UNIVERSIDAD ESTATAL A DISTANCIA
ESCUELA DE CIENCIAS EXACTAS Y NATURALES
PROGRAMA INFORMÁTICA ADMINISTRATIVA

GUÍA DE ESTUDIO DEL CURSO

ESTRUCTURA DE DATOS

CÓDIGO 825

Yenory Carballo Valverde

2007

Producción Académica:
Ana Láscaris-Comneno Slepuhin

Encargada de Cátedra y
especialista de contenidos:
Karol Castro Cháves

Revisión Filológica:
Óscar Alvarado Vega

Diagramación:
Rocío Zúñiga Guzmán

TABLA DE CONTENIDO

PRESENTACIÓN

Esta Guía de Estudio está diseñada con la finalidad de orientarlo a través del cuatrimestre, indicándole claramente los temas y capítulos del libro de texto que debe cubrir, así como la secuencia en la que debe hacerlo. Al final de cada tema encontrará ejercicios sugeridos, que le servirán para evaluar su aprendizaje.

El objetivo de este curso es proporcionar una introducción práctica a las estructuras de datos y los algoritmos desde el punto de vista del pensamiento abstracto y la resolución de problemas.

El curso adopta un enfoque novedoso separando la presentación de cada estructura en su especificación (a través de una interfaz Java) e implementación. Este enfoque proporciona varios beneficios, entre ellos la promoción del pensamiento abstracto. Antes de conocer la implementación se presenta la interfaz de la clase, motivando así al estudiante a pensar desde el principio sobre la funcionalidad y eficiencia potencial de las distintas estructuras de datos.

Los estudiantes que utilizarán el libro de texto que brinda el curso de Estructura de datos deben poseer previamente conocimientos básicos sobre algún lenguaje de programación modular u orientado a objetos. El curso presenta el material utilizando el lenguaje de programación Java. Este es un lenguaje relativamente reciente que a menudo se compara con C++. Java ofrece diversas ventajas; muchos programadores lo consideran un lenguaje más seguro, portable y fácil de usar que C++.

Java proporciona también soporte para la programación concurrente, donde varios procesos se ejecutan en paralelo comunicándose entre sí de forma primitiva. El principal atractivo de Java es que se trata de un lenguaje seguro y portable que soporta las construcciones de la programación moderna orientada a objetos.

DESCRIPCIÓN DEL CURSO

OBJETIVO GENERAL

El objetivo de este curso es introducir al estudiante en un lenguaje orientado a objetos, utilizando un pensamiento abstracto y la resolución de problemas en las estructuras de datos, su análisis y sus implementaciones.

OBJETIVOS ESPECÍFICOS

Al finalizar este curso, usted deberá estar en capacidad de:

- Enseñar aspectos básicos de las estructuras de datos.
- Enseñar aspectos básicos sobre la reutilización de componentes.
- Explicar cómo se implementa la recursión y comprender cuándo se debe utilizar.
- Explicar los algoritmos de ordenación.
- Explicar aspectos básicos de los grafos, y cómo estos se representan, por medio de estructuras de datos.
- Explicar los diferentes algoritmos para resolver diversas variantes del problema de camino mínimo.
- Explicar el funcionamiento de las pilas y colas y explicar su implementación.
- Explicar el funcionamiento de las listas enlazadas y explicar su implementación.
- Explicar el funcionamiento de la estructura de datos conocida como árbol y explicar su implementación.
- Explicar el funcionamiento de la estructura de datos conocida como árbol binario de búsqueda, y explicar su implementación.

REQUISITOS DEL CURSO

Este curso tiene una carga académica asignada de tres créditos. Es parte del plan de diplomado de la carrera Informática Administrativa. En él se asume que usted ha aprobado, como mínimo, el curso *Introducción a la Programación* o que posee conocimientos básicos en estas áreas. El no tener conocimientos previos que le entrega el curso antes mencionado le dificultará enormemente el éxito en esta asignatura. Por favor considere estos aspectos antes de seguir adelante.

MATERIAL DE APOYO

La siguiente lista de materiales didácticos se brinda a los estudiantes el día que matricula el curso. Su objetivo es proporcionar al estudiante la ayuda necesaria para comprender los temas de estudio.

- Weiss, Mark Allen (2000). *Estructura de Datos en JAVA TM*. Pearson Educación S.A. , España. 776 Pp.
- Castro Chaves, Karol (2007). *La orientación al curso de Estructura de Datos.* Costa Rica, EUNED, Universidad Estatal a Distancia.
- Esta guía de estudio que usted está leyendo.

DESGLOSE DE CAPÍTULOS

El curso *Estructura de Datos* consta de 8 capítulos principales:

- Estructuras de datos
- Recursión
- Algoritmos de ordenación
- Grafos y caminos
- Pilas y colas
- Listas enlazadas
- Árboles
- Árboles binarios de búsqueda

Para un adecuado aprovechamiento del curso, se escogió utiliza,r como unidad didáctica, el libro de texto autodidáctico de Weiss, que motiva al estudiante a continuar con el aprendizaje de los temas señalados.

En la siguiente tabla se detallan los temas principales, los subtemas correspondientes, el número del capítulo del libro y el número de páginas del libro donde podrán localizar cada uno de ellos:

Tema	Capítulo del libro	Páginas
Tema 1		
• Estructuras de datos	6	137 -158
• Recursión	7	165-200
Tema 2		
• Algoritmos de ordenación	8	213-239
• Grafos y caminos	14	353-382
Tema 3		
• Pilas y colas	15	395-414
• Listas enlazadas	16	415-434
Tema 4		
• Árboles	17	435-459
• Árboles binarios de búsqueda	18	467-477

GUÍA DE LECTURAS

En cada tema de esta Guía de Estudio usted encontrará una sección llamada *Guía de lecturas.* Esta tiene como finalidad indicarle las páginas respectivas que usted debe leer y estudiar de su libro de texto, para cada capítulo y subcapítulo.

COMENTARIOS GENERALES

Los comentarios generales presentados para cada capítulo en esta Guía de Estudio brindan aspectos importantes de este capítulo, y su ubicación dentro de cada capítulo del libro de texto. Le servirán para sintetizar los conceptos transmitidos. De esta manera, usted podrá determinar si requiere repasar o aclarar alguno de los conceptos antes de desarrollar los ejercicios.

EJERCICIOS SUGERIDOS

Con el propósito de que usted haga una autoevaluación de su comprensión y aprendizaje del capítulo de estudio, esta guía incluye una sección llamada *Ejercicios sugeridos*, que selecciona algunos de todos los ejercicios incluidos al final de cada capítulo del libro de texto. Además, para algunos de los temas, se incluye un ejemplo adicional al presentado en el libro de texto, así como un ejercicio sugerido para desarrollar en la tutoría correspondiente al tema en estudio.

RESOLUCIÓN A EJERCICIOS SUGERIDOS

En esta área usted encontrará las soluciones a los ejercicios sugeridos.

GLOSARIO

En esta área usted encontrará un glosario donde se incluyen términos y la definición de cada uno para que le sea más fácil comprender el tema de estudio.

CAPÍTULO 1

ESTRUCTURAS DE DATOS

PROPÓSITO DEL CAPÍTULO

Al finalizar el estudio de este capítulo, el estudiante deberá contar con las técnicas algorítmicas básicas que le permitirán abordar el desarrollo de programas correctos y eficientes para resolver problemas no triviales. Las técnicas básicas mencionadas incluyen conocimientos teóricos y prácticos, habilidades, experiencias y sentido crítico, todas ellas fundamentadas en teorías y técnicas sólidas, comprobadas y bien establecidas.

OBJETIVOS DE APRENDIZAJE

- Explicar nuevas técnicas de programación. En particular, el uso de la memoria dinámica y las estructuras de datos enlazadas, que están en la base de muchas aplicaciones.

- Introducir herramientas de diseño de algoritmos y la ingeniería algorítmica como selección de las estructuras de datos y de las técnicas algorítmicas más adecuadas para la resolución de un problema concreto.

- Profundizar en el aprendizaje de la programación estructurada. Introducir técnicas para diseñar programas de tamaño mediano. Proporcionar al alumno más experiencia en el campo de la programación mediante la realización de prácticas.

GUÍA DE LECTURAS

Para lograr los objetivos anteriores, se le sugiere seguir la siguiente guía:

Subtema	Capítulo	Páginas
¿Por qué necesitamos estructuras de datos?	6	137 - 138
Las pilas	6	139 - 142
Las colas	6	142 - 143
Listas enlazadas	6	144 - 147
Árboles	6	147 - 153
Tablas Hash	6	153 - 155
Colas de prioridad	6	155 - 158

COMENTARIOS GENERALES

Una vez que usted ha realizado las lecturas anteriores, analice cuidadosamente los comentarios siguientes, ya que con ellos se pretende enfatizar o ampliar algunos contenidos importantes del capítulo. Existen varios aspectos importantes que debemos señalar antes de entrar al desarrollo de los subtemas.

Muchos algoritmos requieren de una representación apropiada de los datos para lograr ser eficientes. Esta representación, junto con las operaciones permitidas, se llama **estructura de datos**.

Las estructuras de datos nos permiten lograr un importante objetivo de la programación orientada a objetos: la reutilización de componentes. Una vez que la estructura de datos ha sido implementada, puede ser utilizada una y otra vez en diversas aplicaciones. Este enfoque –la separación de la interfaz y la implementación– es parte del paradigma de la programación orientada a objetos. El usuario de la estructura de datos no necesita ver la implementación, solo las operaciones disponibles. Esta es la parte de ocultamiento y encapsulación de la programación orientada a objetos. Sin embargo, otra parte importante de la programación orientada a objetos es la **abstracción**. Tenemos que pensar cuidadosamente el diseño de las estructuras de datos porque debemos escribir programas que utilicen esas estructuras de datos sin tener en cuenta su implementación. Esto hace a la interfaz más limpia, más flexible (más reutilizable y, generalmente, más fácil de implementar).

Existen diferentes estructuras de datos, tales como:

- **La pila:** Es una estructura de datos en la cual el acceso está limitado al elemento más reciente insertado. En una pila, las tres operaciones naturales de insertar, eliminar y buscar se renombran por apilar, desafilar y cima.

- **La cola:** Restringe el acceso al elemento insertado menos reciente. Las operaciones básicas permitidas por las colas son: insertar al final de la línea, eliminación del elemento al frente de la línea y primer acceso al elemento en el frente de la línea.

- **Listas enlazadas**: Los elementos se almacenan de forma no contigua, en vez de un vector de posiciones de memoria consecutivas. Para conseguir esto, cada elemento se almacena en un nodo que contiene el objeto y una referencia al siguiente nodo en la lista. En este marco, se mantienen referencias al primer y último nodo de la lista.

- **Árbol**: Estructura de datos utilizada muy a menudo, formada por un conjunto de nodos y un conjunto de aristas que conectan pares de nodos. En este texto solo se consideran árboles con raíz. Un árbol con raíz tiene las siguientes características:

  - Un nodo es distinguido como la raíz.
  - Todo nodo c, excepto la raíz, está conectado por medio de una arista a un único nodo p,; p es el padre de c, y c es uno de los hijos de p.
  - Existe un único camino desde la raíz a cada nodo. El número de aristas que deben atravesarse es la longitud del camino.

- **Árboles binarios de búsqueda**: Aquellos en los que podemos insertar o eliminar elementos del árbol.

- **Tablas hash:** Soportan inserciones, eliminaciones y búsquedas en tiempo constante en promedio. Para poder utilizar una tabla hash, debemos proporcionar una función hash que convierta un objeto específico en un entero.

GLOSARIO

En esta área usted encontrará un glosario donde se incluyen términos y la definición de cada uno para que le sea más fácil comprender el tema de estudio.

Árbol: Estructura de datos muy utilizada, formada por un conjunto de nodos y un conjunto de aristas que conectan pares de nodos.

Árbol binario: Árbol con, a lo sumo, dos hijos por nodo.

Árbol binario de búsqueda: Árbol que permite la inserción, eliminación y la búsqueda. También se puede utilizar para acceder al K-ésimo menor elemento.

Clase iteradora: Clase que permite acceder a una lista. La clase de la lista tiene métodos que reflejan el estado de la lista; el resto de operaciones está en la clase iterador.

Cola: Estructura de datos que limita el acceso al elemento insertando menos recientemente.

Diccionario: Almacena claves, que son las que buscan en el diccionario, y sus correspondientes definiciones.

Estructura de datos: Representación de los datos junto con unas operaciones permitidas sobre dichos datos. Las estructuras de datos nos permiten lograr la reutilización de componentes.

Función hash: Función que convierte un objeto *hashable* en un entero.

Hoja: En un árbol, es un nodo sin hijos.

Longitud de un camino: En un árbol, el número de aristas que hay que atravesar, desde la raíz, para alcanzar un nodo.

Pila: Estructura de datos que restringe el acceso al elemento más recientemente insertado.

EJERCICIOS SUGERIDOS

Para este tema, se sugiere que usted realice los siguientes ejercicios de su libro de texto:

6.7 Escriba una rutina que imprima en orden inverso los elementos de una lista enlazada.

6.10 Muestre cómo implementar eficientemente una cola utilizando una lista enlazada como atributo y manteniendo un objeto.

RESOLUCIÓN DE EJERCICIOS SUGERIDOS

6.7 *Escriba una rutina que imprima en orden inverso los elementos de una lista enlazada.*

R/ /*
* Dada una lista daremos la vuelta al orden de sus elementos
* mediante la operación de colecciones: reverse

```
   */
   package aulambra.j2se.utils;
   import java.util.*;
   public class InvertirLista {
     public InvertirLista() {
       super();
     }
     public static void main(java.lang.String[] args) {
       List list = new ArrayList();
       for (int i = 1; i <= 10; i++)
         list.add(Integer.toString(i));
       // Damos la vuelta a la lista
       Collections.reverse(list);
       // Mostramos el contenido de la lista
       Iterator iter = list.iterator();
       while (iter.hasNext())
         System.out.println(iter.next());
     }
}
```

6.10 *Muestre cómo implementar eficientemente una cola utilizando una lista enlazada como atributo y manteniendo un objeto.*

Los desarrolladores normalmente utilizan un *array* unidimensional para implementar una cola. Sin embargo, si tienen que coexistir múltiples colas o las inserciones en las colas deben ocurrir en posiciones distintas a la última por motivos de prioridades, los desarrolladores suelen cambiar a la lista doblemente enlazada. Con un *array* unidimensional, dos variables enteras (normalmente llamadas ***front*** y ***rear***) contienen los índices del primer y último elemento de la cola, respectivamente. Las implementaciones de colas lineales y circulares usan un *array* unidimensional y empiezan con el interface Queue que puede ver en el siguiente listado:

```
// Queue.java
package com.javajeff.cds;
public interface Queue {
    void insert (Object o);
    boolean isEmpty ();
    boolean isFull ();
    Object remove ();
}
```

Queue declara cuatro métodos para almacenar un dato, determinar si la cola está vacía, determinar si la cola está llena y recuperar/borrar un dato de la cola. Llame a estos métodos (y a un constructor) para trabajar con cualquier implementación de Queue.

El siguiente listado presenta una implementación de Queue de una cola lineal basada en un *array* unidimensional:

```
// ArrayLinearQueue.java
package com.javajeff.cds;

public class ArrayLinearQueue implements Queue {
    private int front = -1, rear = -1;
    private Object [] queue;

    public ArrayLinearQueue (int maxElements) {
        queue = new Object [maxElements];
    }

    public void insert (Object o) {
        if (rear == queue.length - 1)
            throw new FullQueueException ();
        queue [++rear] = o;
    }

    public boolean isEmpty () {
        return front == rear;
    }

    public boolean isFull () {
        return rear == queue.length - 1;
    }

    public Object remove () {
        if (front == rear)
            throw new EmptyQueueException ();
        return queue [++front];
    }
}
```

CAPÍTULO 2

RECURSIÓN

PROPÓSITO DEL CAPÍTULO

Al finalizar el estudio de este capítulo, el estudiante deberá estar en capacidad de programar algoritmos recursivos, lo que le permitirá escribir programas sencillos que resuelven problemas complejos. Si bien la recursión no da lugar a soluciones muy eficientes, sí ofrece soluciones ``robustas'' que se pueden utilizar como prototipo para luego desarrollar y comprobar algoritmos más rápidos.

OBJETIVOS DE APRENDIZAJE

Al finalizar el estudio de este tema, usted deberá estar en capacidad de:

- Explicar el concepto de recursión.
- Explicar diferentes tipos de recursión.
- Programar algoritmos recursivos.

GUÍA DE LECTURAS

Para lograr los objetivos anteriores, se le sugiere seguir la siguiente guía:

Subtema	Capítulo	Página
¿Qué es la recursión?	7	165 - 166
Recursión Básica	7	170 - 179
Aplicaciones numéricas	7	180 - 185
Algoritmos divide y vencerás	7	188 - 195
Programación dinámica	7	197 - 200

COMENTARIOS GENERALES

Una vez que usted ha realizado las lecturas anteriores, analice cuidadosamente los comentarios siguientes, ya que con ellos se pretende enfatizar o ampliar algunos contenidos importantes del capítulo. Existen varios aspectos importantes que debemos señalar, antes de entrar al desarrollo de los subtemas.

Un método parcialmente definido en términos de sí mismo recibe el nombre de **recursivo**. Como muchos otros lenguajes, Java soporta métodos recursivos. La recursión, consistente en el uso de métodos recursivos, es una herramienta de programación potente que en muchos casos puede producir algoritmos cortos y eficientes.

Un método recursivo es un método que, directa o indirectamente, se hace una llamada a sí mismo. Esto puede parecer un circulo vicioso: ¿Cómo un método F puede resolver un problema llamándose a sí mismo? La clave está en que el método F se llama a sí mismo, pero con instancias diferentes, más simples, en algún sentido adecuado.

Sin embargo, es importante darse cuenta de que la recursión no siempre es apropiada. El inconveniente práctico es que las llamadas recursivas consumen tiempo y limitan el valor de n para el cual se puede ejecutar el programa. Una buena regla es que nunca debe utilizarse la recursión en sustitución de un simple bucle.

Una técnica importante de resolución de problemas que hace uso de la recursión es la técnica de **divide y vencerás**. Los algoritmos divide y vencerás son algoritmos recursivos que constan de dos partes:

- **Dividir:** Se resuelven recursivamente problemas más pequeños (excepto, por supuesto, los casos base).

- **Vencer:** La solución al problema original se consigue a partir de las soluciones a los subproblemas.

La programación dinámica resuelve los subproblemas generados por un planteamiento recursivo, de forma no recursiva guardando los valores computados en una tabla.

GLOSARIO

En esta área usted encontrará un glosario donde se incluyen términos y la definición de cada uno para que le sea más fácil comprender el tema de estudio.

Algoritmo divide y vencerás: Tipo de algoritmo recursivo, generalmente muy eficiente. La recursión es la parte “divide”, y la combinación de las soluciones recursivas es el “vencerás”.

Caso base: Instancia que se puede resolver sin hacer llamadas recursivas. Toda llamada recursiva debe hacer progresos hacia un caso base.

Encriptación: Esquema de codificación usado en la transmisión de mensajes para que no puedan ser leídos por otras personas.

Método recursivo: Método que, directa o indirectamente, se hace una llamada a sí mismo.

Programación dinámica: Técnica que evita la explosión recursiva guardando respuestas en una tabla.

EJERCICIOS SUGERIDOS

Para este tema, se sugiere que usted realice los siguientes ejercicios de su libro de texto:

7.9 Escriba en Java la rutina de Fibonacci.

7.22 Proporcione un algoritmo para el problema del cambio que calcule el número de formas distintas de devolver K unidades de cambio.

