quad \text{si } |a| < |u|. \tag{5.1}$$

La relación binaria definida así no resulta ser una relación de orden en el sentido fuerte, como definida en la sección §2.4.4, ya que, por ejemplo, para el par −2, 2 no son ciertos 2 −2, ni −2 2 ni −2 = 2. La segunda condición sobre las relaciones de orden puede relajarse un poco y obtener la definición de relaciones de orden débiles: Definición: “” es una relación de orden débil en el conjunto C si,

1. $\lhd$ *es transitiva, es decir, si* $a \lhd b$ *y* $b \lhd c$*, entonces* $a \lhd c$.

2. *Dados dos elementos cualquiera* a, b *de* C

$$(a \lhd b) \,\&\&\, (b \lhd a) = \texttt{false} \tag{5.2}$$

Esta última condición se llama de antisimetría. Usamos aquí la notación de C++ para los operadores lógicos, por ejemplo && indica “and” y los valores booleanos true y false. Es decir, a b y b a no pueden ser verdaderas al mismo tiempo (son exclusivas).

Los siguientes, son ejemplos de relaciones de orden débiles, pero no cumplen con la condición de antisimetría fuerte

- Menor en valor absoluto para enteros, es decir

$$a \lhd b \text{ si } |a| < |b| \tag{5.3}$$

La antisimetría fuerte no es verdadera para el par 2, -2, es decir, ni es cierto que $-2 \lhd 2$ ni $2 \lhd -2$ ni $2 = -2$.
- Pares de enteros por la primera componente, es decir $(a, b) \lhd (c, d)$ si $a < c$. En este caso la antisimetría fuerte se viola para $(2, 3)$ y $(2, 4)$, por ejemplo.
- Legajos por el nombre del empleado. Dos legajos para diferentes empleados que tienen el mismo nombre (*nombre*=Perez, Juan,*DNI*=14231235) y (*nombre*=Perez, Juan,*DNI*=12765987) violan la condición.

Si dos elementos satisfacen que !(a b) && !(b a) entonces decimos que son equivalentes y lo denotamos por a ≡ b

También se pueden definir otras relaciones derivadas como

$$\begin{array}{ll} \text{mayor:} & (a \rhd b) = (b \lhd a) \\ \text{equivalencia:} & (a \equiv b) = !(a \lhd b) \text{ \&\& } !(b \lhd a) \\ \text{menor o equivalente:} & (a \unlhd b) = !(b \lhd a) \\ \text{mayor o equivalente:} & (a \unrhd b) = !(a \lhd b) \end{array} \tag{5.4}$$

También en algunos lenguajes (e.g. Perl) es usual definir una función int cmp(T x,T y) asociada a una dada relación de orden que retorna 1, 0 o -1 dependiendo si x y, x ≡ y o x y. En ese caso, el valor de cmp(x,y) se puede obtener de

$$\mathrm{cmp}(x, y) = (y \lhd x) - (x \lhd y) \tag{5.5}$$

donde en el miembro derecho se asume que las expresiones lógicas retornan 0 o 1 (como es usual en C). En la Tabla 5.1 se puede observar que todos los operadores de comparación se pueden poner en términos de cualquiera de los siguientes , , ,

---

, cmp(·, ·). Notar que los valores retornados por cmp(·, ·) son comparados directamente con la relación de orden para enteros etc... y no con la relación , , ... y amigos.

4.1.2 SIGNATURA DE LAS RELACIONES DE ORDEN. PREDICADOS BINARIOS

En las STL las relaciones de orden se definen mediante “predicados binarios”, es decir, su signatura es de la forma

La expresión:	Usando: $\lhd$	Usando: $\unlhd$
$a \lhd b$	$a \lhd b$	$!\ (b \unlhd a)$
$a \unlhd b$	$!\ (b \lhd a)$	$a \unlhd b$
$a \rhd b$	$b \lhd a$	$!\ a \unlhd b$
$a \unrhd b$	$!\ a \lhd b$	$b \unrhd a$
$\mathrm{cmp}(a,b)$	$(b \lhd a) - (a \lhd b)$	$(b \unlhd a) - (a \unlhd b)$
$a = b$	$!\ (a \lhd b \vert\vert b \lhd a)$	$a \unlhd b \,\&\&\, b \unlhd a$
$a \neq b$	$a \lhd b \vert\vert b \lhd a$	$(!\ a \unlhd b) \vert\vert (!\ b \unlhd a)$

La expresión:	Usando: $\rhd$	Usando: $\unrhd$
$a \lhd b$	$b \rhd a$	$!\ a \unrhd b$
$a \unlhd b$	$!\ (b \rhd a)$	$b \unrhd a$
$a \rhd b$	$a \rhd b$	$!\ b \unrhd a$
$a \unrhd b$	$!\ b \rhd a$	$a \unrhd b$
$\mathrm{cmp}(a,b)$	$(a \rhd b) - (b \rhd a)$	$(a \unrhd b) - (b \unrhd a)$
$a = b$	$!\ (a \rhd b \vert\vert b \rhd a)$	$a \unrhd b \,\&\&\, b \unrhd a)$
$a \neq b$	$a \rhd b \vert\vert b \rhd a$	$(!\ a \unrhd b) \vert\vert (!\ b \unrhd a)$

