Ingeniería de Sistemas

ESTRUCTURAS DE DATOS

Fundamentación práctica

Miguel Hernández Bejarano
Luis Eduardo Baquero Rey

eduU

Ingeniería de Sistemas

ESTRUCTURAS DE DATOS

Fundamentación práctica

Miguel Hernández Bejarano
Luis Eduardo Baquero Rey

edU®
Conocimiento a su alcance
Bogotá - México, D.F.

Hernández Bejarano, Miguel, *et al.*

Estructuras de datos/ Miguel Hernández Baquero y Luis Eduardo Baquero Rey -- 1a. edición. Bogotá: Ediciones de la U, 2021
404 p. ; 24 cm.
ISBN 978-958-792-270-7 e-ISBN 978-958-792-271-4
1. Ingeniería de Sistemas 2. Programación 3. Datos I. Tít.
519.7 cd 24 ed.

Área: Ingeniería de sistemas
Primera edición: Bogotá, Colombia, junio de 2021
ISBN. 978-958-792-270-7

Ediciones de la U es una empresa editorial que, con una visión moderna y estratégica de las tecnologías, desarrolla, promueve, distribuye y comercializa contenidos, herramientas de formación, libros técnicos y profesionales, e-books, e-learning o aprendizaje en línea, realizados por autores con amplia experiencia en las diferentes áreas profesionales e investigativas, para brindar a nuestros usuarios soluciones útiles y prácticas que contribuyan al dominio de sus campos de trabajo y a su mejor desempeño en un mundo global, cambiante y cada vez más competitivo.

Coordinación editorial: Adriana Gutiérrez M.
Carátula: Ediciones de la U
Impresión: DGP Editores SAS
Calle 63 No. 70 D - 34, Pbx. (571) 7217756

Impreso y hecho en Colombia
Printed and made in Colombia

Dedicatoria

a Dios,
a mi esposa Flor Ángela
e hijos (Jefferson, Julieth y Oscar)
y a mi nieto Alejandro.
Miguel

A Dios
y a mis hijos Luis Alejandro, Sebastián,
Rosa Valentina, Cristal e Isabela.
Eduardo

Contenido

Prólogo 13

Capítulo 1. Excepciones y aserciones 17

1.1 Temática a desarrollar 17
1.2 Introducción 17
1.3 Tipos de excepciones 18
1.4 Sentencias try, catch, finally 19
1.5 Implementación de las excepciones 19
1.6 La importancia de usar excepciones 20
1.7 Excepciones comunes 20
1.8 Ejercicios propuestos 22
1.9 Aserciones 23
1.10 Ejercicios propuestos 26

Capítulo 2. Recursividad y estructuras de datos 27

2.1 Temáticas a desarrollar 27
2.2 Introducción 27
2.3 Características de la recursividad 28
2.4 Tipos de recursividad 29
2.5 Ejercicios propuestos 36
2.6 Estructura de datos 36
2.6.1 Estructuras de datos estáticas 37
2.6.2 Estructuras de datos dinámicas 38

Capítulo 3. Arreglos unidimensionales o vectores 41

3.1 Temática a desarrollar 41
3.2 Introducción 41
3.3 Arreglos 42
3.3.1 Características de un arreglo 42

3.3.2 Tipos de arreglos 43
3.4 Arreglos unidimensionales o vectores 44
3.5 Operaciones con vectores 46
3.6 Implementación de operaciones con vectores 51
3.7 Ordenamiento de arreglos 60
3.8 Introducción a la complejidad computacional 67
3.8.1 Complejidad ciclo for 69
3.9 Ejercicios propuestos 75
3.10 Proyectos propuestos 92

Capítulo 4. Arreglos bidimensionales o matrices 97

4.1 Temática a desarrollar 97
4.2 Introducción 97
4.3 Declaración de matrices en Java 98
4.4 Operaciones con matrices 100
4.5 Ejercicios propuestos 109

Capítulo 5. Cadenas 121

5.1 Temática a desarrollar 121
5.2 Introducción 121
5.3 Clase String 122
5.4 Clase StringTokenizer 130
5.5 Clase StringBuffer 131
5.6 Arreglos de objetos 133
5.7 Ejercicios propuestos 135

Capítulo 6. Listas con enlace sencillo 141

6.1 Temática a desarrollar 141
6.2 Introducción 141
6.3 Estructuras de datos dinámicas lineales 141
6.4 Representación gráfica de un nodo 142
6.5 Representación gráfica de una lista 143
6.6 Operaciones en listas enlazadas 144
6.7 Construcción de una lista en Java 148
6.7.1 Creación de un objeto de la clase Nodo 149
6.7.2 Implementación de operaciones básicas 151
6.7.3 Modelamiento del problema 160
6.7.4 Clase Nodo 161
6.7.5 La clase Lista 163