RESOLUCIÓN DE EJERCICIOS SUGERIDOS

7.9. *Escriba en Java la rutina de Fibonacci.*

```
public class Fibonacci {
   private int[] fibo;

   public Fibonacci() {
      fibo = new int[20];
   }

   public Fibonacci(int range) {
      if (range > 0){
         fibo = new int[range];
      }
   }
```

```
    public void fibonacci() {
        for(int i=0;i<fibo.length;i++){
            fibonaccis(i);
        }
    }

    private void fibonaccis(int n) {
        if (n == 0 || n == 1){
        fibo[n] = 1;
        }
        else if (n >= 3) {
            fibo[n] = fibo[n-1] + fibo[n-2];
        }
    }

    public void print_secuence() {
        for(int i=0;i<fibo.length;i++){
            System.out.println(fibo[i]);
        }
    }

    public int[] get_secuence() {
        return fibo;
    }

    public static void main (String[] args) {
        Fibonacci f = new Fibonacci();
        f.fibonacci();
        f.print_secuence();
    }
}
```

7.22 *Proporcione un algoritmo para el problema del cambio que calcule el número de formas distintas de devolver K unidades de cambio.*

El problema del cambio consiste en disponer de un sistema monetario compuesto en este caso por monedas de 500, 200, 100, 50, 25, 10, 5, y 1 pesetas, devolver una cantidad n de dinero utilizando el menor número de monedas posible. Disponemos de una cantidad ilimitada de monedas de cada tipo.

```
package cambio;

import java.io.*;
```

```
public class Cambio {

  static int devolver;
  static int[] valores = {500, 200, 100, 50, 25, 10, 5, 1};
  static int[] caja = {3, 2, 2, 5, 6, 3, 10, 6};// Añadido 30 - 11 - 2006 por --~~~~
  static int[] cantidades = {0, 0, 0, 0, 0, 0, 0, 0};
  static BufferedReader teclado = new BufferedReader(new InputStreamReader(
      System.in));

  /**
   * Método que halla la solución final.
   * Rellena el array de cantidades en función de la cantidad que hay que devolver
   *
   * @param devolver El importe que hay que devolver
   */
  public static void calcular(int devolver) {
    int i = 0;
    while (devolver > 0) {
      if ((valores[i] <= devolver) && (caja[i] > 0) /*Añadido al if 30 - 11 - 2006 por --
~~~~*/) {
        devolver -= valores[i];
        caja[i]--; // Añadido 30 - 11 - 2006 por --~~~~
        cantidades[i]++;
      }
      else {
        i++;
      }
    }
  }

  /**
   * Método auxiliar empleado para pedir al usuario el importe que hay que devolver
   * y validar si es una cantidad de dinero correcta
   * @return Devuelve el importe introducido por el usuario
   */
  private static int pedirDatos() {
    int importe = 0;
    System.out.println(" Introduce importe: ");
    try {
      importe = Integer.parseInt(teclado.readLine());
    } catch (IOException e) {

    }
    return importe;
  }
```

```
  /**
   * Método main
   * @param args
   */
  public static void main(String[] args) {
    Cambio cambio1 = new Cambio();
    devolver = pedirDatos();
    calcular(devolver);
    for (int i = 0; i < cantidades.length; i++) {
      System.out.println(valores[i] + " - " + cantidades[i]);
    }
  }

}
```

CAPÍTULO 3

ALGORITMOS DE ORDENACIÓN

PROPÓSITOS DEL CAPÍTULO

Al finalizar el estudio de este capítulo el estudiante deberá estar en capacidad de explicar un algoritmo como fórmula matemática que resuelve un problema. Aplicado a los buscadores, una vez dado un valor numérico a ciertos factores (como la importancia de una página, las veces que se cita una palabra, dónde está situada esa palabra en el contexto de la página), los algoritmos se utilizan para hallar un resultado final numérico que sirva para ordenar las páginas web.

OBJETIVOS DE APRENDIZAJE

Al finalizar el estudio de este tema, el estudiante deberá estar en capacidad de:

- Explicar la ordenación y los diferentes algoritmos de ordenación.
- Explicar cómo crear mejores ordenadores para la búsqueda de información.
- Explicar cuál método de ordenación es mejor para un problema determinado.

GUÍA DE LECTURAS

Para lograr los objetivos anteriores, se le sugiere seguir la siguiente guía:

Subtema	Capítulo	Páginas
¿Por qué es importante la ordenación?	8	213 - 214
Análisis de la ordenación por inserción y otras ordenaciones simples	8	215 - 216
Shellsort	8	217 - 221
Mergesort	8	221 - 225
Quicksort	8	225 - 227 236 - 237
Ordenación por selección rápida	8	238 - 239

COMENTARIOS GENERALES

Una vez que usted ha realizado las lecturas anteriores, analice cuidadosamente los comentarios siguientes, ya que con ellos se pretende enfatizar o ampliar algunos contenidos importantes del capítulo. Existen varios aspectos importantes que debemos señalar, antes de entrar al desarrollo de los subtemas.

La **ordenación** es una aplicación fundamental en computación. La mayoría de los datos producidos por un programa están ordenados de alguna manera, y muchos de los cómputos son eficientes porque invocan internamente a un método de ordenación. En consecuencia, la ordenación es muy probablemente la operación más importante y mejor estudiada en computación.

Algunas ordenaciones se pueden realizar directamente en la memoria principal. Dependiendo de si el número de elementos por ordenar sea moderadamente pequeño, si no puede realizarse la ordenación en memoria principal, se utiliza el disco o la cinta como apoyo al ordenamiento.

El método de ordenación más simple, denominado **ordenación por inserción**, es aquel que está compuesto por bucles anidados, cada uno de los cuales puede realizar n iteraciones.

El Shellsort es el primer algoritmo que mejoró de forma sustancial la ordenación por inserción. Es un algoritmo subcuadrático que funciona bien en la práctica y es fácil de implementar. El rendimiento del Shellsort depende en gran medida de la secuencia de incrementos en que se basa.

El Mergesort es un algoritmo que se basa en divide y vencerás, el cual resuelve recursivamente dos problemas con la mitad de tamaño.

El algoritmo Mergesort consta de tres pasos:

1. Si el número de elementos por ordenar es cero o uno, acaba.

2. Ordenar recursivamente las dos mitades del vector.

3. Mezclar las dos mitades ordenadas en un vector ordenado.

El Quicksort es el algoritmo de ordenación más rápido conocido. Su rapidez se debe principalmente a un bucle interno muy ajustado y altamente optimizado. Tiene un rendimiento cuadrático en peor caso, pero este caso puede hacerse estadísticamente improbable con poco esfuerzo.

El elemento básico de este algoritmo es el denominado **pivote**, el cual divide los elementos del vector en dos grupos: los menores y los mayores que él.

Otro elemento es denominado **participación**, el cual coloca cada elemento excepto el pivote en uno de los posibles grupos.

El problema de selección rápida consiste en encontrar el k-enésimo menor elemento. La selección rápida precisa una única llamada recursiva, en comparación con las dos que hace Quicksort. El tiempo de ejecución es lineal en promedio.

GLOSARIO

En esta área usted encontrará un glosario donde se incluyen términos y la definición de cada uno para que le sea más fácil comprender el tema de estudio.

Ordenación: Algoritmo que toma decisiones sobre el orden utilizando sólo comparaciones.

Mergesort: Método de ordenación que utiliza divide y vencerás para obtener una ordenación O(N Log N).

Ordenación por disminución de intervalos: Otro nombre para el Shellsort.

Partición con la mediana de tres: Se utiliza como pivote la mediana del primer elemento, el central y el último. Esto significa la fase de partición de Quicksort.

Partición: Paso de Quicksort que coloca cada elemento del vector, excepto el pivote, en uno de dos posibles grupos, uno formado por los elementos menores que el pivote, y otro formado por los mayores que él.

Pivote: En Quicksort, el elemento que sirve para dividir el vector en dos grupos: uno con los elementos menores que él y otro con los elementos mayores.

Quicksort: Algoritmo divide y vencerás rápido, cuando se implementa apropiadamente. Es, de hecho, el algoritmo de ordenación basado en comparaciones más rápido en la práctica.

Selección: Proceso para buscar el k-enésimo menor elemento de un vector.

Selección rápida: Algoritmo utilizado para realizar una selección, similar a Quicksort, pero con una sola llamada recursiva. El tiempo de ejecución en promedio es lineal.

Shellsort: Algoritmo subcuadrático que funciona bien en la práctica, y es simple de implementar. El rendimiento de Shellsort depende en gran medida de la secuencia de incrementos en que se base. Conlleva un análisis desafiante, todavía no completamente resuelto.

EJERCICIO SUGERIDO

Para este tema, se sugiere que usted realice el siguiente ejercicio de su libro de texto:

8.19 Implemente los métodos de ordenación de Shellsort y Quicksort.

RESOLUCIÓN DE EJERCICIO SUGERIDO

8.19. *Implemente los métodos de ordenación de Shellsort y Quicksort.*

Shellsort

```
// shellsort.java
// demonstrates shellsort
// to run this program: C>java Shellsortapp
//--------------------------------------------------------------
class ArraySh
  {
  private long[] theArray;          // ref to array theArray
  private int nElems;               // number of data items
//--------------------------------------------------------------
  public ArraySh(int max)           // constructor
    {
    theArray = new long[max];   // create the array
    nElems = 0;                       // no items yet
    }
//--------------------------------------------------------------
  public void insert(long value)    // put element into array
    {
    theArray[nElems] = value;      // insert it
    nElems++;                         // increment size
    }
//--------------------------------------------------------------
  public void display()             // displays array contents
    {
    System.out.print("A=");
    for(int j=0; j<nElems; j++)    // for each element,
      System.out.print(theArray[j] + " ");  // display it
    System.out.println("");
```

```
    }
//--------------------------------------------------------------
  public void shellsort()
    {
    int inner, outer;
    long temp;

    int h = 1;                  // find initial value of h
    while(h <= nElems/3)
      h = h*3 + 1;              // (1, 4, 13, 40, 121, ...)

    while(h>0)                  // decreasing h, until h=1
      {
                                // h-sort the file
      for(outer=h; outer<nElems; outer++)
        {
        temp = theArray[outer];
        inner = outer;
                                // one subpass (eg 0, 4, 8)
        while(inner > h-1 && theArray[inner-h] >=  temp)
          {
          theArray[inner] = theArray[inner-h];
          inner -= h;
          }
        theArray[inner] = temp;
        }  // end for
      h = (h-1) / 3;            // decrease h
      }  // end while(h>0)
    }  // end shellsort()
//--------------------------------------------------------------
  }  // end class ArraySh
////////////////////////////////////////////////////////////////
class ShellSortApp
  {
  public static void main(String[] args)
    {
    int maxSize = 10;                   // array size
    ArraySh arr;
    arr = new ArraySh(maxSize);    // create the array

    for(int j=0; j<maxSize; j++)      // fill array with
      {                         // random numbers
      long n = (int)(java.lang.Math.random()*99);
      arr.insert(n);
      }
```

```
    arr.display();                  // display unsorted array
    arr.shellSort();                // shell sort the array
    arr.display();                  // display sorted array
    }                               // end main()
  }                                 // end class ShellSortApp
////////////////////////////////////////////////////////////////
```

Quicksort

```
import java.io.*;
import java.util.*;

public class Quicksort
{
  public static void swap (int A[], int x, int y)
  {
    int temp = A[x];
    A[x] = A[y];
    A[y] = temp;
  }

  // Reorganizes the given list so all elements less than the first are
  // before it and all greater elements are after it.
  public static int partition(int A[], int f, int l)
  {
    int pivot = A[f];
    while (f < l)
    {
      if (A[f] == pivot || A[l] == pivot)
      {
        System.out.println("Only distinct integers allowed - C321");
        System.out.println("students should ignore this if statement");
        System.out.exit(0);
      }
      while (A[f] < pivot) f++;
      while (A[l] > pivot) l--;
      swap (A, f, l);
    }
    return f;
  }

  public static void Quicksort(int A[], int f, int l)
  {
    if (f >= l) return;
    int pivot_index = partition(A, f, l);
    Quicksort(A, f, pivot_index);
```

```
    Quicksort(A, pivot_index+1, l);
  }

  // Usage: java Quicksort [integer] ...
  // All integers must be distinct
  public static void main(String argv[])
  {
    int A[] = new int[argv.length];
    for (int i=0 ; i < argv.length ; i++)
      A[i] = Integer.parseInt(argv[i]);

    Quicksort(A, 0, argv.length-1);

    for (int i=0 ; i < argv.length ; i++) System.out.print(A[i] + " ");
    System.out.println();
  }
}
```

CAPÍTULO 4

GRAFOS Y CAMINOS

PROPÓSITO DEL CAPÍTULO

El estudio de este capítulo deberá contribuir a la formación y el desarrollo del razonamiento científico. Proveerá al estudiante de unas mínimas capacidades de abstracción, concreción, concisión, imaginación, intuición, razonamiento, crítica, objetividad, síntesis y precisión. Capacitará al alumno para modelar matemáticamente una situación, así como para resolver problemas con técnicas matemáticas. Facilitará una base primaria en el análisis y diseño de algoritmos, en lo que respecta a búsquedas en profundidad y anchura, algoritmos para la ordenación, algoritmos voraces.

OBJETIVOS DE APRENDIZAJE

Al finalizar el estudio de este tema, el estudiante deberá estar en capacidad de:

- Explicar las nociones y herramientas elementales propias de la teoría de grafos, y su aplicación en la resolución de una amplia variedad de problemas cotidianos.

- Explicar la teoría de grafos para modelar y resolver problemas de la vida cotidiana.

GUÍA DE LECTURAS

Para lograr los objetivos anteriores, se le sugiere seguir la siguiente guía:

Subtema	Capítulo	Páginas
Definiciones	14	353 - 355
Problema del camino mínimo sin pesos	14	366 - 370
Problema de los caminos mínimos con pesos positivos	14	371 - 375
Problema de caminos en grafos acíclicos	14	379 - 382

COMENTARIOS GENERALES

Una vez que usted ha realizado las lecturas anteriores, analice cuidadosamente los comentarios siguientes, ya que con ellos se pretende enfatizar o ampliar algunos contenidos importantes del capítulo. Existen varios aspectos importantes que debemos señalar, antes de entrar al desarrollo de los subtemas.

Un **grafo dirigido** es un conjunto de vértices (V) y un conjunto de arcos (A).

A los **vértices** se les denomina también **nodos** o **puntos**.

Los **arcos** son llamados también **arcos dirigidos** o **líneas dirigidas**. Un arco es un par ordenado de vértices (v, w); V es la cola y W es la cabeza del arco. El formato es V? W, y se dice que W es adyacente a V.

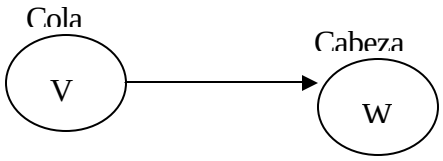


Figura No. 1

Los vértices de un grafo dirigido pueden usarse para representar objetos y los arcos relaciones entre los objetos.

El **camino** es la ruta de vértices. La longitud de un camino es el número de arcos de un vértice a otro.

Los **grafos dirigidos etiquetado**s es cuando los vértices pueden tener, a la vez, un nombre y una etiqueta.

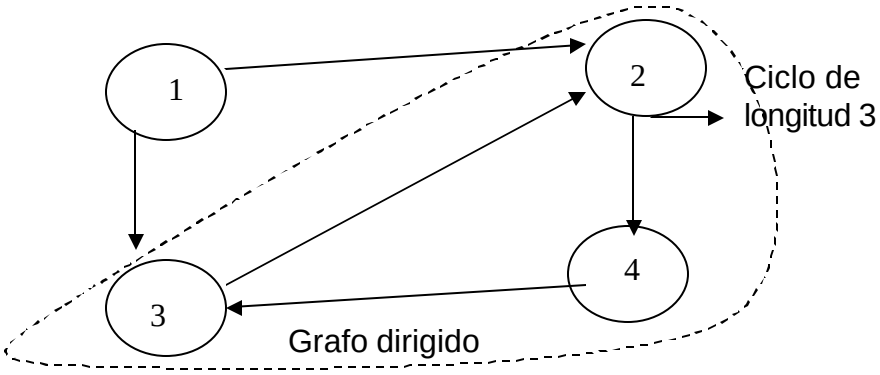


Figura No. 2

Representaciones de grafos dirigidos

Pueden utilizarse varias estructuras de datos, dependiendo de las operaciones que se aplicarán a los vértices y a los arcos del grafo, como puede ser una matriz de adyacencia o una lista de adyacencia.

En la matriz de adyacencia los elementos son booleanos, es decir, verdadero sí y sólo si existe un arco que vaya del vértice i al j, en donde 1 equivale a verdadero y 0 a falso.

Ejemplo

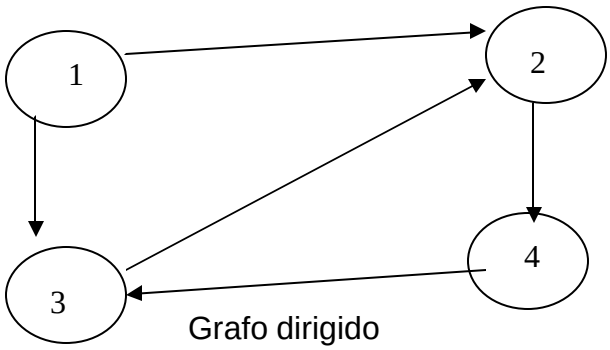


Figura No. 3

Matriz de adyacencia

	1	2	3	4
1	0	1	1	0
2	0	0	0	1

3	0	1	0	0
4	0	0	1	0

Figura No. 4

Como la diagonal es negativa, se dice que la matriz es simétrica. Si alguna de las celdas de la diagonal es 1, se dice que la matriz de adyacencia es asimétrica.

Lista de adyacencia

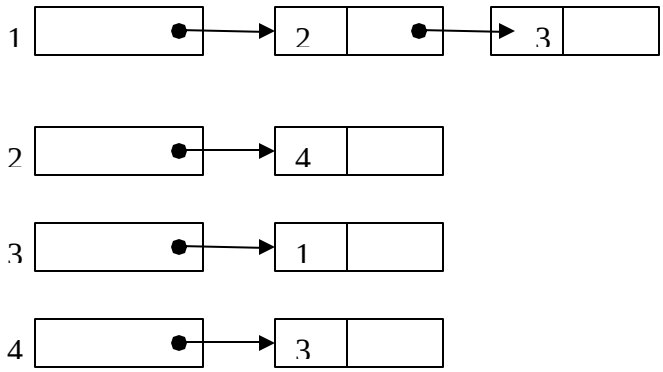


Figura No. 5

(Los ejemplos de programas pueden localizarse en las páginas 204 y 205 de su libro de texto.)

Caminos mínimos con pesos positivos

El problema es determinar el costo del camino más corto del origen a todos los demás vértices de V, donde la longitud de los caminos es la suma de los costos de los arcos del camino.

El algoritmo para resolver este problema es el Dijktra, que es un conjunto de reglas que permiten obtener un resultado determinado a partir de ciertas reglas definidas. Ha de tener las siguientes características: legible, correcto, modular, eficiente, estructurado, no ambiguo y, de ser posible, se ha de desarrollar en el menor tiempo posible.

GLOSARIO

En esta área usted encontrará un glosario donde se incluyen términos y la definición de cada uno para que le sea más fácil comprender el tema de estudio.

Algoritmo Dyjkstra: Algoritmo que resuelve el problema de los caminos mínimos con peso.

Análisis de camino crítico: Forma de análisis utilizada en la planificación de tareas asociadas a un proyecto.

Camino: Secuencia de vértices conectados por aristas.

Camino simple: Camino cuyos vértices son todos distintos, excepto el primero y el último, que pueden ser iguales.

Ciclo: Grafo dirigido, un camino que empieza y termina en el mismo vértice, y que contiene al menos una arista.

Grafo de entrada: Número de aristas entrantes a un vértice.

Grafo: Está formado por un conjunto de vértices y un conjunto de aristas que conectan vértices.

Grafo dirigido: Grafo cuyas aristas son pares ordenados de vértices.

Grafo dirigido acíclico (GDA): Tipo de grafo dirigido que no contiene ciclos.

Lista de adyacencia: Vector de listas utilizado para representar un grafo. Utiliza un espacio lineal.

Longitud de un camino: Número de aristas de un camino.

Matriz de adyacencia: Representación de un grafo mediante una matriz, que utiliza un espacio cuadrático.

EJERCICIO SUGERIDO

Para este tema, se sugiere que usted realice el siguiente ejercicio de su libro de texto:

Implementación del algoritmo de Dijkstra.

RESOLUCIÓN DE EJERCICIO SUGERIDO

Implemente el algoritmo de Dijkstra.