La expresión:	Usando: $\mathrm{cmp}(\cdot,\cdot)$
$a \lhd b$	$\mathrm{cmp}(a,b) = -1$
$a \unlhd b$	$\mathrm{cmp}(a,b) \leq 0$
$a \rhd b$	$\mathrm{cmp}(a,b) = 1$
$a \unrhd b$	$\mathrm{cmp}(a,b) \geq 0$
$\mathrm{cmp}(a,b)$	$\mathrm{cmp}(a,b)$
$a = b$	$!\ \mathrm{cmp}(a,b)$
$a \neq b$	$\mathrm{cmp}(a,b)$

Tabla 5.1: Equivalencia entre los diferentes operadores de comparación.

bool (*binary_pred)(T x, T Y);

Ejemplo 5.1: Consigna: Escribir una relación de orden para comparación lexicográfica de cadenas de caracteres. Hacerlo en forma dependiente e independiente de mayúsculas y minúsculas (case-sensitive y case-insensitive)

La clase string de las STL permite encapsular cadenas de caracteres con ciertas características mejoradas con respecto al manejo básico de C. Entre otras cosas, tiene sobrecargado el operador "

```
1 bool string_less_cs(const string &a,const string &b) {
2   int na = a.size();
```

```
3   int nb = b.size();
4   int n = (na>nb ? nb : na);
5   for (int j=0; j<n; j++) {
6     if (a[j] < b[j]) return true;
7     else if (b[j] < a[j]) return false;
8   }
9   return na<nb;
10 }
```

Código 5.1: *Función de comparación para cadenas de caracteres con orden lexicográfico. [Archivo: stringlcs.cpp]*

En el código 5.1 vemos la implementación del predicado binario correspondiente. Notar que este predicado binario termina finalmente comparando caracteres en las líneas 6–7. A esa altura se está usando el operador de comparación sobre el tipo char que es un subtipo de los enteros. Si ordenamos las cadenas pepes juana PEPE Juana JUANA Pepe, con esta función de comparación obtendremos

`JUANA Juana PEPE Pepe juana pepe` (5.6)

Recordemos que en la serie ASCII las mayúsculas están antes que las minúsculas, las minúsculas a-z están en el rango 97-122 mientras que las mayúsculas A-Z están en el rango 65-90.

```
bool string_less_cs3(const string &a,const string &b) {
  return a<b;
}
```

Código 5.2: *Función de comparación para cadenas de caracteres con orden lexicográfico usando el operador "<" de* C++. *[Archivo: stringlcs3.cpp]*

```
template<class T>
bool less(T &x,T &y) {
  return x<y;
}
```

Código 5.3: *Template de las STL que provee un predicado binario al operador intrínseco "<" del tipo* T. *[Archivo: lesst.h]*

Como ya mencionamos el orden lexicográfico está incluido en las STL, en forma del operador "que devuelve el predicado binario correspondiente al operador intrínseco "

```
char tolower(char c) {
  if (c>='A' && c<='Z') c += 'a'-'A';
  return c;
}

bool string_less_ci(const string &a,
                    const string &b) {
  int na = a.size();
  int nb = b.size();
  int n = (na>nb ? nb : na);
  for (int j=0; j<n; j++) {
    char
      aa = tolower(a[j]),
      bb = tolower(b[j]);
    if (aa < bb) return true;
    else if (bb < aa) return false;
  }
  return na<nb;
}
```

Código 5.4: *Función de comparación para cadenas de caracteres con orden lexicográfico independiente de mayúsculas y minúsculas. [Archivo: stringlci.cpp]*

Si queremos ordenar en forma independiente de mayúsculas/minúsculas, entonces podemos definir una relación binaria que básicamente es

$$(a \lhd b) \;= (\mathtt{tolower}(a) < \mathtt{tolower}(b)) \tag{5.7}$$

donde la función tolower() convierte su argumento a minúsculas. La función tolower() está definida en la librería estándar de C (libc) para caracteres. Podemos entonces definir la función de comparación para orden lexicográfico independiente de mayúsculas/minúsculas como se muestra en el código 5.4. La función string_less_ci() es básicamente igual a la string_less_cs() sólo que antes de comparar caracteres los convierte con tolower() a minúsculas. De esta forma pepe, Pepe y PEPE resultan equivalentes.

4.2 MÉTODOS DE ORDENAMIENTO LENTOS

Llamamos "rápidos" a los métodos de ordenamiento con tiempos de ejecución menores o iguales a O(n log n). Al resto lo llamaremos "lentos". En esta sección estudiaremos tres algoritmos lentos, a saber, burbuja, selección e inserción.

4.2.1 EL MÉTODO DE LA BURBUJA

```
1  template<class T> void
2  bubble_sort(typename std::vector<T>::iterator first,
3           typename std::vector<T>::iterator last,
4               bool (*comp)(T&,T&)) {
5    int size = last-first;
6    for (int j=0; j<size-1; j++) {
7      for (int k=size-1; k>j; k--) {
8        if (comp(*(first+k),*(first+k-1))) {
9          T tmp = *(first+k-1);
10         *(first+k-1) = *(first+k);
11         *(first+k) = tmp;
12       }
13     }
14   }
15 }
16
17 template<class T> void
18 bubble_sort(typename std::vector<T>::iterator first,
19          typename std::vector<T>::iterator last) {
```

```
20   bubble_sort(first,last,less<T>);
21 }
```

Código 5.5: Algoritmo de ordenamiento de la burbuja. [Archivo: bubsort.h

El método de la burbuja fue introducido en la sección §1.4.2. Nos limitaremos aquí a discutir la conversión al formato compatible con la STL. El código correspondiente puede observarse en el código 5.5.

- Para cada algoritmo de ordenamiento presentamos las dos funciones correspondientes, con y sin función de comparación.
- Ambas son templates sobre el tipo T de los elementos a ordenar.

- La que no tiene operador de comparación suele ser un “wrapper” que llama a la primera pasándole como función de comparación less.
- Notar que como las funciones no reciben un contenedor, sino un rango de iteradores, no se puede referenciar directamente a los elementos en la forma v[j] sino a través del operador de dereferenciación *p. Así, donde normalmente pondríamos v[first+j] debemos usar *(first+j).
- Recordar que las operaciones aritméticas con iteradores son válidas ya que los contenedores son de acceso aleatorio. En particular, las funciones presentadas son sólo válidas para vector<>, aunque se podrían modificar para incluir a deque<> en forma relativamente fácil
- Notar la comparación en la línea 8 usando el predicado binario comp() en vez de operator
- Para la discusión de los diferentes algoritmos haremos abstracción de la posición de comienzo first, como si fuera 0. Es decir consideraremos que se está ordenando el rango [0,n) donde n=last-first

4.2.2 EL MÉTODO DE INSERCIÓN

```
1   template<class T> void
2   insertion_sort(typename
3               std::vector<T>::iterator first,
4               typename
5               std::vector<T>::iterator last,
6               bool (*comp)(T&,T&)) {
7     int size = last-first;
8     for (int j=1; j<size; j++) {
9       T tmp = *(first+j);
10      int k=j-1;
11      while (comp(tmp,*(first+k))) {
12        *(first+k+1) = *(first+k);
13        if (--k < 0) break;
14      }
15      *(first+k+1) = tmp;
16    }
17  }
18
19

20  template<class T> void
21  insertion_sort(typename
22              std::vector<T>::iterator first,
23              typename
24              std::vector<T>::iterator last) {
25    insertion_sort(first,last,less<T>);
26  }
```

Código 5.6: *Algoritmo de ordenamiento por inserción. [Archivo: inssorta.h]*

En este método (ver código 5.6) también hay un doble lazo. En el lazo sobre j el rango [0, j) está ordenado e insertamos el elemento j en el rango [0, j), haciendo los desplazamientos necesarios. El lazo sobre k va recorriendo las posiciones desde j hasta 0 para ver donde debe insertarse el elemento que en ese momento está en la posición j.

4.2.3 EL MÉTODO DE SELECCIÓN

```
template<class T> void
selection_sort(typename
            std::vector<T>::iterator first,
            typename
            std::vector<T>::iterator last,
            bool (*comp)(T&,T&)) {
  int size = last-first;
  for (int j=0; j<size-1; j++) {
    typename std::vector<T>::iterator
      min = first+j,
      q = min+1;
    while (q<last) {
      if (comp(*q,*min)) min = q;
      q++;
    }
    T tmp = *(first+j);
    *(first+j) = *min;
    *min = tmp;
  }
}
```

Código 5.7: *Algoritmo de ordenamiento por selección. [Archivo: selsort.h]*

En este método (ver código 5.7) también hay un doble lazo (esta es una característica de todos los algoritmos lentos). En el lazo j se elige el menor del rango [j, N) y se intercambia con el elemento en la posición j.