```
import java.awt.*;
//-------------------------------------------------------------------------/
//  AlgoritmoD.java
//
//
//  Applet para ejecutar el algoritmo de Dijkstra en un grafo dirigido
//  y encontrar el camino mas corto a todos los nodos.
//-------------------------------------------------------------------------/

public class AlgoritmoD extends java.applet.Applet {
   GraphCanvas grafocanvas = new GraphCanvas(this);
   Options options = new Options(this);
   Documentacion documentacion = new Documentacion();
   public void init() {
            setLayout(new BorderLayout(10, 10));
            add("Center", grafocanvas);
            add("North", documentacion);
            add("East", options);
   }
   public Insets insets() {
            return new Insets(10, 10, 10, 10);
   }
   public void lock() {
            grafocanvas.lock();
            options.lock();
   }
   public void unlock() {
            grafocanvas.unlock();
            options.unlock();
   }
}

class Options extends Panel {
// opciones a un lado de la pantalla
   Button b1 = new Button("limpiar");
   Button b2 = new Button("ejecutar");
   Button b3 = new Button("paso");
   Button b4 = new Button("inicializar");
   Button b5 = new Button("ejemplo");
   Button b6 = new Button("salir");
   AlgoritmoD parent;
```

```
boolean Locked=false;

Options(AlgoritmoD myparent) {
        parent = myparent;
        setLayout(new GridLayout(6, 1, 0, 10));
        add(b1);
        add(b2);
        add(b3);
        add(b4);
        add(b5);
        add(b6);
}
public boolean action(Event evt, Object arg) {
        if (evt.target instanceof Button) {
          if (((String)arg).equals("paso")) {
            if (!Locked) {
              b3.setLabel("siguiente paso");
              parent.grafocanvas.stepalg();
            }
            else parent.documentacion.doctext.showline("cerrado");
          }
          if (((String)arg).equals("siguiente paso"))
            parent.grafocanvas.nextstep();
          if (((String)arg).equals("inicializar")) {
            parent.grafocanvas.inicializar();
            b3.setLabel("paso");
            parent.documentacion.doctext.showline("referencia");
          }
          if (((String)arg).equals("limpiar")) {
            parent.grafocanvas.limpiar();
            b3.setLabel("paso");
            parent.documentacion.doctext.showline("referencia");
          }
          if (((String)arg).equals("ejecutar")) {
            if (!Locked)
              parent.grafocanvas.runalg();
            else parent.documentacion.doctext.showline("cerrado");
          }
          if (((String)arg).equals("ejemplo")) {
            if (!Locked)
              parent.grafocanvas.showejemplo();
            else parent.documentacion.doctext.showline("cerrado");
          }
          if (((String)arg).equals("salir")) {
            System.exit(0);
```

```
                }
            }
            return true;
    }

    public void lock() {
            Locked=true;
    }
    public void unlock() {
            Locked=false;
            b3.setLabel("paso");
    }
}

class Documentacion extends Panel {
// Documentacion arriba de la pantalla
    DocOptions docopt = new DocOptions(this);
    DocText doctext = new DocText();
    Documentacion() {
            setLayout(new BorderLayout(10, 10));
            add("West", docopt);
            add("Center", doctext);
    }
}

class DocOptions extends Panel {
    Choice doc = new Choice();
    Documentacion parent;

    DocOptions(Documentacion myparent) {
            setLayout(new GridLayout(2, 1, 5, 0));
            parent = myparent;
            add(new Label("DOCUMENTACION:"));
            doc.addItem("dibujar nodos");
            doc.addItem("remover nodos");
            doc.addItem("mover nodos");
            doc.addItem("el nodo_inicial");
            doc.addItem("dibujar aristas");
            doc.addItem("cambiar pesos");
            doc.addItem("remover aristas");
            doc.addItem("limpiar / inicializar");
            doc.addItem("ejecutar algoritmo");
            doc.addItem("pasar");
            doc.addItem("ejemplo");
            doc.addItem("salir");
```

```
            doc.addItem("referencia");
            add(doc);
    }

    public boolean action(Event evt, Object arg) {
        if (evt.target instanceof Choice) {
                String str=new String(doc.getSelectedItem());
                parent.doctext.showline(str);
        }
            return true;
    }
}

class DocText extends TextArea {
    final String drawnodos = new String("DIBUJAR NODOS:\n"+
            "Dibuje un nodo haciendo click en el mouse.\n\n");
    final String rmvnodos = new String("REMOVER NODOS:\n"+
            "Para remover un nodo presione <ctrl> y haga click en el nodo.\n"+
            "No se puede remover el nodo_inicial.\n"+
            "Seleccione otro nodo_inicial, asi podra remover el nodo.\n\n");
    final String mvnodos = new String("MOVER NODOS\n"+
            "Para mover un nodo presione <Shift>, haga click en el nodo,\ny arrastrelo
a"+
            " su nueva posicion.\n\n");
    final String nodo_inicial = new String("NODO INICIAL:\n"+
            "El nodo_inicial es azul, los otros nodos son grises.\n"+
            "El primer nodo que usted dibuja en la pantalla sera el nodo_inicial.\n"+
            "Para seleccionar otro nodo_inicial presione <ctrl>, haga click en el
nodo_inicial,\n"+
            "y arrastre el mouse a otro nodo.\n"+
            "Para borrar el nodo_inicial, primero seleccione otro nodo_inicial, y
despues"+
            "\nremueva el nodo normalmente.\n\n");
    final String drawaristas = new String("DIBUJAR ARISTAS:\n"+
            "Para dibujar una arista haga click al mouse en un nodo,"+
            "y arrastrelo a otro nodo.\n\n");
    final String peso = new String("CAMBIAR PESOS:\n"+
            "Para cambiar el peso de una arista, haga click en la flecha y \n"+
            "arrastrela sobre la arista.\n\n");
    final String rmvaristas = new String("REMOVER ARISTAS:\n"+
            "Para remover una arista, cambiar su peso a 0.\n\n");
    final String clrreset = new String("BOTON DE<LIMPIAR>: "+
            "Remover el grafo de la pantalla.\n"+
            "BOTON DE<INICIALIZAR>: "+
            "Remover los resultados del algoritmo en el grafo,\n"+
```

```
            " y abrir la pantalla.\n\n");
final String runalg = new String("BOTON DE <EJECUTAR>: "+
            "Ejecuta el algoritmo en el grafo, habra un tiempo\n" +
            "de retraso de +/- 1 segundos entre cada paso.\n"+
            "Para ejecutar el algoritmo mas lento, use <PASO>.\n");
final String paso = new String("BOTON DE <PASO>: " +
            "Recorrer el algoritmo paso a paso.\n");
final String ejemplo = new String("BOTON DE <EJEMPLO>: "+
            "Despliega un grafo en la pantalla.\n"+
            "Usted puede usar <PASO> or <EJECUTAR>\n");
final String exitbutton = new String("BOTON DE <SALIR>: " +
            "Solo funciona si el applet es ejecutado en appletviewer.\n");
final String toclose = new String("ERROR: "+
            "Esta posicion es para cerrar a otro nodo/arista.\n");
final String hecho = new String("El lgoritmo ha terminado, " +
            "siga las aristas naranjas del nodo_inicial a cualquier nodo "+
            "para obtener\nel mas corto camino al " +
            "nodo. La longitud del camino se escribe en el nodo.\n" +
        "presione <INICIALIZAR> para inicializar el grafo, y liberar la pantalla.");
final String alguno = new String("El algoritmo ha terminado, " +
            "siga las aristas naranjas del nodo_inicial a cualquier nodo "+
            "para obtener\nel mas corto camino al " +
            "nodo. La longitud del camino se escribe en el nodo.\n" +
        "No hay caminos del nodo_inicial a ningun otro nodo.\n" +
        "presione <INICIALIZAR> para inicializar el grafo, y liberar la pantalla.");
final String ninguno = new String("El algoritmo ha terminado, " +
            "no hay nodos alcanzables desde el nodo inicial.\n"+
            "presione <INICIALIZAR> para inicializar el grafo, y liberar la pantalla.");

final String maxnodos = new String("ERROR: "+
            "Maximo numero de nodos alcanzado!\n\n");
final String info = new String("DOCUMENTACION:\n"+
            "Usted puede ver toda la documentacion u obtener documentacion\n"+
            "de algo especifico "+
            "seleccionando el item a la izquierda.\nSeleccionar <Referencia> "+
            "lo regresa "+
            " al texto.\n\n");
final String cerrado = new String("ERROR: "+
            "Teclado/mouse cerrado para esta accion.\n"+
            "Presione <SIGUIENTE PASO> o <INICIALIZAR>.\n");
final String doc = info + drawnodos + rmvnodos + mvnodos +
                        nodo_inicial + drawaristas + peso + rmvaristas +
                        clrreset + runalg + paso + ejemplo + exitbutton;
DocText() {
        super(5, 2);
```

```
            setText(doc);
    }

    public void showline(String str) {
            if (str.equals("dibujar nodos"))  setText(drawnodos);
            else if (str.equals("remover nodos"))  setText(rmvnodos);
            else if (str.equals("mover nodos"))  setText(mvnodos);
            else if (str.equals("el nodo_inicial")) setText(nodo_inicial);
            else if (str.equals("dibujar aristas")) setText(drawaristas);
            else if (str.equals("cambiar pesos")) setText(peso);
            else if (str.equals("remover aristas")) setText(rmvaristas);
            else if (str.equals("limpiar / inicializar")) setText(clrreset);
            else if (str.equals("ejecutar algoritmo")) setText(runalg);
            else if (str.equals("pasar")) setText(paso);
            else if (str.equals("ejemplo"))  setText(ejemplo);
       else if (str.equals("salir"))  setText(exitbutton);
            else if (str.equals("referencia"))  setText(doc);
            else if (str.equals("toclose"))  setText(toclose);
            else if (str.equals("hecho"))  setText(hecho);
            else if (str.equals("cerrado"))  setText(cerrado);
            else if (str.equals("maxnodos"))  setText(maxnodos);
       else if (str.equals("ninguno"))  setText(ninguno);
       else if (str.equals("alguno"))  setText(alguno);
            else setText(str);
    }
}

class GraphCanvas extends Canvas implements Runnable {
// área de dibujo del grafo

    final int MAXNODOS = 20;
    final int MAX = MAXNODOS+1;
    final int NODOSIZE = 26;
    final int NODORADIX = 13;
    final int DIJKSTRA = 1;

    // informacion basica del grafo
    Point nodo[] = new Point[MAX];         // nodo
    int peso[][] = new int[MAX][MAX];     // peso de arista
    Point arista[][] = new Point[MAX][MAX];  // posició actual de la flecha
    Point startp[][] = new Point[MAX][MAX]; // punto inicial
    Point endp[][] = new Point[MAX][MAX];   // y final de arista
    float dir_x[][] = new float[MAX][MAX];  // dirección de arista
    float dir_y[][] = new float[MAX][MAX];  // dirección de arista
```

```
// información del grafo al ejecutar el algoritmo
boolean algedge[][] = new boolean[MAX][MAX];
int dist[] = new int[MAX];
int finaldist[] = new int[MAX];
Color colornodo[] = new Color[MAX];
 boolean changed[] = new boolean[MAX];   // indica cambio de distancia durante el
                                                    // algoritmo
int numchanged =0;
int neighbours=0;
int paso=0;

// información usada por el algoritmo para encontrar el siguiente
// nodo con mínima distancia
int mindist, minnodo, minstart, minend;
int numnodos=0;      // número de nodos
int emptyspots=0;    // lugares vacios en el arreglo nodo[] (por borrado de nodos)
int startgrafo=0;    // comienzo de grafo
int hitnodo;         // click del mouse en o cerca del nodo
int nodo1, nodo2;    // número de nodos envueltos in la accion
Point thispoint=new Point(0,0); // posición actual del mouse
Point oldpoint=new Point(0, 0); // posición previa del nodo
// acción actual
boolean newarista = false;
boolean movearista = false;
boolean movestart = false;
boolean deletenodo = false;
boolean movenodo = false;
boolean performalg = false;
boolean clicked = false;
// fonts
Font roman= new Font("TimesRoman", Font.BOLD, 12);
Font helvetica= new Font("Helvetica", Font.BOLD, 15);
FontMetrics fmetrics = getFontMetrics(roman);
int h = (int)fmetrics.getHeight()/3;
// buffer doble
private Image offScreenImage;
private Graphics offScreenGraphics;
private Dimension offScreenSize;

// opción de ejecutar
Thread algrthm;
// algoritmo actual, (se pueden aniadir mas)
int algoritmo;
// información del algoritmo para ser desplegado en la documentación
String showstring = new String("");
```

```
boolean stepthrough=false;
// cerrar la pantalla mientras se ejecuta el algoritmo
boolean Locked = false;
AlgoritmoD parent;
GraphCanvas(AlgoritmoD myparent) {
        parent = myparent;
        init();
        algoritmo=DIJKSTRA;
        setBackground(Color.white);
}
public void lock() {
// cerrar la pantalla mientras se ejecuta el algoritmo
        Locked=true;
}
public void unlock() {
        Locked=false;
}
public void start() {
        if (algrthm != null) algrthm.resume();
}
public void init() {
        for (int i=0;i<MAXNODOS;i++) {
          colornodo[i]=Color.gray;
          for (int j=0; j<MAXNODOS;j++)
             algedge[i][j]=false;
        }
        colornodo[startgrafo]=Color.blue;
        performalg = false;
}
public void limpiar() {
// remueve grafo de la pantalla
        startgrafo=0;
        numnodos=0;
        emptyspots=0;
        init();
        for(int i=0; i<MAXNODOS; i++) {
          nodo[i]=new Point(0, 0);
          for (int j=0; j<MAXNODOS;j++)
             peso[i][j]=0;
        }
        if (algrthm != null) algrthm.stop();
        parent.unlock();
        repaint();
```

```
 }
 public void inicializar() {
 // inicializa aun grafo despues de ejecutar un algoritmo
          init();
          if (algrthm != null) algrthm.stop();
          parent.unlock();
          repaint();
 }
 public void runalg() {
 // anima el algoritmo
          parent.lock();
          initalg();
          performalg = true;
          algrthm = new Thread(this);
          algrthm.start();
 }
 public void stepalg() {
 // le permite pasar por el algoritmo
          parent.lock();
          initalg();
          performalg = true;
          nextstep();
 }
 public void initalg() {
          init();
          for(int i=0; i<MAXNODOS; i++) {
            dist[i]=-1;
            finaldist[i]=-1;
            for (int j=0; j<MAXNODOS;j++)
          algedge[i][j]=false;
          }
    dist[startgrafo]=0;
          finaldist[startgrafo]=0;
          paso=0;
 }

public void nextstep() {
 // calcula un paso en el algoritmo (encuentra uno más corto
 // camino al siguiente nodo).
    finaldist[minend]=mindist;
          algedge[minstart][minend]=true;
          colornodo[minend]=Color.orange;
    // más información para la documentación
          paso++;
          repaint();
```

```
    }

 public void stop() {
            if (algrthm != null) algrthm.suspend();
    }

public void run() {
            for(int i=0; i<(numnodos-emptyspots); i++){
              nextstep();
              try { algrthm.sleep(2000); }
              catch (InterruptedException e) {}
            }
            algrthm = null;
    }

public void showejemplo() {
    // dibuja un grafo en la pantalla
            int w, h;
            limpiar();
            init();
            numnodos=10;
            emptyspots=0;
            for(int i=0; i<MAXNODOS; i++) {
              nodo[i]=new Point(0, 0);
              for (int j=0; j<MAXNODOS;j++)
                  peso[i][j]=0;
            }
            w=this.size().width/8;
            h=this.size().height/8;
            nodo[0]=new Point(w, h);     nodo[1]=new Point(3*w, h);
            nodo[2]=new Point(5*w, h);   nodo[3]=new Point(w, 4*h);
            nodo[4]=new Point(3*w, 4*h); nodo[5]=new Point(5*w, 4*h);
            nodo[6]=new Point(w, 7*h);   nodo[7]=new Point(3*w, 7*h);
            nodo[8]=new Point(5*w, 7*h); nodo[9]=new Point(7*w, 4*h);
            peso[0][1]=4;  peso[0][3]=85;
            peso[1][0]=74; peso[1][2]=18; peso[1][4]=12;
            peso[2][5]=74; peso[2][1]=12; peso[2][9]=12;
            peso[3][4]=32; peso[3][6]=38;
            peso[4][3]=66; peso[4][5]=76; peso[4][7]=33;
            peso[5][8]=11; peso[5][9]=21;
            peso[6][7]=10; peso[6][3]=12;
            peso[7][6]=2;  peso[7][8]=72;
            peso[8][5]=31; peso[8][9]=78; peso[8][7]=18;
            peso[9][5]=8;
            for (int i=0;i<numnodos;i++)
```

```
            for (int j=0;j<numnodos;j++)
              if (peso[i][j]>0)
                aristaupdate(i, j, peso[i][j]);
          repaint();
 }

public boolean mouseDown(Event evt, int x, int y) {

          if (Locked)
            parent.documentacion.doctext.showline("cerrado");
          else {
            clicked = true;
            if (evt.shiftDown()) {
            // mover un nodo
              if (nodohit(x, y, NODOSIZE)) {
                oldpoint = nodo[hitnodo];
                nodo1 = hitnodo;
                movenodo=true;
              }
            }
            else if (evt.controlDown()) {
            // borrar un nodo
              if (nodohit(x, y, NODOSIZE)) {
                nodo1 = hitnodo;
                if (startgrafo==nodo1) {
                  movestart=true;
                  thispoint = new Point(x,y);
             colornodo[startgrafo]=Color.gray;
                }
                else
                  deletenodo= true;
              }
            }
            else if (aristahit(x, y, 5)) {
            // cambiar peso de una arista
              movearista = true;
              repaint();
            }
            else if (nodohit(x, y, NODOSIZE)) {
            // dibuja una nueva arista
              if (!newarista) {
                newarista = true;
                thispoint = new Point(x, y);
                nodo1 = hitnodo;
              }
```

```
            }
            else if ( !nodohit(x, y, 50) && !aristahit(x, y, 50) )  {
            // dibuja nuevo nodo
              if (emptyspots==0) {
              // toma el siguiente punto disponible en el arreglo
                if (numnodos < MAXNODOS)
                  nodo[numnodos++]=new Point(x, y);
                else  parent.documentacion.doctext.showline("maxnodos");
              }
              else {
              // tomar un punto vacio en el array (de algun nodo borrado previamente)
                int i;
                for (i=0;i<numnodos;i++)
                  if (nodo[i].x==-100) break;
                nodo[i]=new Point(x, y);
                emptyspots--;
              }
            }
            else
            // mouseclick para cerrar a un point r flecha, probablemente un error
              parent.documentacion.doctext.showline("toclose");
            repaint();
          }
          return true;
  }

  public boolean mouseDrag(Event evt, int x, int y) {
          if ( (!Locked) && clicked ) {
            if (movenodo) {
            // mover nodo y ajustar aristas entrando/saliendo del nodo
              nodo[nodo1]=new Point(x, y);
              for (int i=0;i<numnodos;i++) {
                if (peso[i][nodo1]>0) {
                  aristaupdate(i, nodo1, peso[i][nodo1]);
                }
                if (peso[nodo1][i]>0) {
                  aristaupdate(nodo1, i, peso[nodo1][i]);
                }
              }
              repaint();
            }
            else if (movestart || newarista) {
              thispoint = new Point(x, y);
              repaint();
            }
```