4.3 ORDENAMIENTO INDIRECTO

Si el intercambio de elementos es muy costoso (pensemos en largas listas, por ejemplo) podemos reducir notablemente el número de intercambios usando "ordenamiento indirecto", el cual se basa en ordenar un vector de cursores o punteros a los objetos reales. De esta forma el costo del intercambio es en realidad el de intercambio de los cursores o punteros, lo cual es mucho más bajo que el intercambio de los objetos mismos.

```
1  template<class T>
2  void apply_perm(typename std::vector<T>::iterator first,
3                 typename std::vector<T>::iterator last,
4                 std::vector<int> &indx) {
5   int size = last-first;
6   assert(indx.size()==size);
7   int sorted = 0;
8   T tmp;
9   while (sorted<size) {
10    if(indx[sorted]!=sorted) {
11      int k = sorted;
12      tmp = *(first+k);
13      while (indx[k]!=sorted) {
14        int kk = indx[k];
15        *(first+k)=*(first+kk);
16        indx[k] = k;
17        k = kk;
18      }
19      *(first+k) = tmp;
20      indx[k] = k;
21    }
22    sorted++;
23  }
24 }
25
26 template<class T>
27 void ibubble_sort(typename std::vector<T>::iterator first,
28                   typename std::vector<T>::iterator last,
29                   bool (*comp)(T&,T&)) {
30  int size = last-first;
31  std::vector<int> indx(size);
32  for (int j=0; j<size; j++) indx[j] = j;
33
34  for (int j=0; j<size-1; j++) {
35    for (int k=size-1; k>j; k--) {
36      if (comp(*(first+indx[k]),*(first+indx[k-1]))) {
37        int tmp = indx[k-1];
38        indx[k-1] = indx[k];
39        indx[k] = tmp;
```

```
40       }
41     }
42   }
43   apply_perm<T>(first,last,indx);
44 }
45
46 template<class T>
47 void ibubble_sort(typename std::vector<T>::iterator first,
48                   typename std::vector<T>::iterator last) {
49   ibubble_sort(first,last,less<T>);
50 }
```

Código 5.8: *Método de la burbuja con ordenamiento indirecto. [Archivo: ibub.h]*

En el código 5.8 vemos el método de la burbuja combinado con ordenamiento indirecto. Igual que para el ordenamiento directo existen versiones con y sin función de comparación

(ibubble_sort(first,last,comp) y ibubble_sort(first,last)). Se utiliza un vector auxiliar de enteros indx. Inicialmente este vector contiene los enteros de 0 a size-1. El método de la burbuja procede en las líneas 34–42, pero en vez de intercambiar los elementos en el contenedor real, se mueven los cursores en indx. Por ejemplo, después de terminar la ejecución del lazo para j=0, en vez de quedar el menor en *first, en realidad queda la posición correspondiente al menor en indx[0]. En todo momento indx[] es una permutación de los enteros en [0,size). Cuando termina el algoritmo de la burbuja, al llegar a la línea 43, indx[] contiene la permutación que hace ordena [first,last). Por ejemplo el menor de todos los elementos está en la posición first+indx[0], el segundo menor en first+indx[1], etc... La última operación de ibubble_sort() es llamar a la función apply_perm(first,last,indx) que aplica la permutación obtenida a los elementos en [first,last). Puede verse que esta función realiza n intercambios o menos. Esta función es genérica y puede ser combinada con cualquier otro algoritmo de ordenamiento indirecto. En la figura 5.1 vemos como queda el vector v[] con los elementos desordenados, y la permutación indx[].

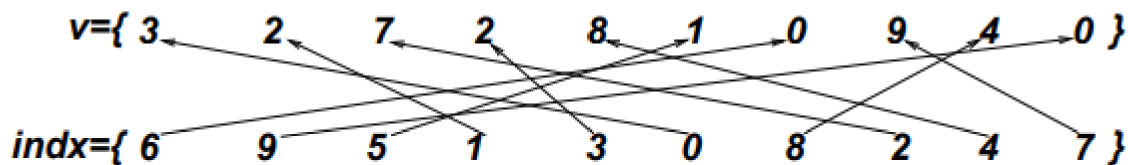


Figura 5.1: Ordenamiento indirecto.

El código en las líneas 34–42, es básicamente igual al de la versión directa bubble_sort() sólo que donde en ésta se referencian elementos como *(first+j) en la versión indirecta se referencian como *(first+indx[j]). Por otra parte, donde la versión indirecta intercambia los elementos *(first+k-1) y *(first+k), la versión indirecta intercambia los índices en indx[k-1] y indx[k].

En el caso de usar ordenamiento indirecto debe tenerse en cuenta que se requiere memoria adicional para almacenar indx[].

4.3.1 MINIMIZAR LA LLAMADA A FUNCIONES

Si la función de comparación se obtiene por composición, y la función de mapeo es muy costosa. Entonces tal vez convenga generar primero un vector auxiliar con los valores de la función de mapeo, ordenarlo y después aplicar la permutación resultante a los elementos reales. De esta forma el número de llamadas a la función de mapeo pasa de ser O(n 2) a O(n). Por ejemplo, supongamos que tenemos un conjunto de árboles y queremos ordenarlos por la suma de sus valores nodales. Podemos escribir una función

```
int sum_node_val(tree<int> &A); y escribir una función de comparación
1  bool comp_tree(tree<int> &A,tree<int> &A) {
2    return sum_node_val(A) < sum_node_val(B);
3  }
```

Si aplicamos por ejemplo bubble_sort, entonces el número de llamadas a la función será O(n 2). Si los árboles tienen muchos nodos, entonces puede ser preferible generar un vector de enteros con los valores de las sumas y ordenarlo. Luego se aplicaría la permutación correspondiente al vector de árboles.

En este caso debe tenerse en cuenta que se requiere memoria adicional para almacenar el vector de valores de la función de mapeo, además del vector de índices para la permutación.

4.4. EL MÉTODO DE ORDENAMIENTO RÁPIDO, QUICK-SORT

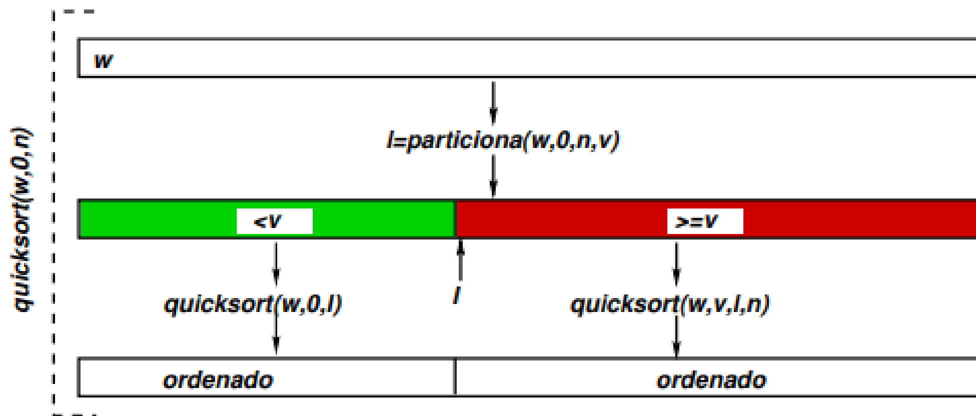


Figura 5.2: Esquema *"dividir para vencer"* para el algoritmo de ordenamiento rápido, *quick-sort*.

Este es probablemente uno de los algoritmos de ordenamiento más usados y se basa en la estrategia de "dividir para vencer". Se escoge un elemento del vector v llamado "pivote" y se "particiona" el vector de manera de dejar los elementos ≥ v a la derecha (rango [l, n), donde l es la posición del primer elemento de la partición derecha) y los < v a la izquierda (rango [0, l)). Está claro que a partir de entonces, los elementos en cada una de las particiones quedarán en sus respectivos rangos, ya que todos los elementos en la partición derecha son estrictamente mayores que los de la izquierda. Podemos entonces aplicar quick-sort recursivamente a cada una de las particiones.

```
1 void quicksort(w,j1,j2) {
2   // Ordena el rango [j1,j2) de 'w'
```

```
3   if (n==1) return;
4   // elegir pivote v . . .
5   l = partition(w,j1,j2,v);
6   quicksort(w,j1,l);
7   quicksort(w,l,j2);
8 }
```

Código 5.9: *Seudocódigo para el algoritmo de ordenamiento rápido. [Archivo: qsortsc.cpp]*

Si garantizamos que cada una de las particiones tiene al menos un elemento, entonces en cada nivel de recursividad los rangos a los cuales se le aplica quick-sort son estrictamente menores. La recursión termina cuando el vector a ordenar tiene un sólo elemento.

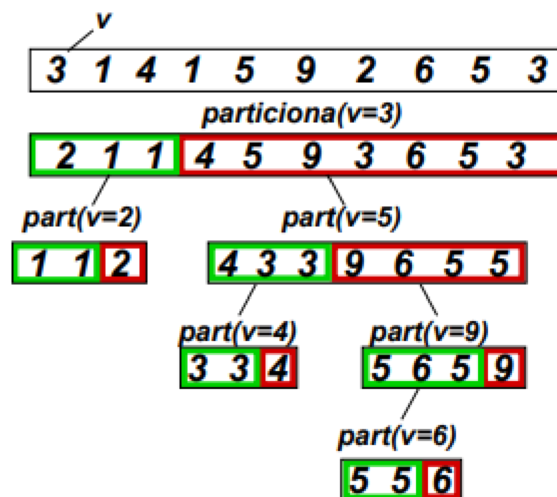


Figura 5.3: Ejemplo de aplicación de quick-sort

En la figura 5.3 vemos un ejemplo de aplicación de quick-sort a un vector de enteros. Para el ejemplo hemos elegido como estrategia para elegir el pivote tomar el mayor de los dos primeros elementos distintos. Si la secuencia a ordenar no tiene elementos distintos, entonces la recursión termina. En el primer nivel los dos primeros elementos distintos (de izquierda a derecha) son el 3 y el 1, por lo que el pivote será v = 3. Esto induce la partición