```
            else if (movearista) {
              changepeso(x, y);
              repaint();
            }
          }
          return true;
  }

public boolean mouseUp(Event evt, int x, int y) {
          if ( (!Locked) && clicked ) {
            if (movenodo) {
            // mover el nodo si la nueva posicion no esta muy cerca a
            // otro nodo o fuera del panel
              nodo[nodo1]=new Point(0, 0);
              if ( nodohit(x, y, 50) || (x<0) || (x>this.size().width) ||
                                                    (y<0) || (y>this.size().height) ) {
                nodo[nodo1]=oldpoint;
                parent.documentacion.doctext.showline("toclose");
              }
              else nodo[nodo1]=new Point(x, y);
              for (int i=0;i<numnodos;i++) {
                if (peso[i][nodo1]>0)
                  aristaupdate(i, nodo1, peso[i][nodo1]);
                if (peso[nodo1][i]>0)
                  aristaupdate(nodo1, i, peso[nodo1][i]);
              }
              movenodo=false;
            }
            else if (deletenodo) {
              nododelete();
              deletenodo=false;
            }
            else if (newarista) {
              newarista = false;
              if (nodohit(x, y, NODOSIZE)) {
                nodo2=hitnodo;
                if (nodo1!=nodo2) {
                  aristaupdate(nodo1, nodo2, 50);
                  if (peso[nodo2][nodo1]>0) {
                    aristaupdate(nodo2, nodo1, peso[nodo2][nodo1]);
                  }
                  parent.documentacion.doctext.showline("cambiar pesos");
                }
                else System.out.println("zelfde");
              }
```

```
            }
            else if (movearista) {
              movearista = false;
              if (peso[nodo1][nodo2]>0)
                changepeso(x, y);
            }
            else if (movestart) {
            // si la nueva posicion es un nodo, este nodo es el nodo_inicial
              if (nodohit(x, y, NODOSIZE))
                startgrafo=hitnodo;
              colornodo[startgrafo]=Color.blue;
              movestart=false;
            }
            repaint();
          }
          return true;
  }

public boolean nodohit(int x, int y, int dist) {
  // checa si hay click sobre un nodo
          for (int i=0; i<numnodos; i++)
            if ( (x-nodo[i].x)*(x-nodo[i].x) +
                                        (y-nodo[i].y)*(y-nodo[i].y) < dist*dist ) {
              hitnodo = i;
              return true;
            }
          return false;
  }

 public boolean aristahit(int x, int y, int dist) {
  // checa si hay click sobre una arista
          for (int i=0; i<numnodos; i++)
            for (int j=0; j<numnodos; j++) {
              if ( ( peso[i][j]>0 ) &&
                            (Math.pow(x-arista[i][j].x, 2) +
                             Math.pow(y-arista[i][j].y, 2) < Math.pow(dist, 2) ) ) {
                nodo1 = i;
                nodo2 = j;
                return true;
              }
            }
          return false;
  }
```

```
public void nododelete() {
   // borra un nodo y las aristas que entran/salen del nodo
            nodo[nodo1]=new Point(-100, -100);
            for (int j=0;j<numnodos;j++) {
              peso[nodo1][j]=0;
              peso[j][nodo1]=0;
            }
            emptyspots++;
   }

public void changepeso(int x, int y)
{
   // cambia el peso de una arista, cuando el usuario arrastra
   // la flecha sobre la arista que conecta el nodo1 al nodo2.
            // direccion de la arista
            int diff_x = (int)(20*dir_x[nodo1][nodo2]);
            int diff_y = (int)(20*dir_y[nodo1][nodo2]);
            // dependiendo de la dirección de la arista , sigue el x, o el y
            // del mouse mientras la flecha se arrastra
            boolean siga_x=false;
              if (Math.abs(endp[nodo1][nodo2].x-startp[nodo1][nodo2].x) >
                        Math.abs(endp[nodo1][nodo2].y-startp[nodo1][nodo2].y) ) {
              siga_x = true;
            }
            // encuentra la nueva posición de la flecha, y calcula
            // el peso correspondiente
            if (siga_x) {
               int hbound = Math.max(startp[nodo1][nodo2].x,
                                          endp[nodo1][nodo2].x)-Math.abs(diff_x);
               int lbound = Math.min(startp[nodo1][nodo2].x,
                                          endp[nodo1][nodo2].x)+Math.abs(diff_x);
               // la arista debe quedarse entre los nodos
               if (x<lbound) { arista[nodo1][nodo2].x=lbound; }
               else if (x>hbound) { arista[nodo1][nodo2].x=hbound; }
               else arista[nodo1][nodo2].x=x;
               arista[nodo1][nodo2].y=
                        (arista[nodo1][nodo2].x-startp[nodo1][nodo2].x) *
                           (endp[nodo1][nodo2].y-startp[nodo1][nodo2].y)/
                              (endp[nodo1][nodo2].x-startp[nodo1][nodo2].x) +
                        startp[nodo1][nodo2].y;
               // nuevo peso
               peso[nodo1][nodo2]=
                        Math.abs(arista[nodo1][nodo2].x-startp[nodo1][nodo2].x-diff_x)*
                                 100/(hbound-lbound);
            }
```

```
            // hacer lo mismo si sigue y
            else {
               int hbound = Math.max(startp[nodo1][nodo2].y,
                                     endp[nodo1][nodo2].y)-Math.abs(diff_y);
               int lbound = Math.min(startp[nodo1][nodo2].y,
                                     endp[nodo1][nodo2].y)+Math.abs(diff_y);
               if (y<lbound) { arista[nodo1][nodo2].y=lbound; }
               else if (y>hbound) { arista[nodo1][nodo2].y=hbound; }
               else arista[nodo1][nodo2].y=y;
               arista[nodo1][nodo2].x=
                        (arista[nodo1][nodo2].y-startp[nodo1][nodo2].y) *
                           (endp[nodo1][nodo2].x-startp[nodo1][nodo2].x)/
                                 (endp[nodo1][nodo2].y-startp[nodo1][nodo2].y) +
                        startp[nodo1][nodo2].x;
               peso[nodo1][nodo2]=
                        Math.abs(arista[nodo1][nodo2].y-startp[nodo1][nodo2].y-diff_y)*
                                 100/(hbound-lbound);
            }
      }

   public void aristaupdate(int p1, int p2, int w) {
      // hacer una arista nueva del nodo p1 al p2 con peso w, o cambiar
      // el peso de la arista a w, calcular la
      // posición resultante de la flecha
            int dx, dy;
            float l;
            peso[p1][p2]=w;
            // linea de direccion entre p1 y p2
            dx = nodo[p2].x-nodo[p1].x;
            dy = nodo[p2].y-nodo[p1].y;
            // distancia entre p1 y p2
            l = (float)( Math.sqrt((float)(dx*dx + dy*dy)));
            dir_x[p1][p2]=dx/l;
            dir_y[p1][p2]=dy/l;
            // calcular el start y endpoints de la arista,
            // ajustar startpoints si tambien hay una arista de p2 a p1
            if (peso[p2][p1]>0) {
              startp[p1][p2] = new Point((int)(nodo[p1].x-5*dir_y[p1][p2]),
                                         (int)(nodo[p1].y+5*dir_x[p1][p2]));
              endp[p1][p2] = new Point((int)(nodo[p2].x-5*dir_y[p1][p2]),
                                       (int)(nodo[p2].y+5*dir_x[p1][p2]));
            }
            else {
              startp[p1][p2] = new Point(nodo[p1].x, nodo[p1].y);
              endp[p1][p2] = new Point(nodo[p2].x, nodo[p2].y);
```

```
        }
        // la distancia de la flecha no es todo el camino a los puntos de inicio/final
        int diff_x = (int)(Math.abs(20*dir_x[p1][p2]));
        int diff_y = (int)(Math.abs(20*dir_y[p1][p2]));
        // calcular nueva posicion en x de la flecha
        if (startp[p1][p2].x>endp[p1][p2].x) {
          arista[p1][p2] = new Point(endp[p1][p2].x + diff_x +
                (Math.abs(endp[p1][p2].x-startp[p1][p2].x) - 2*diff_x )*
                        (100-w)/100 , 0);
        }
        else {
          arista[p1][p2] = new Point(startp[p1][p2].x + diff_x +
                (Math.abs(endp[p1][p2].x-startp[p1][p2].x) - 2*diff_x )*
                        w/100, 0);
        }
        // calcular nueva posicion en y de la flecha
        if (startp[p1][p2].y>endp[p1][p2].y) {
          arista[p1][p2].y=endp[p1][p2].y + diff_y +
                (Math.abs(endp[p1][p2].y-startp[p1][p2].y) - 2*diff_y )*
                        (100-w)/100;
        }
        else {
          arista[p1][p2].y=startp[p1][p2].y + diff_y +
                (Math.abs(endp[p1][p2].y-startp[p1][p2].y) - 2*diff_y )*
                        w/100;
        }
}

public String intToString(int i) {
        char c=(char)((int)'a'+i);
        return ""+c;
}
public final synchronized void update(Graphics g) {
// preparar nueva imagen fuera de la pantalla
        Dimension d=size();
        if ((offScreenImage == null) || (d.width != offScreenSize.width) ||
                        (d.height != offScreenSize.height)) {
           offScreenImage = createImage(d.width, d.height);
           offScreenSize = d;
           offScreenGraphics = offScreenImage.getGraphics();
        }
        offScreenGraphics.setColor(Color.white);
        offScreenGraphics.fillRect(0, 0, d.width, d.height);
        paint(offScreenGraphics);
        g.drawImage(offScreenImage, 0, 0, null);
```

```
 }

public void drawarista(Graphics g, int i, int j) {
  // dibuja arista entre nodo i y nodo j
          int x1, x2, x3, y1, y2, y3;
          // calcular flecha
          x1= (int)(arista[i][j].x - 3*dir_x[i][j] + 6*dir_y[i][j]);
          x2= (int)(arista[i][j].x - 3*dir_x[i][j] - 6*dir_y[i][j]);
          x3= (int)(arista[i][j].x + 6*dir_x[i][j]);
          y1= (int)(arista[i][j].y - 3*dir_y[i][j] - 6*dir_x[i][j]);
          y2= (int)(arista[i][j].y - 3*dir_y[i][j] + 6*dir_x[i][j]);
          y3= (int)(arista[i][j].y + 6*dir_y[i][j]);
          int flecha_x[] = { x1, x2, x3, x1 };
          int flecha_y[] = { y1, y2, y3, y1 };
          // si la arista ya se escogio por el algoritmo cambiar color
          if (algedge[i][j]) g.setColor(Color.orange);
          // dibuja arista
          g.drawLine(startp[i][j].x, startp[i][j].y, endp[i][j].x, endp[i][j].y);
          g.fillPolygon(flecha_x, flecha_y, 4);
          // escribe el peso de la arista en una posicion apropiada
          int dx = (int)(Math.abs(7*dir_y[i][j]));
          int dy = (int)(Math.abs(7*dir_x[i][j]));
          String str = new String("" + peso[i][j]);
          g.setColor(Color.black);
          if ((startp[i][j].x>endp[i][j].x) && (startp[i][j].y>=endp[i][j].y))
            g.drawString( str, arista[i][j].x + dx, arista[i][j].y - dy);
          if ((startp[i][j].x>=endp[i][j].x) && (startp[i][j].y<endp[i][j].y))
            g.drawString( str, arista[i][j].x - fmetrics.stringWidth(str) - dx ,
                                  arista[i][j].y - dy);
          if ((startp[i][j].x<endp[i][j].x) && (startp[i][j].y<=endp[i][j].y))
            g.drawString( str, arista[i][j].x - fmetrics.stringWidth(str) ,
                                  arista[i][j].y + fmetrics.getHeight());
          if ((startp[i][j].x<=endp[i][j].x) && (startp[i][j].y>endp[i][j].y))
             g.drawString( str, arista[i][j].x + dx,
                                   arista[i][j].y + fmetrics.getHeight() );
 }

 public void detailsDijkstra(Graphics g, int i, int j) {
 // checar que arista entre nodo i y nodo j esta cerca de la arista s para
 //escoger durante este paso del algoritmo
 // checar si el nodo j tiene la siguiente minima distancia al nodo_inicial
          if ( (finaldist[i]!=-1) && (finaldist[j]==-1) ) {
            g.setColor(Color.red);
            if ( (dist[j]==-1) || (dist[j]>=(dist[i]+peso[i][j])) ) {
        if ( (dist[i]+peso[i][j])<dist[j] ) {
```

```
            changed[j]=true;
            numchanged++;
          }
            dist[j] = dist[i]+peso[i][j];
            colornodo[j]=Color.red;
            if ( (mindist==0) || (dist[j]<mindist) ) {
              mindist=dist[j];
              minstart=i;
              minend=j;
            }
          }
        }
        else g.setColor(Color.gray);
    }

public void endstepDijkstra(Graphics g) {
    // despliega distancias parcial y total de los nodos, ajusta la distancia final
    // para el nodo que tuvo la mínima distancia en este paso
    // explica el algoritmo en el panel de documentacion
      for (int i=0; i<numnodos; i++)
          if ( (nodo[i].x>0) && (dist[i]!=-1) ) {
            String str = new String(""+dist[i]);
            g.drawString(str, nodo[i].x - (int)fmetrics.stringWidth(str)/2 -1,
                                  nodo[i].y + h);
            if (finaldist[i]==-1) {
              neighbours++;
              if (neighbours!=1)
                showstring = showstring + ", ";
              showstring = showstring + intToString(i) +"=" + dist[i];
            }
          }
      showstring = showstring + ". ";

      if ( (paso>1) && (numchanged>0) ) {
        if (numchanged>1)
          showstring = showstring + "Note que las distancias a ";
        else showstring = showstring + "Note que la distancia a ";
        for (int i=0; i<numnodos; i++)
          if ( changed[i] )
            showstring = showstring + intToString(i) +", ";
        if (numchanged>1)
          showstring = showstring + "han cambiado!\n";
        else showstring = showstring + "ha cambiado!\n";
      }
      else showstring = showstring + " ";
```

```
            if (neighbours>1) {
            // si hay otros candidatos explicar porque se tomo este
              showstring = showstring + "El nodo " + intToString(minend) +
                                                  " tiene la distancia minima.\n";
              //checar sy hay otros caminos a minend.
              int newcaminos=0;
              for (int i=0; i<numnodos; i++)
                if ( (nodo[i].x>0) && (peso[i][minend]>0) && ( finaldist[i] == -1 ) )
                  newcaminos++;
              if (newcaminos>0)
                showstring = showstring + "Cualquier otro camino a " +
intToString(minend) +
                 " visita otro nodo de la red, y sera mas largo que " +  mindist + ".\n";
          else showstring = showstring +
                                      "No hay otras aristas entrando a "+
                                      intToString(minend) + ".\n";
            }
            else {
          boolean morenodos=false;
          for (int i=0; i<numnodos; i++)
                if ( ( nodo[i].x>0 ) && ( finaldist[i] == -1 ) && ( peso[i][minend]>0 ) )
                  morenodos=true;
          boolean bridge=false;
          for (int i=0; i<numnodos; i++)
                if ( ( nodo[i].x>0 ) && ( finaldist[i] == -1 ) && ( peso[minend][i]>0 ) )
                  bridge=true;
          if ( morenodos && bridge )
            showstring = showstring + "Dado que este nodo forma un 'puente' a "+
             "los nodos restantes,\ncualquier otro camino a este nodo sera mas largo.\n";
          else if ( morenodos && (!bridge) )
            showstring = showstring + "Los nodos grises restantes no son
alcanzables.\n";
          else showstring = showstring + "No hay otras aristas entrando a "+
                  intToString(minend) + ".\n";
       }
              showstring = showstring + "Node " + intToString(minend) +
                " sera coloreado naranja para indicar que " + mindist +
                " es la longitud del camino mas corto a " + intToString(minend) +".";
              parent.documentacion.doctext.showline(showstring);
    }

public void detailsalg(Graphics g, int i, int j) {
    // mas algoritmos pueden ser aniadidos
            if (algoritmo==DIJKSTRA)
```

```
            detailsDijkstra(g, i, j);
   }

public void endstepalg(Graphics g) {
   // mas algoritmos pueden ser aniadidos
           if (algoritmo==DIJKSTRA)
              endstepDijkstra(g);
           if ( ( performalg ) && (mindist==0) ) {
             if (algrthm != null) algrthm.stop();
             int nalcanzable = 0;
             for (int i=0; i<numnodos; i++)
               if (finaldist[i] > 0)
                       nalcanzable++;
             if (nalcanzable == 0)
               parent.documentacion.doctext.showline("ninguno");
       else if (nalcanzable< (numnodos-emptyspots-1))
               parent.documentacion.doctext.showline("alguno");
             else
               parent.documentacion.doctext.showline("hecho");
           }
   }

public void paint(Graphics g) {
      mindist=0;
           minnodo=MAXNODOS;
           minstart=MAXNODOS;
           minend=MAXNODOS;
      for(int i=0; i<MAXNODOS; i++)
        changed[i]=false;
      numchanged=0;
      neighbours=0;
           g.setFont(roman);
           g.setColor(Color.black);
      if (paso==1)
        showstring="Algoritmo ejecutando: las aristas rojas apuntan a nodos
alcanzables desde " +
              " el nodo_inicial.\nLa distancia a: ";
      else
       showstring="Paso " + paso + ": Las aristas rojsa apuntan a nodos alcanzables
desde " +
              "nodos que ya tienen una distancia final ." +
              "\nLa distancia a: ";
           // dibuja una nueva arista en la posicion del mouse
           if (newarista)
            g.drawLine(nodo[nodo1].x, nodo[nodo1].y, thispoint.x, thispoint.y);
```