que se observa en la línea siguiente, donde tenemos los elementos menores que 3 en el rango [0, 3) y los mayores o iguales que 3 en el rango [3, 10). Todavía no hemos explicado cómo se hace el algoritmo de partición, pero todavía esto no es necesario para entender como funciona quick-sort. Ahora aplicamos quick-sort a cada uno de los dos rangos. En el primer caso, los dos primeros elementos distintos son 2 y 1 por lo que el pivote es 2. Al particionar con este pivote quedan dos rangos en los cuales no hay elementos distintos, por lo que la recursión termina allí. Para la partición de la derecha, en cambio, todavía debe aplicarse quick-sort dos niveles más. Notar que la estrategia propuesta de elección del pivote garantiza que cada una de las particiones tendrá al menos un elemento ya que si los dos primeros elementos distintos de la secuencia son a y b, y a < b, por lo cual v = b, entonces al menos hay un elemento en la partición derecha (el elemento b) y otro en la partición izquierda (el elemento a).

4.4.1 TIEMPO DE EJECUCIÓN. CASOS EXTREMOS

El tiempo de ejecución del algoritmo aplicado a un rango $[j_1, j_2)$ de longitud $n = j_2 - j_1$ es

$$T(n) = T_{\text{part-piv}}(n) + T(n_1) + T(n_2) \quad (5.10)$$

donde n1 = l − j1 y n2 = j2 − l son las longitudes de cada una de las particiones. Tpart−piv(n) es el costo de particionar la secuencia y elegir el pivote.

El mejor caso es cuando podemos elegir el pivote de tal manera que las particiones resultan ser bien balanceadas, es decir n1 ≈ n2 ≈ n/2. Si además asumimos que el algoritmo de particionamiento y elección del pivote son O(n), es decir

UDS Mi Universidad

$$T_{\text{part-piv}} = cn \tag{5.11}$$

(más adelante se explicará un algoritmo de particionamiento que satisface esto) Entonces

$$\begin{aligned} T(n) &= T_{\text{part-piv}}(n) + T(n_1) + T(n_2) \\ &= cn + T(n/2) + T(n/2) \end{aligned} \tag{5.12}$$

Llamando $T(1) = d$ y aplicando sucesivamente

$$\begin{aligned} T(2) &= c + 2T(1) = c + 2d \\ T(4) &= 4c + 2T(2) = 3 \cdot 4c + 4d \\ T(8) &= 8c + 2T(4) = 4 \cdot 8c + 8d \\ T(16) &= 16c + 2T(8) = 5 \cdot 16c + 16d \\ \vdots &= \vdots \\ T(2^p) &= (p+1)n(c+d) \end{aligned} \tag{5.13}$$

pero como $n = 2^p$ entonces $p = \log_2 n$ y por lo tanto

$$T(n) = O(n \log n). \tag{5.14}$$

Por otro lado el peor caso es cuando la partición es muy desbalanceada, es decir $n_1 = 1$ y $n_2 = n - 1$ o viceversa. En este caso tenemos

$$\begin{aligned} T(n) &= cn + T(1) + T(n-1) \\ &= cn + d + T(n-1) \end{aligned} \tag{5.15}$$

y aplicando sucesivamente,

$$\begin{aligned} T(2) &= 2c + 2d \\ T(3) &= 3c + d + (2c + 2d) = 5c + 3d \\ T(4) &= 4c + d + (5c + 3d) = 9c + 4d \\ T(5) &= 5c + d + (9c + 4d) = 14c + 5d \\ \vdots &= \vdots \\ T(n) &= \left(\frac{n(n+1)}{2} - 2\right) c + nd = O(n^2) \end{aligned} \tag{5.16}$$

El peor caso ocurre, por ejemplo, si el vector está inicialmente ordenado y usando como estrategia para el pivote el mayor de los dos primeros distintos. Si tenemos en v, por ejemplo los enteros 1 a 100 ordenados, que podemos denotar como un rango [1, 100], entonces el pivote sería inicialmente 2. La partición izquierda tendría sólo al 1 y la derecha

sería el rango [2, 99]. Al particionar [2, 99] tendríamos el pivote 3 y las particiones serían 2 y [3, 99] (ver figura 5.4). Puede verificarse que lo mismo ocurriría si el vector está ordenado al revés.

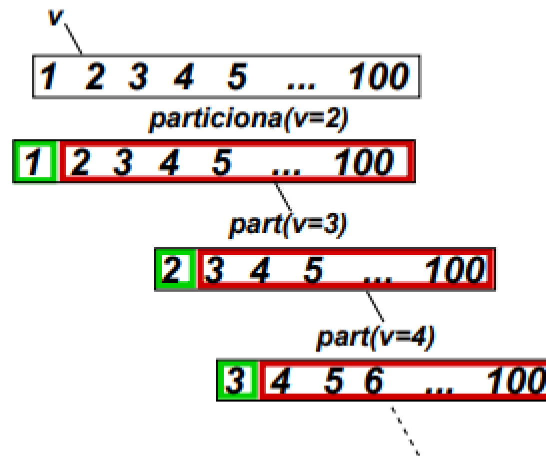


Figura 5.4: Particionamiento desbalanceado cuando el vector ya está ordenado.