```
            // dibuja todas las aristas
            for (int i=0; i<numnodos; i++)
              for (int j=0; j<numnodos; j++)
                if (peso [i][j]>0) {
 // si el algoritmo se esta ejecutando entonces hacer el siguiente paso para esta
arista
                  if (performalg)
                    detailsalg(g, i, j);
                  drawarista(g, i, j);
                }
            // si la flecha ha sido arrastrada a  0, dibujala, para que el usuario
            //tenga la opcion de hacerla positiva de nuevo
            if (movearista && peso[nodo1][nodo2]==0) {
              drawarista(g, nodo1, nodo2);
              g.drawLine(startp[nodo1][nodo2].x, startp[nodo1][nodo2].y,
                         endp[nodo1][nodo2].x, endp[nodo1][nodo2].y);
            }
            // dibuja los nodos
            for (int i=0; i<numnodos; i++)
              if (nodo[i].x>0) {
                g.setColor(colornodo[i]);
                g.fillOval(nodo[i].x-NODORADIX, nodo[i].y-NODORADIX,
                           NODOSIZE, NODOSIZE);
              }
            // refleja el nodo_inicial que se mueve
            g.setColor(Color.blue);
            if (movestart)
              g.fillOval(thispoint.x-NODORADIX, thispoint.y-NODORADIX,
                           NODOSIZE, NODOSIZE);
            g.setColor(Color.black);
            // termina este paso del algoritmo
            if (performalg) endstepalg(g);
            // dibuja circulos negros alrededor de los nodos, escribe sus nombres a la
pantalla
            g.setFont(helvetica);
            for (int i=0; i<numnodos; i++)
              if (nodo[i].x>0) {
                g.setColor(Color.black);
                g.drawOval(nodo[i].x-NODORADIX, nodo[i].y-NODORADIX,
                           NODOSIZE, NODOSIZE);
                g.setColor(Color.blue);
                g.drawString(intToString(i), nodo[i].x-14, nodo[i].y-14);
              }
    }
}
```

CAPÍTULOS 5 Y 6

PILAS, COLAS Y LISTAS

PROPÓSITO DEL CAPÍTULO

Este capítulo deberá proporcionar una base importante en la teoría y aplicación de estructuras de datos, familiarizando al estudiante con las estructuras de datos fundamentales que todo buen programador debe conocer.

Enseñará un estilo moderno de implementación de estructuras de datos conocido como abstracción de datos, incorporando además a éste los mecanismos de programación más recientes propios del estilo orientado a objetos.

OBJETIVOS DE APRENDIZAJE

Al finalizar el estudio de este tema, el estudiante deberá estar en capacidad de:

- Explicar los mecanismos de abstracción y su importancia para la resolución de problemas.
- Explicar conceptos de programación modular y de reutilización de los componentes de software.
- Desarrollar programas basándose en tipos abstractos de datos (TAD).

GUÍA DE LECTURAS

Para lograr los objetivos anteriores, se le sugiere seguir la siguiente guía:

Subtema	Capítulo	Páginas
Implementación dinámica de vectores	15	395 - 403
Implementación con listas enlazadas	15	404 - 408
Listas enlazadas	16	415 - 434

COMENTARIOS GENERALES

Una vez que usted ha realizado las lecturas anteriores, analice cuidadosamente los comentarios siguientes, ya que con ellos se pretende enfatizar o ampliar algunos contenidos importantes del capítulo. Existen varios aspectos importantes que debemos señalar, antes de entrar al desarrollo de los subtemas.

Una **lista** (TDA) se define como una serie de *N* elementos E_1, E_2, ..., E_N, ordenados de manera consecutiva; es decir, el elemento E_k (que se denomina *elemento k-ésimo*) es previo al elemento E_{k+1}. Si la lista contiene 0 elementos, se denomina como **lista vacía**.

Las operaciones que se pueden realizar en la lista son: insertar un elemento en la posición *k*, borrar el k-ésimo elemento, buscar un elemento dentro de la lista y preguntar si la lista esta vacía.

Una manera simple de implementar una lista es utilizando un arreglo. Sin embargo, las operaciones de inserción y borrado de elementos en arreglos son ineficientes, puesto que, para insertar un elemento en la parte media del arreglo, es necesario mover todos los elementos que se encuentren delante de él, para hacer espacio y, al borrar un elemento, es necesario mover todos los elementos para ocupar el espacio desocupado. Una implementación más eficiente del TDA se logra utilizando listas enlazadas.

A continuación, se presenta una implementación en Java del TDA utilizando listas enlazadas y sus operaciones asociadas:

- **estaVacia()**: Devuelve *verdadero* si la lista esta vacía, falso en caso contrario.
- **insertar(x, k)**: Inserta el elemento *x* en la k-ésima) posición de la lista.
- **buscar(x)**: Devuelve la posición en la lista del elemento *x*.
- **buscarK(k)**: Devuelve el k-ésimo) elemento de la lista.
- **eliminar(x)**: Elimina de la lista el elemento *x*.

En la implementación con listas enlazadas es necesario tener en cuenta algunos detalles importantes. Si solamente se dispone de la referencia al primer elemento, el añadir o remover en la primera posición es un caso especial, puesto que la referencia a la lista enlazada debe modificarse según la operación realizada. Además, para eliminar un elemento en particular, es necesario conocer el elemento que lo antecede. En este caso, ¿qué pasa con el primer elemento, que no tiene un predecesor?

Para solucionar estos inconvenientes, se utiliza la implementación de lista enlazada con nodo cabecera. Con esto, todos los elementos de la lista tendrán un elemento

previo, puesto que el previo del primer elemento es la cabecera. Una lista vacía corresponde, en este caso, a una cabecera cuya referencia siguiente es *null*.

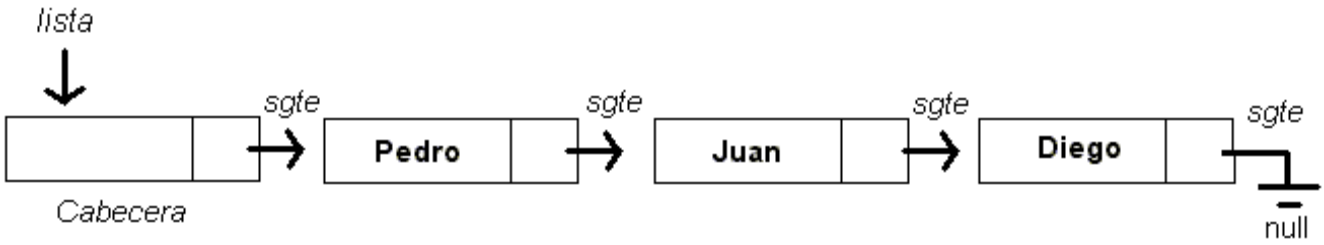


Figura No. 6

Los archivos **NodoLista.java, IteradorLista.java y Lista.java** contienen una implementación completa del TDA lista. La clase *NodoLista* implementa los nodos de la lista enlazada; la clase *Lista* implementa las operaciones de la lista propiamente tal, y la clase *IteradorLista* implementa objetos que permiten recorrer la lista y posee la siguiente interfaz:

- **avanzar()**: Avanza el iterador al siguiente nodo de la lista.
- **obtener()**: Retorna el elemento del nodo en donde se encuentra el iterador.

Costo de las operaciones en tiempo:

- Insertar/eliminar elemento en k-ésima posición: *O(k)* (¿Se puede hacer en *O(1)*?)

- Buscar elemento *x*: *O(N)* (promedio)

Una **pila** (*stack* o *pushdown* en inglés) es una lista de elementos de la que sólo se puede extraer el último elemento insertado. La posición en la que se encuentra este elemento se denomina **tope** de la pila. También se conoce a las pilas como **listas LIFO** (*LAST IN - FIRST OUT*: el último que entra es el primero que sale).

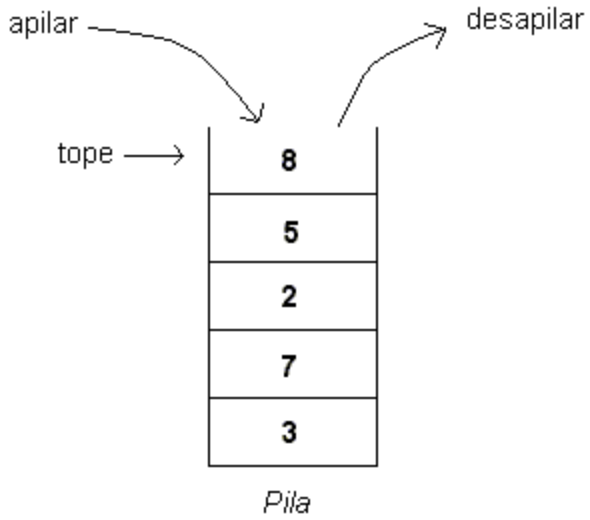


Figura No. 7

La interfaz de este TDA provee las siguientes operaciones:

- **apilar(x)**: Inserta el elemento *x* en el tope de la pila (*push* en inglés).
- **desapilar()**: Retorna el elemento que se encuentre en el tope de la pila, y lo elimina de ésta (*pop* en inglés).
- **tope()**: Retorna el elemento que se encuentre en el tope de la pila, pero sin eliminarlo de esta (*top* en inglés).
- **estaVacia()**: Retorna *verdadero* si la pila no contiene elementos, *falso* en caso contrario (*isEmpty* en inglés).

Nota: Algunos autores definen **desapilar** como sacar el elemento del tope de la pila sin retornarlo.

Implementación del TDA pila

A continuación, se muestran dos maneras de implementar una pila: utilizando un arreglo y utilizando una lista enlazada. En ambos casos, el costo de las operaciones es de *O(1)*.

Implementación utilizando arreglos

Para implementar una pila utilizando un arreglo, basta con definir el arreglo del tipo de dato que se almacenará en la pila. Una variable de instancia indicará la posición del tope de la pila, lo cual permitirá realizar las operaciones de inserción y borrado, y

también permitirá saber si la pila está vacía, definiendo que dicha variable vale **-1** cuando no hay elementos en el arreglo.

```
class PilaArreglo
{
  private Object[] arreglo;
  private int tope;
  private int MAX_ELEM=100; // máximo número de elementos en la pila

  public PilaArreglo()
  {
    arreglo=new Object[MAX_ELEM];
    tope=-1; // inicialmente la pila está vacía
  }

  public void apilar(Object x)
  {
    if (tope+1<MAX_ELEM) // si está llena se produce OVERFLOW
    {
      tope++;
      arreglo[tope]=x;
    }
  }

  public Object desapilar()
  {
    if (!estaVacia()) // si está vacía se produce UNDERFLOW
    {
      Object x=arreglo[tope];
      tope--;
      return x;
    }
  }

  public Object tope()
  {
    if (!estaVacia()) // si está vacía es un error
    {
      Object x=arreglo[tope];
      return x;
    }
  }

  public boolean estaVacia()
  {
```

```
   if (tope==-1)
   {
     return true;
   }
   else
   {
     return false;
   }
  }
}
```

El inconveniente de esta implementación es que es necesario fijar de antemano el número máximo de elementos que puede contener la pila, *MAX_ELEM*. Por lo tanto, al apilar un elemento es necesario controlar que no se inserte un elemento si la pila está llena. Sin embargo, en Java es posible solucionar este problema creando un nuevo arreglo más grande que el anterior (el doble, por ejemplo, y copiando los elementos de un arreglo a otro:

```
public void apilar(Object x)
{
  if (tope+1<MAX_ELEM) // si está llena se produce OVERFLOW
  {
    tope++;
    arreglo[tope]=x;
  }
  else
  {
    MAX_ELEM=MAX_ELEM*2;
    Object[] nuevo_arreglo=new Object[MAX_ELEM];
    for (int i=0; i<arreglo.length; i++)
    {
      nuevo_arreglo[i]=arreglo[i];
    }
    tope++;
    nuevo_arreglo[tope]=x;
    arreglo=nuevo_arreglo;
  }
}
```

Implementación utilizando listas enlazadas

En este caso no existe el problema de tener que fijar el tamaño máximo de la pila (aunque siempre se está acotado por la cantidad de memoria disponible). La implementación es bastante simple: los elementos siempre se insertan al principio de la lista (apilar), y siempre se extrae el primer elemento de la lista (desapilar y

tope), por lo que basta con tener una referencia al principio de la lista enlazada. Si esta referencia es *null*, entonces la pila está vacía.

```
class PilaLista
{
  private NodoLista lista;

  public PilaLista()
  {
    lista=null;
  }
public void apilar(Object x)
  {
    lista=new NodoLista(x, lista);
  }

public Object desapilar() // si está vacía se produce UNDERFLOW
  {
    if (!estaVacia())
    {
      Object x=lista.elemento;
      lista=lista.siguiente;
      return x;
    }
  }

  public Object tope()
  {
    if (!estaVacia()) // si está vacía es un error
    {
      Object x=lista.elemento;
      return x;
    }
  }

  public boolean estaVacia()
  {
    return lista==null;
  }
}
```

Dependiendo de la aplicación que se le dé a la pila, es necesario definir qué acción realizar en caso de que ocurra *overflow* (rebalse de la pila) o *underflow* (intentar desapilar cuando la pila está vacía). Java posee un mecanismo denominado

excepciones, que permite realizar acciones cuando se producen ciertos eventos específicos (por ejemplo: *overflow* o *underflow* en una pila).

En ambas implementaciones el costo de las operaciones que provee el TDA es costo *O(1)*.

Ejemplo de uso: Eliminación de recursividad

Suponga que una función *F* realiza un llamado recursivo dentro de su código, lo que se ilustra en la siguiente figura:

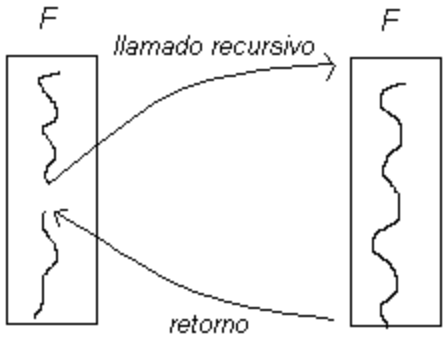


Figura No. 8

Si la llamada recursiva es lo último que hace la función *F*, entonces dicha llamada se puede substituir por un ciclo *while*. Este caso es conocido como *tail recursion*, y en lo posible hay que evitarlo en la programación, ya que cada llamada recursiva ocupa espacio en la memoria del computador. En el caso del *tail recursion,* es muy simple eliminarlo. Por ejemplo:

```
void imprimir(int[] a, int j) // versión recursiva
{
  if (j<a.length)
  {
    System.out.println(a[j]);
    imprimir(a, j+1); // tail recursión
  }
}

void imprimir(int[] a, int j) // versión iterativa
{
  while (j<a.length)
  {
    System.out.println(a[j]);
    j=j+1;
```

```
  }
}
```

En el caso general, cuando el llamado recursivo se realiza en medio de la función *F*, la recursión se puede eliminar utilizando una pila.

Por ejemplo: recorrido en preorden de un árbol binario.

```
// "raíz" es la referencia a la raíz del árbol
// llamado inicial: preorden(raiz)
// versión recursiva

void preorden(Nodo nodo)
{
  if (nodo!=null)
  {
    System.out.print(nodo.elemento);
    preorden(nodo.izq);
    preorden(nodo.der);
  }
}

// primera versión iterativa
void preorden(Nodo nodo)
{
  Nodo aux;
  Pila pila=new Pila(); // pila de nodos
  pila.apilar(nodo);
  while(!pila.estaVacia()) // mientras la pila no esté vacía
  {
    aux=pila.desapilar();
    if (aux!=null)
    {
      System.out.print(aux.elemento);
      // primero se apila el nodo derecho y luego el izquierdo
      // para mantener el orden correcto del recorrido
      // al desapilar los nodos
      pila.apilar(aux.der);
      pila.apilar(aux.izq);
    }
  }
}

// segunda versión iterativa
// dado que siempre el último nodo apilado dentro del bloque if es
```

```
// aux.izq podemos asignarlo directamente a aux hasta que éste sea
// null, es decir, el bloque if se convierte en un bloque while
// y se cambia el segundo apilar por una asignación de la referencia

void preorden(Nodo nodo)
{
  Nodo aux;
  Pila pila=new Pila(); // pila de nodos
  pila.apilar(nodo);
  while(!pila.estaVacia()) // mientras la pila no este vacía
  {
    aux=pila.desapilar();
    while (aux!=null)
    {
      System.out.print(aux.elemento);
      pila.apilar(aux.der);
      aux=aux.izq;
    }
  }
}
```

Si bien los programas no recursivos son más eficientes que los recursivos, la eliminación de recursividad (excepto en el caso de *tail recursion*) le quita claridad al código del programa. Por lo tanto:

- A menudo es conveniente eliminar el *tail recursion*.
- Un método recursivo es menos eficiente que uno no recursivo, pero sólo en pocas oportunidades vale la pena eliminar la recursión.

Una **cola** (*queue* en inglés) es una lista de elementos en donde siempre se insertan nuevos elementos al final de la lista y se extraen elementos desde el inicio de la lista. También se conoce a las colas como **listas FIFO** (*FIRST IN - FIRST OUT*: el primero que entra es el primero que sale).

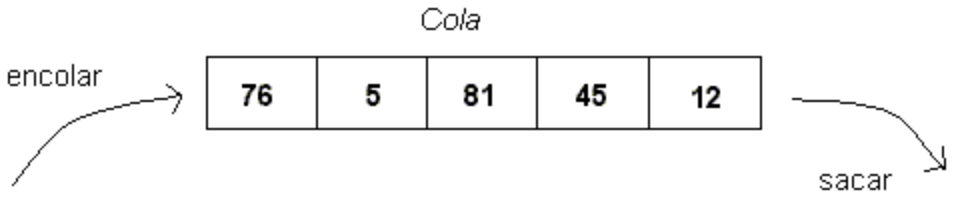


Figura No. 9

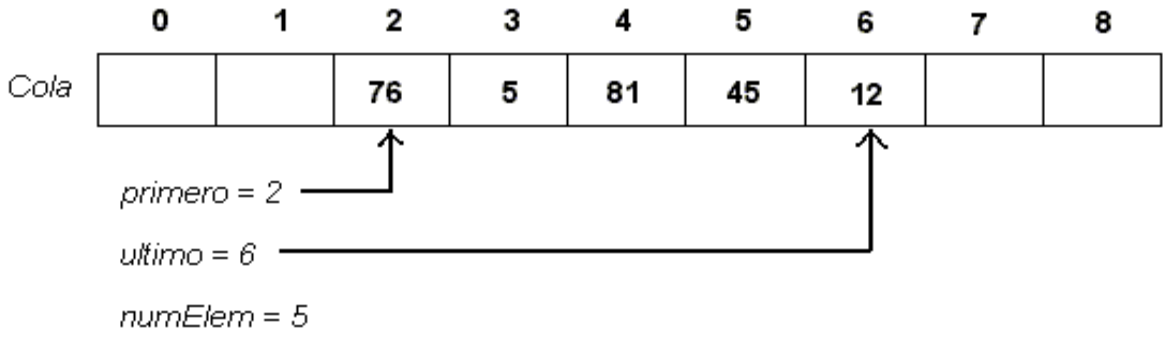
Las operaciones básicas en una cola son:

- **encolar(x)**: Inserta el elemento *x* al final de la cola (*enqueue* en inglés).
- **sacar()**: Retorna el elemento que se ubica al inicio de la cola (*dequeue* en inglés).
- **estaVacia()**: Retorna *verdadero* si la cola esta vacía, *falso* en caso contrario.

Al igual que con el TDA pila, una cola puede implementarse tanto con arreglos como con listas enlazadas. A continuación, se verá la implementación utilizando un arreglo.

Las variables de instancia necesarias en la implementación son:

- ***primero***: Indica el índice de la posición del primer elemento de la cola, es decir, la posición del elemento por retornar cuando se invoque **sacar**.
- ***ultimo***: Indica el índice de la posición de último elemento de la cola. Si se invoca **encolar**, el elemento debe ser insertado en el casillero siguiente al que indica la variable.
- ***numElem***: Indica cuántos elementos posee la cola. Definiendo *MAX_ELEM* como el tamaño máximo del arreglo y, por lo tanto, de la cola, entonces la cola está vacía si *numElem==0* y está llena si *numElem==MAX_ELEM*.



Un detalle faltante es el siguiente: ¿Qué pasa si la variable *ultimo* sobrepasa el rango de índices del arreglo? Esto se soluciona definiendo que, si después de insertar un elemento, el índice *ultimo* == *MAX_ELEM*, entonces se asigna *ultimo* = *0*, y los siguientes elementos serán insertados al comienzo del arreglo. Esto no produce ningún efecto en la lógica de las operaciones del TDA, pues siempre se saca el elemento referenciado por el índice *primero*, aunque en valor absoluto *primero* > *ultimo*. Este enfoque es conocido como **implementación con arreglo**

circular, y la forma más fácil de implementarlo es haciendo la aritmética de subíndices módulo *MAX_ELEM*.

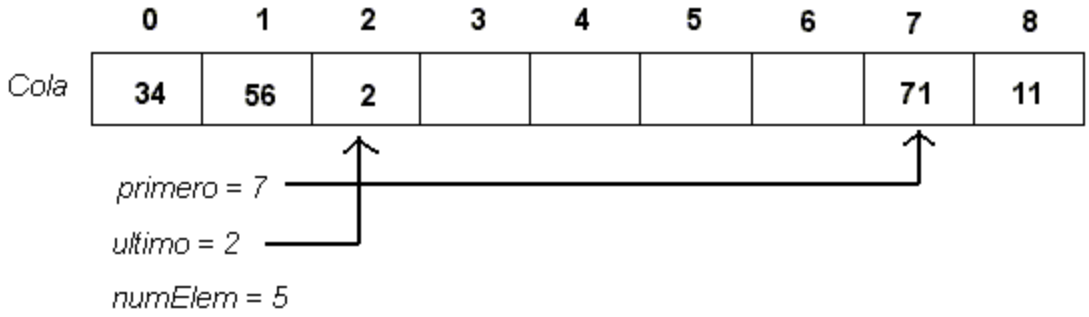


Figura No. 10

```
class ColaArreglo
{
  private Object[] arreglo;
  private int primero, ultimo, numElem;
  private int MAX_ELEM=100; // máximo número de elementos en la cola

  public ColaArreglo()
  {
    arreglo=new Object[MAX_ELEM];
    primero=0;
    ultimo=MAX_ELEM-1;
    numElem=0;
  }

  public void encolar(Object x)
  {
    if (numElem<MAX_ELEM) // si esta llena se produce OVERFLOW
    {
      ultimo=(ultimo+1)%MAX_ELEM;
      arreglo[ultimo]=x;
      numElem++;
    }
  }

  public Object sacar()
  {
    if (!estaVacia()) // si esta vacía se produce UNDERFLOW
    {
      Object x=arreglo[primero];
```

```
    primero=(primero+1)%MAX_ELEM;
    numElem--;
    return x;
   }
  }

  public boolean estaVacia()
  {
   return numElem==0;
  }
}
```

Nuevamente en este caso, dependiendo de la aplicación, se debe definir qué hacer en caso de producirse *OVERFLOW* o *UNDERFLOW*.

Con esta implementación, todas las operaciones del TDA cola tienen costo *O(1)*.

Una **cola de prioridad** es un tipo de datos abstracto que almacena un conjunto de datos que poseen una llave perteneciente a algún conjunto ordenado, y permite *insertar* nuevos elementos y *extraer el máximo* (o el mínimo, en caso de que la estructura se organice con un criterio de orden inverso).

Es frecuente interpretar los valores de las llaves como prioridades, con lo cual la estructura permite insertar elementos de prioridad cualquiera, y extraer el de mejor prioridad.

Dos formas simples de implementar colas de prioridad son:

- Una lista ordenada:
  - Inserción: O(n)
  - Extracción de máximo: O(1)
- Una lista desordenada:
  - Inserción: O(1)
  - Extracción de máximo: O(n)

Heaps

Un ***heap*** es un árbol binario de una forma especial, que permite su almacenamiento en un arreglo sin usar punteros.

Un *heap* tiene todos sus niveles llenos, excepto posiblemente el de más abajo, y en este último los nodos están lo más a la izquierda posible.

Ejemplo:

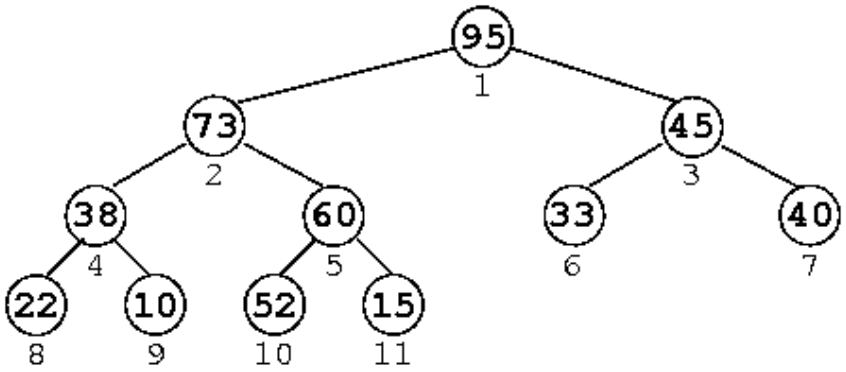


Figura No. 11

La numeración por niveles (indicada bajo cada nodo) son los subíndices en donde cada elemento sería almacenado en el arreglo. En el caso del ejemplo, el arreglo sería:

95	73	45	38	60	33	40	22	10	52	15
1	2	3	4	5	6	7	8	9	10	11

Figura No. 12

La característica que permite que un *heap* se pueda almacenar sin punteros es aquella en la que, si se utiliza la numeración por niveles indicada, entonces la relación entre padres e hijos es:

Hijos del nodo j = {2*j, 2*j+1}
Padre del nodo k = floor(k/2)

Un *heap* puede utilizarse para implementar una cola de prioridad almacenando los datos de modo que las llaves estén siempre ordenadas de arriba a abajo (a diferencia de un árbol de búsqueda binaria, que ordena sus llaves de izquierda a derecha). En otras palabras, el padre debe tener siempre mayor prioridad que sus hijos (ver ejemplo).

Implementación de las operaciones básicas

Inserción:

La **inserción** se realiza agregando el nuevo elemento en la primera posición libre del *heap*, esto es, el próximo nodo que debería aparecer en el recorrido por niveles o, equivalentemente, un casillero que se agrega al final del arreglo.

Después de agregar este elemento, la *forma* del *heap* se preserva, pero la restricción de orden no tiene por qué cumplirse. Para resolver este problema, si el nuevo elemento es mayor que su padre, se intercambia con él, y ese proceso se repite mientras sea necesario. Una forma de describir esto es diciendo que el nuevo elemento **trepa** en el árbol hasta alcanzar el nivel correcto, según su prioridad.

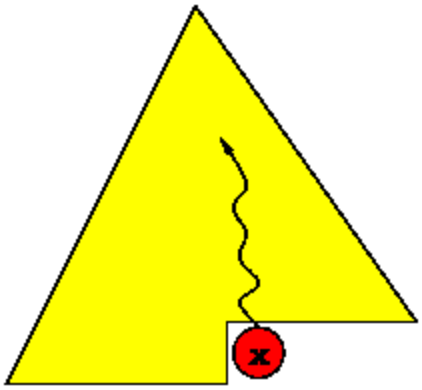


Figura No. 13

El siguiente trozo de programa muestra el proceso de inserción de un nuevo elemento x:

```
a[++n]=x;
for(j=n; j>1 && a[j]>a[j/2]; j/=2)
 { # intercambiamos con el padre
   t=a[j];
   a[j]=a[j/2];
   a[j/2]=t;
 }
```

El proceso de inserción, en el peor caso, toma un tiempo proporcional a la altura del árbol, esto es, O(log n).

Extracción del máximo

El máximo evidentemente está en la raíz del árbol (casillero 1 del arreglo). Al sacarlo de ahí, podemos imaginar que ese lugar queda vacante. Para llenarlo, tomamos al *último* elemento del *heap* y lo trasladamos al lugar vacante. En caso de que no esté bien ahí de acuerdo con su prioridad (¡que es lo más probable!), lo hacemos descender intercambiándolo siempre con el mayor de sus hijos. Decimos que este elemento "se hunde" hasta su nivel de prioridad.

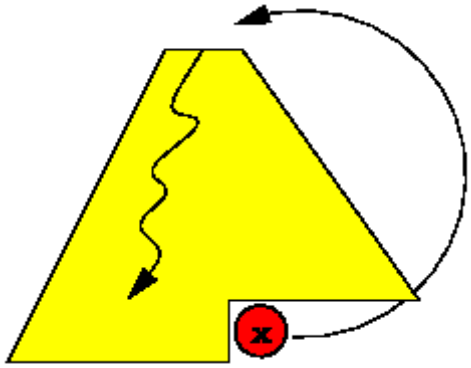


Figura No. 14

El siguiente trozo de programa implementa este algoritmo:

```
m=a[1];  # La variable m lleva el máximo
a[1]=a[n--];  # Movemos el último a la raíz y achicamos el heap
j=1;
while(2*j<n) # mientras tenga algún hijo
 {
   k=2*j; # el hijo izquierdo
   if(k+1<=n && a[k+1]>a[k])
      k=k+1;  # el hijo derecho es el mayor
   if(a[j]>a[k])
      break;  # es mayor que ambos hijos
   t=a[j];
   a[j]=a[k];
   a[k]=t;
   j=k;   # lo intercambiamos con el mayor hijo
 }
```

Este algoritmo también demora un tiempo proporcional a la altura del árbol en el peor caso, esto es, O(log n).

GLOSARIO

En esta área usted encontrará un glosario donde se incluyen términos y la definición de cada uno para que le sea más fácil comprender el tema de estudio.

Clase iteradota: Clase que mantiene la posición actual y contiene todas las rutinas que dependen del conocimiento de la posición de la lista en que nos encontramos.

Lista: Una serie de *N* elementos E_1, E_2, ..., E_N, ordenados de manera consecutiva.

Lista doblemente enlazada: Permite recorridos en ambos sentidos, mediante el almacenamiento de dos referencias por nodo.

Lista enlazada ordenada: Lista enlazada en la que los elementos están ordenados.

EJERCICIO SUGERIDO

Para este tema, se sugiere que usted realice el siguiente ejercicio de su libro de texto:

16.14 Implemente un clase de listas doblemente enlazadas. Este ejemplo consta de 3 partes:

Primera parte: La implementación de la clase subnodo
Segunda parte: La implementación de la clase ListaDoble
Tercera parte: Un pequeño programita que hace algunas operaciones para testearla.

RESOLUCIÓN DE EJERCICIO SUGERIDO

16.14 Implemente un clase de listas doblemete enlazadas. Este ejemplo consta de 3 partes:

Primera parte: *La implementacion de la clase Subnodo*
Segunda parte: *La implementación de la clase ListaDoble*
Tercera parte: *Un pequeño programita que hace algunas operaciones para testearla.*

Primera parte:

```
PUBLIC CLASS SUBNODO {
   PRIVATE INT INFO;
   PRIVATE SUBNODO PROX;
   PRIVATE SUBNODO ANT;

   SUBNODO()
   {INFO=0;
    PROX=NULL;
    ANT=NULL;
```

```
   }

   PUBLIC VOID SETINFO(INT X)
   {INFO=X;
   }
   PUBLIC VOID SETPROX(SUBNODO P)
   {PROX=P;
   }
   PUBLIC VOID SETANT(SUBNODO A)
   {ANT=A;
   }

   PUBLIC INT GETINFO()
   {RETURN INFO;
   }
   PUBLIC SUBNODO GETPROX()
   { RETURN PROX;

   }
   PUBLIC SUBNODO GETANT()
   { RETURN ANT;

   }

   PUBLIC STRING  TOSTRING()
   { RETURN ("\T"+INFO);
   }

}
```

Segunda Parte: La de la clase generica TDA ListaDoble

```
PUBLIC CLASS LISTADOBLE {
   SUBNODO FRENTE;

  PUBLIC LISTADOBLE()
  {FRENTE=NULL;
  }

PUBLIC VOID INSERTARDOBLE(INT  X )
{ SUBNODO  P;
  P = NEW SUBNODO();
  IF(P!=NULL)
```

```
  { P.SETINFO(X);

    P.SETPROX(FRENTE);

    IF( FRENTE != NULL)  FRENTE.SETANT(P);
     FRENTE = P;
  }
 ELSE
   SYSTEM.OUT.PRINTLN("\N ERROR FALTA MEMORIA  ");

}
PUBLIC VOID IMPRIMIRLISTA()
 {
  SUBNODO P;
  P=FRENTE;
   SYSTEM.OUT.PRINTLN(" IMPRESION DE LA LISTA DOBLE DE ENTEROS ");
  WHILE (P!=NULL)
   { SYSTEM.OUT.PRINT(" "+P.TOSTRING());
     P=P.GETPROX();
   }
 SYSTEM.OUT.PRINTLN( );

 }

 PUBLIC STRING TOSTRING()
 {STRING AUX=" ";
  SUBNODO  P;
   P=FRENTE;
   WHILE (P!=NULL)
    { AUX=AUX+P.TOSTRING();
      P=P.GETPROX();
    }
  RETURN AUX;

 }

PUBLIC VOID BUSCAR(INT X)
{SUBNODO  P;
 P=FRENTE;
 WHILE(P!=NULL && P.GETINFO()!=X)
  {P=P.GETPROX();
  }
  IF(P!=NULL)
    {
       IF(P.GETPROX()!=NULL)
```

```
        SYSTEM.OUT.PRINTLN("EL ELEMENTO POSTERIOR ES "+P.GETPROX().G
ETINFO());
       ELSE
        SYSTEM.OUT.PRINTLN("NO HAY  ELEMENTO POSTERIOR ");

       IF(P.GETANT()!=NULL)
        SYSTEM.OUT.PRINTLN("EL ELEMENTO ANTERIOR ES "+P.GETANT().GETI
NFO());
       ELSE
        SYSTEM.OUT.PRINTLN("NO HAY  ELEMENTO ANTERIOR ");
   }
  }
}
```

Tercera parte: La del *main method* para consumir la clase ListaDoble

```
CLASS TESTLISTADOBLE
  {
    PUBLIC STATIC VOID MAIN (STRING []ARG)
    { LISTADOBLE A=NEW LISTADOBLE();  //CREACION DEL OBJETO
      INT X;
      INT OP;

      DO{
        SYSTEM.OUT.PRINTLN("1 CARGA  ");
        SYSTEM.OUT.PRINTLN("2 IMPRIMIR  ");
        SYSTEM.OUT.PRINTLN("3  BUSCAR ");
        SYSTEM.OUT.PRINTLN("4   ");
        SYSTEM.OUT.PRINTLN("5   ");
        SYSTEM.OUT.PRINTLN("6   ");
        SYSTEM.OUT.PRINTLN("0 FIN  ");
        SYSTEM.OUT.PRINT("  INGRSE LA OPCION  : ");

        OP=LEER.DATOINT();

        SWITCH(OP)
        {
          CASE 1:SYSTEM.OUT.PRINTLN(" INGRESO DE DATOS A LA LISTA ");
            SYSTEM.OUT.PRINT(" INGRESE UN NUMERO ENTERO [0 = FIN]");
            X=LEER.DATOINT();

            WHILE(X!=0)
            { A.INSERTARDOBLE(X);
              SYSTEM.OUT.PRINT(" INGRESE UN NUMERO ENTERO [0 = FIN]");
              X=LEER.DATOINT();
```

```
            }
            BREAK;
          CASE 2:  A.IMPRIMIRLISTA();
            BREAK;

          CASE 3 :SYSTEM.OUT.PRINT(" INGRESE UN NUMERO PARA BUSCAR ");
            X=LEER.DATOINT();
            A.BUSCAR(X);

            BREAK;
        }
      }WHILE(OP!=0);
    }
}
```

CAPÍTULO 7

ÁRBOLES

PROPÓSITO DEL CAPÍTULO

Brindar a los estudiantes una introducción a los árboles y mostrarles la forma de implementar un árbol binario, ya sea en una estructura estática (un vector) o dinámicamente (semejante a una lista ligada).

OBJETIVOS DE APRENDIZAJE

Al finalizar el estudio de este tema, usted deberá estar en capacidad de:

- Explicar la ordenación y los diferentes algoritmos de ordenación utilizando árboles.
- Explicar cuáles son los diferentes tipos de árboles y su utilización.
- Implementar algoritmos en Java para la utilización de árboles binarios

GUÍA DE LECTURAS

Para lograr los objetivos anteriores, se le sugiere seguir la siguiente guía:

Subtema	Capítulo	Páginas
Árboles generales	17	435 - 438
Árboles binarios	17	439 - 440
Árboles y recursión	17	448 - 449
Recorrido de árboles	17	450 - 459

COMENTARIOS GENERALES

Una vez que usted ha realizado las lecturas anteriores, analice cuidadosamente los comentarios siguientes, ya que con ellos se pretende enfatizar o ampliar algunos contenidos importantes del capítulo. Existen varios aspectos importantes que debemos señalar, antes de entrar al desarrollo de los subtemas.

Un árbol es una estructura de datos, que puede definirse de forma recursiva como:

- Una estructura vacía o
- Un elemento o clave de información (nodo) más un número finito de estructuras tipo árbol, disjuntos, llamados subárboles. Si dicho número de estructuras es inferior o igual a 2, se tiene un árbol binario.

Es, por tanto, una estructura no secuencial.

Otra definición nos da el árbol como un tipo de grafo; un árbol es un grafo acíclico, conexo y no dirigido. Es decir, es un grafo no dirigido en el que existe exactamente un camino entre todo par de nodos. Esta definición permite implementar un árbol y sus operaciones empleando las representaciones que se utilizan para los grafos. Sin embargo, en esta sección no se tratará esta implementación.

Existen diferentes formas de representación:

- Mediante un grafo:

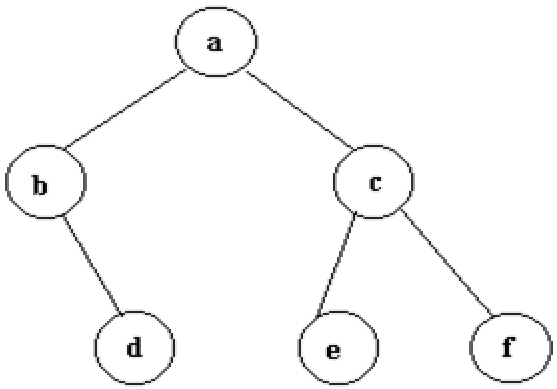


Figura No. 15

- Mediante un diagrama encolumnado:

a
  b
    d
  c
    e
    f

En computación se utiliza mucho una estructura de datos, que son los árboles binarios. Estos árboles tienen 0, 1 ó 2 descendientes como máximo. El árbol de la figura anterior es un ejemplo válido de árbol binario.

Declaración de árbol binario

Se definirá el árbol con una clave de tipo entero (puede ser cualquier otro tipo de datos) y dos hijos: izquierdo (izq) y derecho (der). Para representar los enlaces con los hijos se utilizan punteros. El árbol vacío se representará con un puntero nulo.

Un árbol binario puede declararse de la siguiente manera:

```
typedef struct tarbol
{
  int clave;
  struct tarbol *izq,*der;
} tarbol;
```

Otras declaraciones también añaden un enlace al nodo padre, pero no se estudiarán aquí.

Recorridos sobre árboles binarios

Se consideran dos tipos de recorrido: recorrido en profundidad, y recorrido en anchura o a nivel. Puesto que los árboles no son secuenciales como las listas, hay que buscar estrategias alternativas para visitar todos los nodos.

- **Recorridos en profundidad**

o ***Recorrido en preorden***: Consiste en visitar el nodo actual (visitar puede ser simplemente mostrar la clave del nodo por pantalla), y después visitar el subárbol izquierdo y, una vez visitado, visitar el subárbol derecho. Es un proceso recursivo por naturaleza.

Si se hace el recorrido en preorden del árbol de la figura 1, las visitas serían en el orden siguiente: a, b, d, c, e, f.

```
void preorden(tarbol *a)
{
  if (a != NULL) {
    visitar(a);
    preorden(a->izq);
    preorden(a->der);
  }
}
```

- ***Recorrido en inorden u orden central***: Se visita el subárbol izquierdo, el nodo actual, y después se visita el subárbol derecho. En el ejemplo de la figura 1, las visitas serían en este orden: b, d, a, e, c, f.

```
void inorden(tarbol *a)
{
  if (a != NULL) {
    inorden(a->izq);
    visitar(a);
    inorden(a->der);
  }
}
```

- ***Recorrido en postorden***: Se visitan primero el subárbol izquierdo, después el subárbol derecho y, por último el nodo actual. En el ejemplo de la figura 1, el recorrido quedaría así: d, b, e, f, c, a.

```
void postorden(arbol *a)
{
  if (a != NULL) {
    postorden(a->izq);
    postorden(a->der);
    visitar(a);
  }
}
```

La ventaja del recorrido en postorden es que permite borrar el árbol de forma consistente. Es decir, si visitar se traduce por borrar el nodo actual, al ejecutar este recorrido se borrará el árbol o subárbol que se pasa como parámetro. La razón para hacer esto es que no se debe borrar un nodo y después sus subárboles, porque al borrarlo se pueden perder los enlaces. Y, aunque no se perdieran, se rompe con la

regla de manipular una estructura de datos inexistente. Una alternativa es utilizar una variable auxiliar, pero es innecesario aplicando este recorrido.

- **Recorrido en amplitud**:

Consiste en ir visitando el árbol por niveles. Primero se visitan los nodos de nivel 1 (como máximo hay uno, la raíz), después los nodos de nivel 2, así hasta que ya no queden más.

Si se hace el recorrido en amplitud del árbol de la figura No. 15, uno visitaría los nodos en este orden: a,b,c,d,e,f.

En este caso, el recorrido no se realizará de forma recursiva sino iterativa, utilizando una cola, como estructura de datos auxiliar. El procedimiento consiste en encolar (si no están vacíos) los subárboles izquierdo y derecho del nodo extraído de la cola, y seguir desencolando y encolando hasta que la cola esté vacía.

En la codificación que viene a continuación no se implementan las operaciones sobre colas.