4.4.2 ELECCIÓN DEL PIVOTE

En el caso promedio, el balance de la partición dependerá de la estrategia de elección del pivote y de la distribución estadística de los elementos del vector. Compararemos a continuación las estrategias que corresponden a tomar la "mediana" de los k primeros distintos. Recordemos que para k impar la mediana de k elementos es el elemento que queda en la posición del medio del vector (la posición $(k - 1)/2$ en base 0) después de haberlos ordenado. Para k par tomamos el elemento en la posición $k/2$ (base 0) de los primeros k distintos, después de ordenarlos.

No confundir la mediana con el "promedio" o "media" del vector que consiste en sumar los valores y dividirlos por el número de elementos. Si bien es muy sencillo calcular el promedio en O(n) operaciones, la mediana requiere en principio ordenar el vector, por lo que

claramente no se puede usar la mediana como estrategia para elegir el pivote. En lo que resta, cuando hablamos de elegir la mediana de k valores del vector, asumimos que k es un valor constante y pequeño, más concretamente que no crece con n.

En cuanto al promedio, tampoco es una buena opción para el pivote. Consideremos un vector con 101 elementos 1,2,...,99,100,100000. La mediana de este vector es 51 y, por supuesto da una partición perfecta, mientras que el promedio es 1040.1, lo cual daría una pésima partición con 100 elementos en la partición izquierda y 1 elemento en la derecha. Notemos que esta mala partición es causada por la no uniformidad en la distribución de los elementos, para distribuciones más uniformes de los elementos tal vez, es posible que el promedio sea una elección razonable. De todas formas, el promedio es un concepto que es sólo aplicable a tipos para los cuales las operaciones algebraicas tienen sentido. No es claro cómo podríamos calcular el promedio de una serie de cadenas de caracteres.

Volviendo En el caso $k = 2$ tomamos de los dos primeros elementos distintos el que está en la posición 1, es decir el mayor de los dos, de manera que $k = 2$ equivale a la estrategia propuesta en las secciones anteriores. El caso del balance perfecto (5.12) se obtiene tomando como pivote la mediana de todo el vector, es decir $k = n$.

Para una dada estrategia de elección del pivote podemos preguntarnos, cual es la probabilidad $P(n, n1)$ de que el pivote genere subparticiones de longitud $n1$ y $n-n1$, con $1 \leq n1 < n$. Asumiremos que los elementos del vector están distribuidos aleatoriamente. Si, por ejemplo, elegimos como pivote el primer elemento del vector (o sea la mediana de los primeros $k = 1$ distintos), entonces al ordenar el vector este elemento puede terminar en cualquier posición del vector, ordenado de manera que $P(n, n1) = 1/(n - 1)$ para cualquier $n1 = 1, .., n - 1$. Por supuesto, recordemos que esta no es una elección aceptable para el pivote en la práctica ya que no garantizaría que ambas particiones sean no nulas. Si el primer

elemento resultara ser el menor de todos, entonces la partición izquierda resultaría ser nula. En la figura 5.6 vemos esta distribución de probabilidad. Para que la curva sea independiente de n hemos graficado nP(n, n1) en función de n1/n.

$$
\begin{array}{cccc}
a & b & x & x \\
a & x & b & x \\
a & x & x & b \\
b & a & x & x \\
x & a & b & x \\
x & a & x & b \\
b & x & a & x \\
x & b & a & x \\
x & x & a & b \\
b & x & x & a \\
x & b & x & a \\
x & x & b & a
\end{array}
$$

Figura 5.5: Posibilidades al ordenar un vector 5 elementos. a y b son los primeros dos elementos antes de ordenar.

Ahora consideremos la estrategia propuesta en las secciones anteriores, es decir el mayor de los dos primeros elementos distintos. Asumamos por simplicidad que todos los elementos son distintos. Sean a y b los dos primeros elementos del vector, entonces después de ordenar los elementos estos elementos pueden terminar en cualquiera de las posiciones j, k del vector con la misma probabilidad. Si la longitud del vector es n = 4, entonces hay n(n − 1) = 12 posibilidades esencialmente distintas como se muestra en la figura 5.5.