```
void amplitud(tarbol *a)
{
  tCola cola;   /* las claves de la cola serán de tipo árbol binario */
  arbol *aux;

  if (a != NULL) {
    CrearCola(cola);
    encolar(cola, a);
    while (!colavacia(cola)) {
      desencolar(cola, aux);
      visitar(aux);
      if (aux->izq != NULL) encolar(cola, aux->izq);
      if (aux->der != NULL) encolar(cola, aux->der);
    }
  }
}
```

Construcción de un árbol binario

Hasta el momento se ha visto la declaración y el recorrido de un árbol binario. Sin embargo, no se ha estudiado ningún método para crearlos. A continuación se

estudia un método para crear un árbol binario que no tenga claves repetidas partiendo de su recorrido en preorden e inorden, almacenados en sendos *arrays*.

Antes de explicarlo se recomienda al lector que intente hacerlo por su cuenta. Es sencillo cuando uno es capaz de construir el árbol viendo sus recorridos pero sin haber visto el árbol terminado.

Partiendo de los recorridos preorden e inorden del árbol de la figura 1, puede determinarse que la raíz es el primer elemento del recorrido en preorden. Ese elemento se busca en el *array* inorden. Los elementos en el *array* inorden entre **izq** y la raíz forman el subárbol izquierdo. Asimismo, los elementos entre **der** y la raíz forman el subárbol derecho. Por tanto se tiene este árbol:

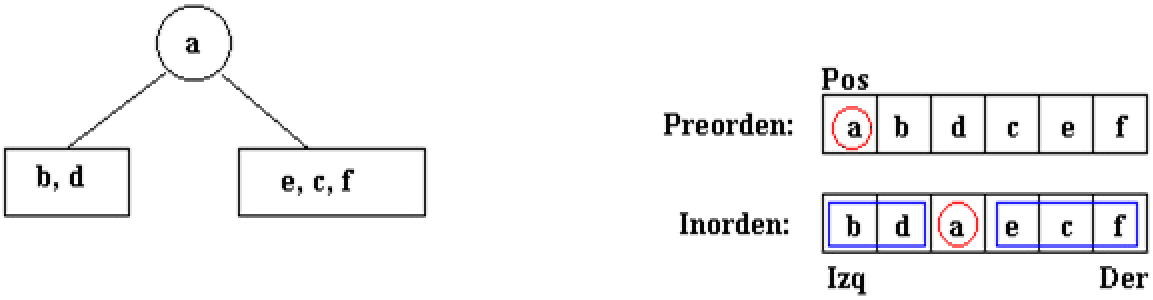


Figura No. 16

A continuación comienza un proceso recursivo. Se procede a crear el subárbol izquierdo, cuyo tamaño está limitado por los índices **izq** y **der**. La siguiente posición en el recorrido en preorden es la raíz de este subárbol. Queda esto:

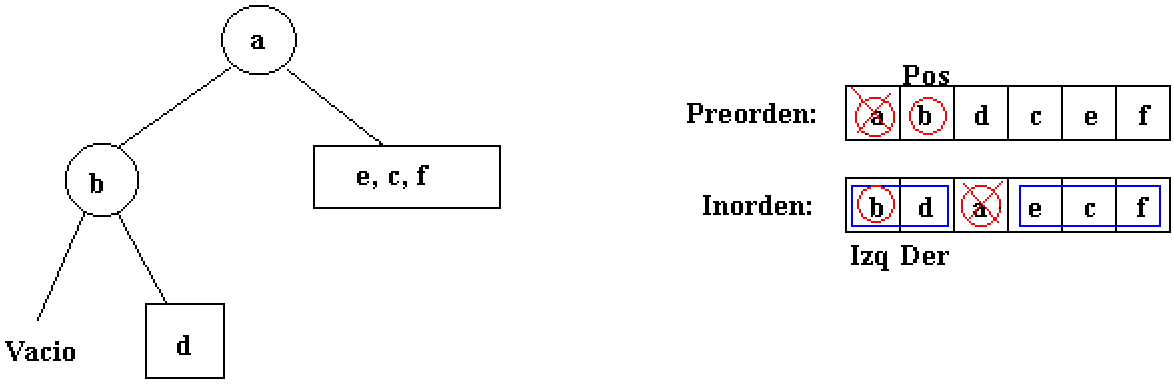


Figura No. 17

El subárbol *b* tiene un subárbol derecho, que no tiene ningún descendiente, tal y como indican los índices **izq** y **der**. Se ha obtenido el subárbol izquierdo completo de la raíz *a*, puesto que *b* no tiene subárbol izquierdo:

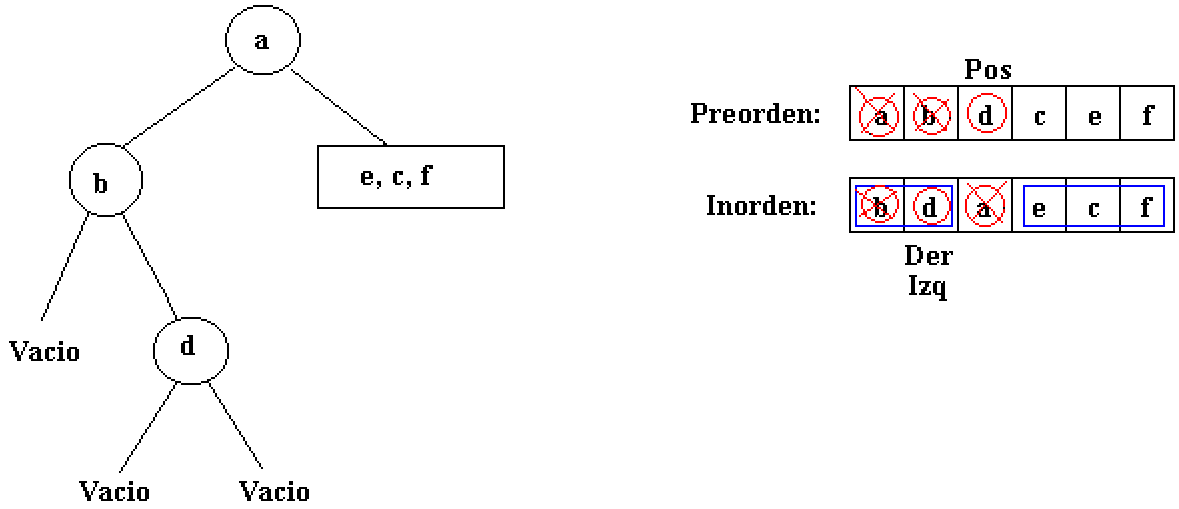


Figura No. 18

Después seguirá construyéndose el subárbol derecho a partir de la raíz *a*.

GLOSARIO

En esta área usted encontrará un glosario donde se incluyen términos y la definición de cada uno para que le sea más fácil comprender el tema de estudio.

Árbol: Estructura de datos que puede definirse de forma recursiva.

Raíz: Aquel elemento que no tiene antecesor.

Rama: Una arista entre dos nodos.

Antecesor: Un nodo X es antecesor de un nodo Y si por alguna de las ramas de X se puede llegar a Y.

Sucesor: un nodo X es sucesor de un nodo Y si por alguna de las ramas de Y se puede llegar a X.

Grado de un nodo: El número de descendientes directos que tiene. Ejemplo: *c* tiene grado 2, *d* tiene grado 0, *a* tiene grado 2.

Hoja: Nodo que no tiene descendientes: grado 0. Ejemplo: *d*

Nodo interno: Aquel que tiene al menos un descendiente.

Nivel: Número de ramas que hay que recorrer para llegar de la raíz a un nodo.

Altura: El nivel más alto del árbol.

Anchura: El mayor valor del número de nodos que hay en un nivel.

EJERCICIO SUGERIDO

Para este tema, se sugiere que usted realice el siguiente ejercicio de su libro de texto:

17.12 Implemente un comando que enliste todos los ficheros de un directorio dado.

RESOLUCIÓN DE EJERCICIO SUGERIDO

17.12. *Implemente un comando que enliste todos los ficheros de un directorio dado.*

```
package org.neos.arboles.binarios.estaticos;

import java.util.ArrayList;

import org.neos.arboles.binarios.estaticos.beans.Informacion;
import org.neos.arboles.binarios.estaticos.beans.Nodo;
import
org.neos.arboles.binarios.estaticos.constantes.ConstantesArbolBinarioEstatico;
import
org.neos.arboles.binarios.estaticos.exceptions.ExceptionDesbordamientoVector;
import org.neos.arboles.binarios.estaticos.exceptions.ExceptionDimensionInvalida;
import org.neos.arboles.binarios.estaticos.exceptions.ExceptionElementoDuplicado;
import org.neos.arboles.binarios.estaticos.exceptions.ExceptionNoEncontrado;

public class ArbolBinarioEstatico implements ConstantesArbolBinarioEstatico {
        private Nodo[] arbol;
        private int posicion_raiz;
        private ArrayList recorrido;
```

```
	public ArbolBinarioEstatico(int num_elementos) throws
ExceptionDimensionInvalida {
		if(num_elementos <= 0) {
			String msg = "La dimension no puede ser cero o
negativa!!";
			throw new ExceptionDimensionInvalida(msg);
		} else {
			this.arbol = new Nodo[num_elementos];
		}
		this.posicion_raiz    = NULL;
		this.recorrido       = null;
	}

	public boolean esArbolVacio() {
		return (NULL == posicion_raiz) ? true : false;
	}

	public int obtenerPosicionRaiz() {
		return posicion_raiz;
	}

	private int obtenerPosicionElemento(Object info, int posicion_actual) throws
ArrayIndexOutOfBoundsException {
		Nodo nodo = arbol[posicion_actual];
		Informacion obj_info    = new Informacion(info);
		Informacion obj_info_pa = new Informacion( nodo.getInformacion()
);

		if( obj_info_pa.equals(obj_info) ) {
			return posicion_actual;
		} else if( obj_info_pa.compareTo(obj_info) > 0 ) {
			return obtenerPosicionElemento(info, nodo.getIzquierdo());
		} else {
			return obtenerPosicionElemento(info, nodo.getDerecho());
		}
	}

	public int obtenerPosicionElemento(Object info) throws
ArrayIndexOutOfBoundsException {
		Nodo nodo_raiz = arbol[posicion_raiz];
		Informacion obj_info   = new Informacion(info);
		Informacion obj_info_r = new Informacion(
nodo_raiz.getInformacion() );
```

```
                if( obj_info_r.equals(obj_info) ) {
                        return posicion_raiz;
                } else if( obj_info_r.compareTo(obj_info) > 0 ) {
                        return obtenerPosicionElemento(info,
nodo_raiz.getIzquierdo());
                } else {
                        return obtenerPosicionElemento(info,
nodo_raiz.getDerecho());
                }
        }

        public boolean existeElemento(Object info) {
                boolean respuesta     = false;
                int posicion_elemento = -1;

                try {
                        posicion_elemento = obtenerPosicionElemento(info);
                        respuesta = true;
                } catch(ArrayIndexOutOfBoundsException aioob_e) {
                        respuesta = false;
                }
                return respuesta;
        }

        public Nodo obtenerElemento(Object info) {
                int posicion_elemento = -1;
                Nodo elemento         = null;

                try {
                        posicion_elemento = obtenerPosicionElemento(info);
                        elemento = arbol[posicion_elemento];
                } catch(ArrayIndexOutOfBoundsException aioob_e) {
                        ;
                }

                return elemento;
        }

        public int obtenerNumeroElementos(int posicion) {
                int num_elems = 0;

                if(NULL != posicion) {
                        Nodo nodo = arbol[posicion];
```

```
                        num_elems +=
obtenerNumeroElementos(nodo.getIzquierdo());
                        num_elems++;
                        num_elems +=
obtenerNumeroElementos(nodo.getDerecho());
                }

                return (num_elems);
        }

        public int obtenerPeso(int raiz) {
                int peso      = 0;
                Nodo nodo     = null;
                int num_h_izq = 0;
                int num_h_der = 0;

                if(NULL != raiz) {
                        nodo = arbol[raiz];

                        num_h_izq =
obtenerNumeroElementos(nodo.getIzquierdo());
                        num_h_der =
obtenerNumeroElementos(nodo.getDerecho());

                        peso = num_h_izq + num_h_der;
                }

                return peso;
        }

        public int obtenerAltura(int raiz) {
                int altura    = 0;
                Nodo nodo     = null;
                int ref_h_izq;
                int ref_h_der;
                int altura_r_izq = 0;
                int altura_r_der = 0;

                if(NULL != raiz) {
                        altura = 1;

                        nodo = arbol[raiz];
                        ref_h_izq = nodo.getIzquierdo();
                        ref_h_der = nodo.getDerecho();
```

```
                        if( (ref_h_izq != NULL) && (ref_h_der != NULL) ) {
                                altura_r_izq = obtenerAltura(ref_h_izq);
                                altura_r_der = obtenerAltura(ref_h_der);
                                if(altura_r_izq >= altura_r_der) {
                                        altura += altura_r_izq;
                                } else {
                                        altura += altura_r_der;
                                }
                        } else if( (ref_h_izq != NULL) && (ref_h_der == NULL) ) {
                                altura += obtenerAltura(ref_h_izq);
                        } else if( (ref_h_izq == NULL) && (ref_h_der != NULL) ) {
                                altura += obtenerAltura(ref_h_der);
                        } else if( (ref_h_izq == NULL) && (ref_h_der == NULL) ) {
                                altura -= 1;
                        }
                }

                return altura;
        }

        public int obtenerCapacidadArbol() {
                return arbol.length;
        }

        private int obtenerPosicionLibre() {
                int i, longitud = arbol.length;

                for(i = 0; i < longitud; i++) {
                        if(null == arbol[i]) {
                                return i;
                        }
                }
                return NULL;
        }

        private boolean insertarElemento(Object info, int posicion_actual) {
                Nodo nodo        = arbol[posicion_actual];
                Nodo nuevo_nodo  = null;
                int posicion     = NULL;
                Informacion obj_info    = new Informacion(info);
                Informacion obj_info_pa = new Informacion( nodo.getInformacion()
);

                if( obj_info_pa.compareTo(obj_info) > 0 ) {
```

```
                        if(nodo.getIzquierdo() == NULL) {
                                posicion = obtenerPosicionLibre();
                                nodo.setIzquierdo(posicion);

                                nuevo_nodo = new Nodo(info);
                                nuevo_nodo.setPadre(posicion_actual);
                                arbol[posicion] = nuevo_nodo;

                                return true;
                        } else {
                                return insertarElemento(info, nodo.getIzquierdo());
                        }
                } else {
                        if(nodo.getDerecho() == NULL) {
                                posicion = obtenerPosicionLibre();
                                nodo.setDerecho(posicion);

                                nuevo_nodo = new Nodo(info);
                                nuevo_nodo.setPadre(posicion_actual);
                                arbol[posicion] = nuevo_nodo;

                                return true;
                        } else {
                                return insertarElemento(info, nodo.getDerecho());
                        }
                }
        }

        public boolean insertarElemento(Object info)
                        throws ExceptionDesbordamientoVector,
ExceptionElementoDuplicado {

                        Nodo nuevo_nodo    = null;
                        boolean se_inserto = false;

                        if( this.esArbolVacio() ) {
                                nuevo_nodo = new Nodo(info);
                                arbol[0] = nuevo_nodo;

                                this.posicion_raiz = 0;
                                se_inserto = true;
                        } else {
```

```
                                        if(obtenerNumeroElementos(posicion_raiz) ==
arbol.length) {
                                                throw new
ExceptionDesbordamientoVector("El árbol esta lleno, no pueden agregarse más
elementos!!");
                                        } else {
                                                if( existeElemento(info) ) {
                                                        throw new
ExceptionElementoDuplicado("El elemento ya existe dentro del árbol!!");
                                                } else {
                                                        se_inserto =
insertarElemento(info, posicion_raiz);
                                                }
                                        }
                                }

                                return se_inserto;
        }

        public boolean borrarElemento(Object info) throws ExceptionNoEncontrado {
                Nodo nodo       = null;
                int posicion_elem = NULL;
                boolean se_borro  = false;
                Nodo nodo_padre   = null;
                Nodo nodo_aux     = null;
                int posicion_aux  = NULL;

                if( this.esArbolVacio() ) {
                        throw new ExceptionNoEncontrado("El árbol esta vacío!!");
                } else {
                        if( existeElemento(info) ) {
                                posicion_elem = obtenerPosicionElemento(info);
                                nodo = arbol[posicion_elem];

                                if(nodo.getPadre() != NULL) { // Si no es el nodo
raiz
                                        if( (nodo.getIzquierdo() != NULL) ||
(nodo.getDerecho() != NULL) ) {
                                                se_borro =
borrarNodoInterior(nodo, posicion_elem);
                                        } else {
                                                se_borro =
borrarNodoHoja(nodo, posicion_elem);
                                        }
                                } else {
```

```
                                        se_borro = borrarNodoRaiz(nodo,
posicion_elem);
                                }
                        } else {
                                throw new ExceptionNoEncontrado("El elemento
no existe dentro del árbol!!");
                        }
                }

                return se_borro;
        }

        private boolean insertarElemento(Nodo nuevo_nodo, int posicion_actual) {
                Nodo nodo       = arbol[posicion_actual];
                int posicion    = NULL;
                Informacion obj_info    = new Informacion(
nuevo_nodo.getInformacion() );
                Informacion obj_info_pa = new Informacion( nodo.getInformacion()
);

                if( obj_info_pa.compareTo(obj_info) > 0 ) {
                        if(nodo.getIzquierdo() == NULL) {
                                posicion = obtenerPosicionLibre();
                                nodo.setIzquierdo(posicion);

                                nuevo_nodo.setPadre(posicion_actual);
                                arbol[posicion] = nuevo_nodo;

                                return true;
                        } else {
                                return insertarElemento(nuevo_nodo,
nodo.getIzquierdo());
                        }
                } else {
                        if(nodo.getDerecho() == NULL) {
                                posicion = obtenerPosicionLibre();
                                nodo.setDerecho(posicion);

                                nuevo_nodo.setPadre(posicion_actual);
                                arbol[posicion] = nuevo_nodo;

                                return true;
                        } else {
```

```
                                        return insertarElemento(nuevo_nodo,
nodo.getDerecho());
                                }
                        }
                }

                private boolean borrarNodoInterior(Nodo nodo, int posicion_nodo) {
                        boolean respuesta = false;
                        Nodo nodo_padre   = null;
                        Nodo nodo_aux     = null;
                        int posicion_aux  = NULL;
                        int posicion_ins  = NULL;

                        if( (nodo.getIzquierdo() != NULL) && (nodo.getDerecho() != NULL) )
{
                                nodo_padre = arbol[nodo.getPadre()];
                                posicion_aux = nodo.getIzquierdo();
                                nodo_aux = arbol[posicion_aux];
                                posicion_ins = nodo.getDerecho();

                                if(nodo_padre.getIzquierdo() == posicion_nodo) {
                                        nodo_padre.setIzquierdo( nodo.getDerecho() );
                                } else {
                                        nodo_padre.setDerecho( nodo.getDerecho() );
                                }

                                nodo = new Nodo(nodo_aux.getInformacion());
                                nodo.setDerecho(nodo_aux.getDerecho());
                                nodo.setIzquierdo(nodo_aux.getIzquierdo());

                                arbol[posicion_nodo] = null;
                                arbol[posicion_aux]  = null;
                                insertarElemento(nodo, posicion_ins);
                                respuesta = true;
                        } else if(nodo.getIzquierdo() != NULL) {
                                nodo_padre = arbol[nodo.getPadre()];

                                if(posicion_nodo == nodo_padre.getDerecho()) {
                                        nodo_padre.setDerecho( nodo.getIzquierdo() );
                                } else {
                                        nodo_padre.setIzquierdo( nodo.getIzquierdo() );
                                }

                                arbol[posicion_nodo] = null;
                                respuesta = true;
```

```
                } else {
                        nodo_padre = arbol[nodo.getPadre()];

                        if(posicion_nodo == nodo_padre.getDerecho()) {
                                nodo_padre.setDerecho( nodo.getDerecho() );
                        } else {
                                nodo_padre.setIzquierdo( nodo.getDerecho() );
                        }

                        arbol[posicion_nodo] = null;
                        respuesta = true;
                }
                return respuesta;
        }

        private boolean borrarNodoHoja(Nodo nodo, int posicion_nodo) {
                boolean se_borro = false;
                Nodo nodo_padre  = null;

                nodo_padre = arbol[nodo.getPadre()];
                if(posicion_nodo == nodo_padre.getDerecho()) {
                        nodo_padre.setDerecho(NULL);
                } else {
                        nodo_padre.setIzquierdo(NULL);
                }
                arbol[posicion_nodo] = null;
                se_borro = true;

                return se_borro;
        }

        private boolean borrarNodoRaiz(Nodo nodo, int posicion_nodo) {
                boolean se_borro = false;
                Nodo nodo_aux    = null;
                int posicion_aux = NULL;

                if( (nodo.getIzquierdo() != NULL) && (nodo.getDerecho() != NULL) )
{
                        this.posicion_raiz = nodo.getDerecho();
                        nodo_aux = arbol[posicion_raiz];
                        nodo_aux.setPadre(NULL);

                        posicion_aux = nodo.getIzquierdo();
                        nodo_aux = arbol[posicion_aux];
```

```
                    nodo = new Nodo(nodo_aux.getInformacion());
                    nodo.setDerecho(nodo_aux.getDerecho());
                    nodo.setIzquierdo(nodo_aux.getIzquierdo());

                    arbol[posicion_nodo] = null;
                    arbol[posicion_aux]  = null;
                    insertarElemento(nodo, posicion_raiz);
                    se_borro = true;
            } else if(nodo.getIzquierdo() != NULL) {
                    this.posicion_raiz = nodo.getIzquierdo();
                    nodo_aux = arbol[posicion_raiz];
                    nodo_aux.setPadre(NULL);

                    arbol[posicion_nodo] = null;
                    se_borro = true;
            } else if(nodo.getDerecho() != NULL) {
                    this.posicion_raiz = nodo.getDerecho();
                    nodo_aux = arbol[posicion_raiz];
                    nodo_aux.setPadre(NULL);

                    arbol[posicion_nodo] = null;
                    se_borro = true;
            } else { // no tiene ramas
                    arbol[posicion_raiz] = null;
                    posicion_raiz       = NULL;
                    se_borro = true;
            }
            return se_borro;
    }

    public ArrayList recorridoPreorden(int posicion) {
            if(posicion == posicion_raiz) {
                    recorrido = new ArrayList();
            }

            if(NULL != posicion) {
                    Nodo nodo = arbol[posicion];
                    recorrido.add(nodo.getInformacion());
                    recorridoPreorden(nodo.getIzquierdo());
                    recorridoPreorden(nodo.getDerecho());
            }
            return recorrido;
    }

    public ArrayList recorridoInorden(int posicion) {
```

```
                if(posicion == posicion_raiz) {
                        recorrido = new ArrayList();
                }

                if(NULL != posicion) {
                        Nodo nodo = arbol[posicion];
                        recorridoInorden(nodo.getIzquierdo());
                        recorrido.add(nodo.getInformacion());
                        recorridoInorden(nodo.getDerecho());
                }
                return recorrido;
        }

        public ArrayList recorridoPostorden(int posicion) {
                if(posicion == posicion_raiz) {
                        recorrido = new ArrayList();
                }

                if(NULL != posicion) {
                        Nodo nodo = arbol[posicion];
                        recorridoPostorden(nodo.getIzquierdo());
                        recorridoPostorden(nodo.getDerecho());
                        recorrido.add(nodo.getInformacion());
                }
                return recorrido;
        }
}
```

CAPÍTULO 8

ÁRBOLES BINARIOS DE BÚSQUEDA

PROPÓSITO DEL CAPÍTULO

Al finalizar este capítulo el estudiante deberá conocer los tipos de datos más usuales en programación, sus implementaciones más comunes y su utilidad. Concretamente se espera que el estudiante sea capaz de organizar un determinado volumen de datos de la forma más racional posible en función de los requerimientos del problema que ha de resolver.

OBJETIVOS DE APRENDIZAJE

Al finalizar el estudio de este capítulo, el estudiante deberá estar en capacidad de:

- Realizar un análisis y síntesis.
- Resolver problemas.
- Aplicar los conocimientos teóricos en la práctica.

GUÍA DE LECTURA

Para lograr los objetivos anteriores, se le sugiere seguir la siguiente guía:

Subtema	Capítulo	Páginas
Árboles binarios de búsqueda	18	467 - 477

COMENTARIOS GENERALES

Una vez que usted ha realizado las lecturas anteriores, analice cuidadosamente los comentarios siguientes, ya que con ellos se pretende enfatizar o ampliar algunos contenidos importantes del capítulo. Existen varios aspectos importantes que debemos señalar, antes de entrar al desarrollo de los subtemas.

Un árbol binario de búsqueda es aquel que es:

- Una estructura vacía o

- Un elemento o clave de información (nodo) más un número finito -a lo sumo dos- de estructuras tipo árbol, disjuntos, llamados subárboles y que además cumplen lo siguiente:

    - Todas las claves del subárbol izquierdo al nodo son menores que la clave del nodo.
    - Todas las claves del subárbol derecho al nodo son mayores que la clave del nodo.
    - Ambos subárboles son árboles binarios de búsqueda.

Un ejemplo de árbol binario de búsqueda:

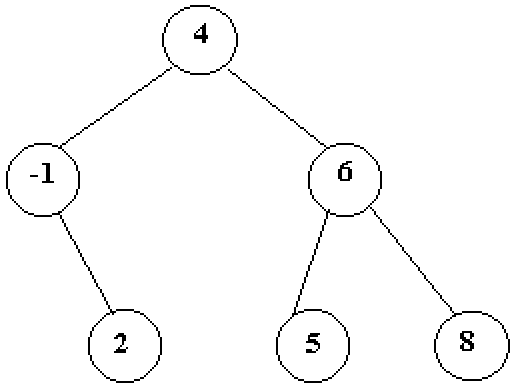


Figura No. 19

Al definir el tipo de datos que representa la clave de un nodo dentro de un árbol binario de búsqueda es necesario que en dicho tipo se pueda establecer una relación de orden: por ejemplo, suponer que el tipo de datos de la clave es un puntero (da igual a lo que apunte). Si se codifica el árbol en Pascal, no se puede establecer una relación de orden para las claves, puesto que Pascal no admite determinar si un puntero es mayor o menor que otro.

En el ejemplo de la figura 5 las claves son números enteros. Dada la raíz *4*, las claves del subárbol izquierdo son menores que 4, y las claves del subárbol derecho son mayores que 4. Esto se cumple también para todos los subárboles. Si se hace el recorrido de este árbol en orden central, se obtiene una lista de los números ordenada de menor a mayor.

Cuestión: ¿Qué hay que hacer para obtener una lista de los números ordenada de mayor a menor?

Una ventaja fundamental de los árboles de búsqueda es que son, en general, mucho más rápidos para localizar un elemento que una lista enlazada. Por tanto, son más rápidos para insertar y borrar elementos. Si el árbol está perfectamente equilibrado -esto es, la diferencia entre el número de nodos del subárbol izquierdo y el número de nodos del subárbol derecho es a lo sumo 1, para todos los nodos- entonces el número de comparaciones necesarias para localizar una clave es aproximadamente de logN en el peor caso. Además, el algoritmo de inserción en un árbol binario de búsqueda tiene la ventaja -sobre los *arrays* ordenados, donde se emplearía búsqueda dicotómica para localizar un elemento- de que no necesita hacer una reubicación de los elementos de la estructura para que esta siga ordenada después de la inserción. Dicho algoritmo funciona avanzando por el árbol, escogiendo la rama izquierda o derecha en función de la clave que se inserta y la clave del nodo actual, hasta encontrar su ubicación. Por ejemplo, insertar la clave 7 en el árbol de la figura 5 requiere avanzar por el árbol hasta llegar a la clave 8, e introducir la nueva clave en el subárbol izquierdo a *8*.

El algoritmo de borrado en árboles es algo más complejo, pero más eficiente que el de borrado en un *array* ordenado.

> Suponer que se tiene un árbol vacío, que admite claves de tipo entero. Suponer que se van a ir introduciendo las claves de forma ascendente. Ejemplo: 1, 2, 3, 4, 5, 6.

Se crea un árbol cuya raíz tiene la clave 1. Se inserta la clave 2 en el subárbol derecho de *1*. A continuación se inserta la clave 3 en el subárbol derecho de *2*.

Continuando las inserciones, puede verse que el árbol degenera en una lista secuencial, reduciendo drásticamente su eficacia para localizar un elemento. De cualquier manera, es poco probable que se dé un caso de este tipo en la práctica. Si las claves por introducir llegan de forma más o menos aleatoria, entonces la implementación de operaciones sobre un árbol binario de búsqueda que vienen a continuación son, en general, suficientes.

Existen variaciones sobre estos árboles, como los AVL o Red-Black (no se tratan aquí) que, sin llegar a cumplir al 100% el criterio de árbol perfectamente equilibrado, evitan problemas como el de obtener una lista degenerada.

Operaciones básicas sobre árboles binarios de búsqueda

- **Búsqueda**

Si el árbol no es de búsqueda, es necesario emplear uno de los recorridos anteriores sobre el árbol para localizarlo. El resultado es idéntico al de una búsqueda secuencial. Aprovechando las propiedades del árbol de búsqueda, se puede acelerar la localización. Simplemente hay que descender a lo largo del árbol a izquierda o derecha, dependiendo del elemento que se busca.

```
boolean buscar(tarbol *a, int elem)
{
  if (a == NULL) return FALSE;
  else if (a->clave < elem) return buscar(a->der, elem);
  else if (a->clave > elem) return buscar(a->izq, elem);
  else return TRUE;
}
```

- **Inserción**

La inserción tampoco es complicada. Es más, resulta prácticamente idéntica a la búsqueda. Cuando se llega a un árbol vacío, se crea el nodo en el puntero que se pasa como parámetro por referencia. De esta manera, los nuevos enlaces mantienen la coherencia. Si el elemento por insertar ya existe, entonces no se hace nada.

```
void insertar(tarbol **a, int elem)
{
  if (*a == NULL) {
    *a = (arbol *) malloc(sizeof(arbol));
    (*a)->clave = elem;
    (*a)->izq = (*a)->der = NULL;
  }
  else if ((*a)->clave < elem) insertar(&(*a)->der, elem);
  else if ((*a)->clave > elem) insertar(&(*a)->izq, elem);
}
```

- **Borrado**

La operación de borrado resulta ser algo más complicada. Se recuerda que el árbol debe seguir siendo de búsqueda tras el borrado. Pueden darse tres casos, una vez encontrado el nodo que se ha de borrar:

a. El nodo no tiene descendientes. Simplemente se borra.

b. El nodo tiene al menos un descendiente por una sola rama. Se borra dicho nodo, y su primer descendiente se asigna como hijo del padre del nodo borrado.

   Ejemplo: En el árbol de la figura 5 se borra el nodo cuya clave es -1. El árbol resultante es:

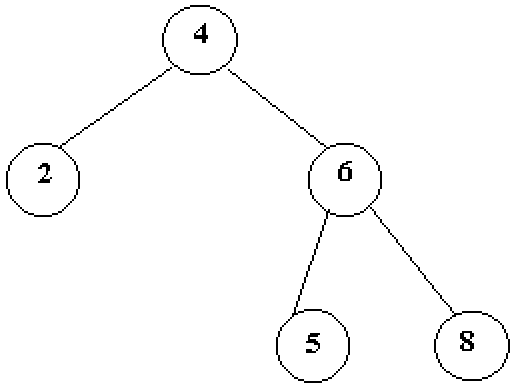


Figura No. 20

c. El nodo tiene al menos un descendiente por cada rama. Al borrar dicho nodo, es necesario mantener la coherencia de los enlaces, además de seguir manteniendo la estructura como un árbol binario de búsqueda. La solución consiste en sustituir la información del nodo que se borra por el de una de las hojas, y borrar a continuación dicha hoja. ¿Puede ser cualquier hoja? No, debe ser la que contenga una de estas dos claves:

   - La **mayor** de las claves **menores** al nodo que se borra. Suponer que se quiere borrar el nodo *4* del árbol de la figura 5. Se sustituirá la clave 4 por la clave 2.

   - La **menor** de las claves **mayores** al nodo que se borra. Suponer que se quiere borrar el nodo *4* del árbol de la figura 5. Se sustituirá la clave 4 por la clave 5.

El algoritmo de borrado que se implementa a continuación realiza la sustitución por la mayor de las claves menores (aunque se puede escoger la otra opción sin pérdida de generalidad). Para lograr esto, es necesario descender primero a la izquierda del nodo que se va a borrar, y después avanzar siempre a la derecha hasta encontrar un nodo hoja. A continuación, se muestra gráficamente el proceso de borrar el nodo de clave 4:

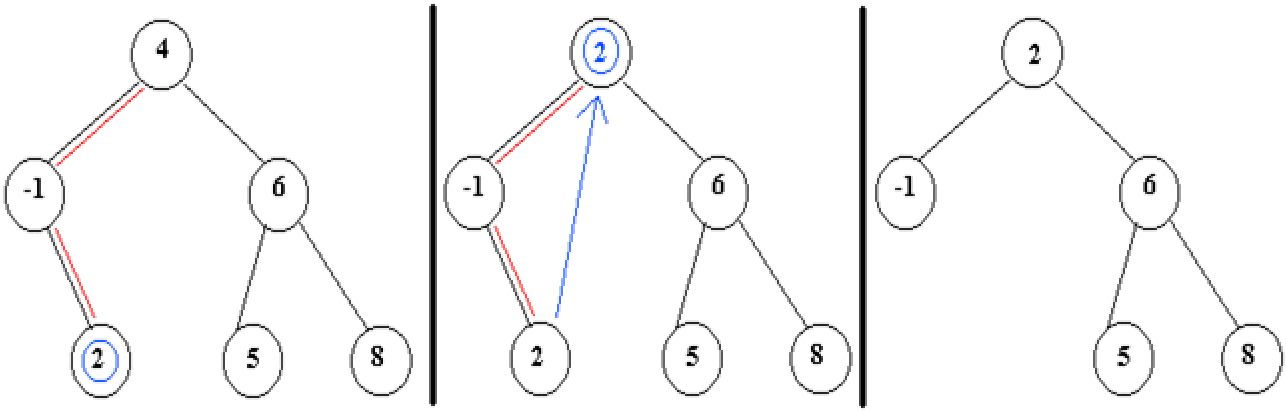


Figura No. 21

Codificación: El procedimiento sustituir es el que desciende por el árbol cuando se da el caso del nodo con descendientes por ambas ramas.

```
void borrar(tarbol **a, int elem)
{
  void sustituir(tarbol **a, tarbol **aux);
  tarbol *aux;

  if (*a == NULL) /* no existe la clave */
    return;

  if ((*a)->clave < elem) borrar(&(*a)->der, elem);
  else if ((*a)->clave > elem) borrar(&(*a)->izq, elem);
  else if ((*a)->clave == elem) {
    aux = *a;
    if ((*a)->izq == NULL) *a = (*a)->der;
    else if ((*a)->der == NULL) *a = (*a)->izq;
```

```
   else sustituir(&(*a)->izq, &aux); /* se sustituye por la mayor de las menores */

   free(aux);
  }
}
```

GLOSARIO

En esta área usted encontrará un glosario donde se incluyen términos y la definición de cada uno para que le sea más fácil comprender el tema de estudio.

Árbol binario de búsqueda: Estructura de datos que permite la inserción, búsqueda y eliminación en un tiempo medio de O(log N).

B-árbol: Estructura de datos más popular para la búsqueda de datos en disco.

Longitud del camino externo: Suma del coste de acceso a todos los nodos externos del árbol de un árbol binario.

Longitud del camino interno: Suma de las profundidades de los nodos de un árbol binario.

Nivel de un nodo: Número de enlaces izquierdos que hay en el camino desde el nodo hasta el centinela nodoNulo.

EJERCICIO SUGERIDO

Para este tema, se sugiere que usted realice el siguiente ejercicio de su libro de texto:

18.16 Implemente una forma recursiva utilizando árboles

```
import java.awt.*;
import java.awt.event.*;
import com.sun.java.swing.*;
import com.sun.java.swing.tree.*;
```

RESOLUCIÓN DE EJERCICIO SUGERIDO

18.16 *Implemente una forma recursiva utilizando árboles*

```
import java.awt.*;
import java.awt.event.*;
```

```
        import com.sun.java.swing.*;
        import com.sun.java.swing.tree.*;

// Esta clase toma un array de Strings, haciendo que el primer elemento
// del array sea un nodo y el resto sean ramas de ese nodo
// Con ello se consiguen las ramas del árbol general cuando se pulsa
// el botón de test
class Rama {
  DefaultMutableTreeNode r;
  public Rama( String datos[] ) {
    r = new DefaultMutableTreeNode( datos[0] );
    for( int i=1; i < datos.length; i++ )
      r.add( new DefaultMutableTreeNode( datos[i] ) );
  }

  public DefaultMutableTreeNode node() {
    return( r );
  }
}

public class java1414 extends JPanel {
  String datos[][] = {
    { "Colores","Rojo","Verde","Azul" },
    { "Sabores","Salado","Dulce","Amargo" },
    { "Longitud","Corta","Media","Larga" },
    { "Intensidad","Alta","Media","Baja" },
    { "Temperatura","Alta","Media","Baja" },
    { "Volumen","Alto","Medio","Bajo" },
  };
  static int i=0;
  DefaultMutableTreeNode raiz,rama,seleccion;
  JTree arbol;
  DefaultTreeModel modelo;

  public java1414() {
    setLayout( new BorderLayout() );
    raiz = new DefaultMutableTreeNode( "raiz" );
    arbol = new JTree( raiz );
    // Se añade el árbol y se hace sobre un ScrollPane, para
    // que se controle automáticamente la longitud del árbol
    // cuando está desplegado, de forma que aparecerá una
    // barra de desplazamiento para poder visualizarlo en su
    // totalidad
    add( new JScrollPane( arbol ),BorderLayout.CENTER );
    // Se obtiene el modelo del árbol
```

```
    modelo =(DefaultTreeModel)arbol.getModel();
    // Y se añade el botón que va a ir incorporando ramas
    // cada vez que se pulse
    JButton botonPrueba = new JButton( "Pulsame" );
    botonPrueba.addActionListener( new ActionListener() {
      public void actionPerformed( ActionEvent evt ) {
        if( i < datos.length ) {
          rama = new Rama( datos[i++] ).node();
          // Control de la útlima selección realizada
          seleccion = (DefaultMutableTreeNode)
            arbol.getLastSelectedPathComponent();
          if( seleccion == null )
                seleccion = raiz;
              // El modelo creará el evento adecuado, y en respuesta
              // a él, el árbol se actualizará automáticamente
          modelo.insertNodeInto( rama,seleccion,0 );
        }
      }
    } );

    // Cambio del color del botón
    botonPrueba.setBackground( Color.blue );
    botonPrueba.setForeground( Color.white );
    // Se crea un panel para contener al botón
    JPanel panel = new JPanel();
    panel.add( botonPrueba );
    add( panel,BorderLayout.SOUTH );
  }

  public static void main( String args[] ) {
    JFrame frame = new JFrame( "Tutorial de Java, Swing" );
    frame.addWindowListener( new WindowAdapter() {
      public void windowClosing( WindowEvent evt ) {
        System.exit( 0 );
      }
    });
    frame.getContentPane().add( new java1414(),BorderLayout.CENTER );
    frame.setSize( 200,500 );
    frame.setVisible( true );
  }
}
```

BIBLIOGRAFÍA

- Weiss, Mark Allen (2000). *Estructura de Datos en JAVA TM*. Primera edición. Pearson Educación S.A , España. 776 Pp.

- Castro Chaves, Karol (2007). *La orientación al curso de Estructura de Datos.* Costa Rica, EUNED, Universidad Estatal a Distancia.