Notemos que las primeras tres corresponden a que a termine en la posición 0 (base 0) y b en cada una de las tres posiciones restantes. Las siguientes 3 corresponden a a en la posición 1 (base 0), y así siguiendo. En el primero de los doce casos el pivote sería b y terminaría en la posición 1. Revisando todos los posibles casos tenemos que en 2 casos (líneas 1 y 4) el pivote termina en la posición 1, en 4 casos (líneas 2, 5, 7 y 8) termina en la posición 2 y en 6 casos (líneas 3, 6, 9, 10, 11 y 12) termina en la posición 3. Notemos que en este caso es más probable que el pivote termine en las posiciones más a la derecha que en las que están más a

la izquierda. Por supuesto esto se debe a que estamos tomando el mayor de los dos. Una forma de contar estas posibilidades es considerar que para que el mayor esté en la posición j debe ocurrir que a esté en la posición j y b en las posiciones 0 a j −1, o que b quede en la posición j y a en las posiciones 0 a j −1, o sea un total de 2j posibilidades.

Ahora veamos que ocurre si tomamos como estrategia para la elección del pivote la mediana de los primeros k = 3 distintos. En este caso, si denotamos los tres primeros distintos como a, b y c, entonces

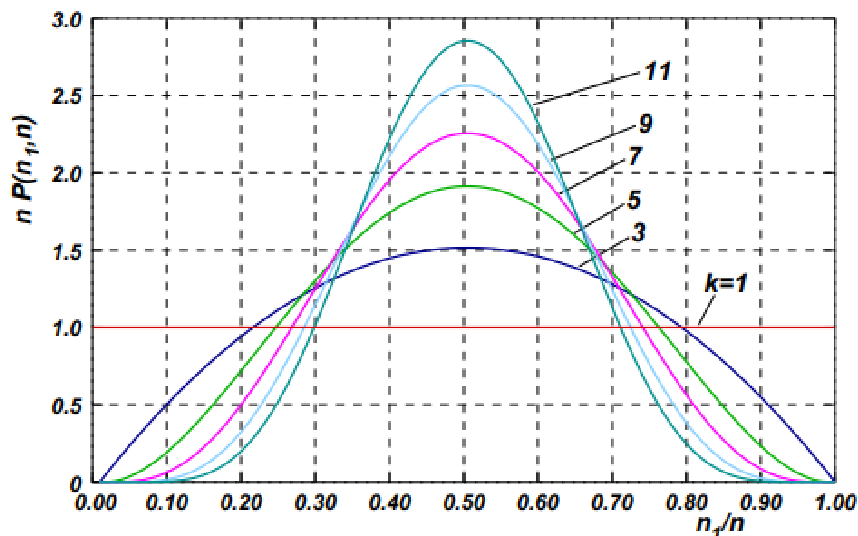


Figura 5.6: Probabilidad de que el pivote aparezca en una dada posición n_1, para diferentes valores de k.

existen n(n − 1)(n − 2) casos distintos: a en cualquiera de las n posiciones, b en cualquiera de las n − 1 restantes y c en cualquiera de las n − 2 restantes. Para que el pivote quede en la posición j debería ocurrir que, por ejemplo, a quede en la posición j, b en una posición [0, j) y c en una posición (j, n), o sea j(n−j−1) posibilidades. Las restantes posibilidades se obtienen por permutación de los elementos a, b y c, en total

$$
\begin{array}{lll}
a \text{ en la posición } j, & b \text{ en } [0,j) & c \text{ en } (j,n) \\
a \text{ en la posición } j, & c \text{ en } [0,j) & b \text{ en } (j,n) \\
b \text{ en la posición } j, & a \text{ en } [0,j) & c \text{ en } (j,n) \\
b \text{ en la posición } j, & c \text{ en } [0,j) & a \text{ en } (j,n) \\
c \text{ en la posición } j, & a \text{ en } [0,j) & b \text{ en } (j,n) \\
c \text{ en la posición } j, & b \text{ en } [0,j) & a \text{ en } (j,n)
\end{array}
$$

O sea que en total hay $6j(n-j-1)$ posibilidades. Esto está graficado en la figura 5.6 como $k = 3$. Notemos que al tomar $k = 3$ hay mayor probabilidad de que el pivote particione en forma más balanceada. En la figura se muestra la distribución de probabilidades para los k impares y se observa que a medida que crece k a más certeza de que el pivote va a quedar en una posición cercana al centro de la partición.

BIBLIOGRAFIA

- Aho, J. Hopcroft, and J. Ullman. Data Structures and Algorithms. Addison Wesley, 1987. Free Software Foundation. GNU Compiler Collection GCC manual, a. GCC version 3.2, http://www.gnu. org.
- Free Software Foundation. The GNU C Library Reference Manual, b. Version 2.3.x of the GNU C Library, edition 0.10, http://www.gnu.org/software/libc/libc.html.
- R. Hernández, J.C. Lázaro, R. Dormido, and S. Ros. Estructuras de Datos y Algoritmos. Prentice Hall, 2001. D. E. Knuth. The Art of Computer Programming. Addison-Wesley, 1981.
- SGI. Standard Template Library Programmer's Guide, 1999. http://www.sgi.com/tech/stl