6.7.6 Validar lista 166
6.7.7 Clase Aplicación Lista 167
6.7.8 Ejecución de la aplicación 171
6.8 Ejercicios propuestos 172

Capítulo 7. Listas de acceso restringido – Estructura pila 177

7.1 Temática a desarrollar 177
7.2 Introducción 177
7.3 Definición de una pila 178
7.4 Operaciones con las pilas 180
7.5 Representación de una pila 182
7.6 Implementación de las operaciones básicas en una pila 183
7.6.1 Validación de contenido 183
7.6.2 Método de inserción o push 183
7.6.3 Método de borrado o pop 184
7.6.4 Modelamiento y codificación 186
7.7 Ejercicios propuestos 193

Capítulo 8. Listas de acceso restringido – Estructura cola 197

8.1 Temática a desarrollar 197
8.2 Introducción 197
8.3 Operaciones básicas 198
8.4 Implementación de las operaciones básicas 199
8.5 Ejercicios propuestos 211

Capítulo 9. Listas con doble enlace 213

9.1 Temática a desarrollar 213
9.2 Introducción 213
9.3 Listas doblemente enlazadas 214
9.4 Operaciones con las listas de doble enlace 216
9.5 Implementación de las operaciones 220
9.6 Ejercicios propuestos 233

Capítulo 10. Listas circulares enlazadas 235

10.1 Temática a desarrollar 235
10.2 Introducción 235
10.3 Lista circular sencilla 235
10.4 Operaciones con las listas de circulares sencillas 237

10.5 Modelamiento e implementación de operaciones 240
10.6 Lista circular doblemente enlazada 244
10.7 Operaciones con las listas de circulares doblemente enlazadas 245
10.8 Ejercicios propuestos 246

Capítulo 11. Estructuras de datos dinámicas no lineales 249

11.1 Temática a desarrollar 249
11.2 Introducción 249
11.3 Árboles 250
11.3.1 Características de los árboles 250
11.3.2 Representación gráfica de un árbol 251
11.4 Árbol binario 252
11.4.1 Elementos de un árbol 252
11.4.2 Representación de un árbol binario en la memoria 253
11.4.3 Operaciones en un árbol binario 254
11.5 Arboles binarios de búsqueda 254
11.5.1 Creación de un ABB 254
11.5.2 Recorridos en los ABB 257
11.6. Modelamiento e implementación en un ABB 260
11.7 Arboles AVL 269
11.7.1 Operaciones con árboles AVL 269
11.7.2 Rotaciones 270
11.7.3 Factor de equilibro 270
11.8 Árboles n-arios 283
11.8.1 Representación gráfica del nodo de un árbol n-ario 284
11.8.2 Representación gráfica en memoria de un árbol n-ario 284
11.8.3 Recorridos de un árbol n-ario 284
11.8.4 Árbol genealógico 285
11.9 Ejercicios propuestos 285

Capítulo 12. Grafos 287

12.1 Temática a desarrollar 287
12.2 Introducción 287
12.3 Matriz de adyacencia 288
12.4 Lista de adyacencia 291
12.5 Recorridos de los grafos 294
12.5.1 Recorrido en profundidad 294
12.5.2 Recorrido en anchura 297
12.6 Árboles de expansión mínima 300
12.7 Algoritmos de grafos 301

12.7.1 Algoritmo de Dijkstra 301
12.7.2 Algoritmo de Prim 308
12.7.3 Algoritmo de Kruskal 315
12.8 Implementaciones 318
12.9 Ejercicios propuestos 334

Capítulo 13. Colecciones 337

13.1 Temática a desarrollar 337
13.2 Introducción 337
13.3 Colecciones 339
13.4 Jerarquía de las colecciones 340
13.5 La interfaz Collection 342
13.6 La interfaz List 343
13.7 La interfaz Set 354
13.8 HashSet 354
13.9 La interfaz Map 361
13.10 HashMap<Clave,Valor> 361
13.10.1 HashTable<Clave,Valor> 362
13.10.2 TreeMap<Clave,Valor> 366
13.11 La interfaz Comparable 377
13.12 La interfaz Queue y Deque 379
13.13 Programación de las colecciones 379
13.14 Ejercicios propuestos 384

Capítulo 14. Programación genérica 389

14.1 Temática a desarrollar 389
14.2 Introducción 389
14.3 Clase genérica 389
14.4 Lista sencilla 392
14.5 Pila 396
14.6 Ejercicios propuestos 400

Referencias bibliográficas 401

Índice de figuras

Figura 1. Concepto de excepción 18
Figura 2. Concepto de aserción 24
Figura 3. Concepto de recursividad 28
Figura 4. Comportamiento recursivo 31
Figura 5. Representación gráfica del método recursivo 32
Figura 6. Concepto de estructuras de datos 37
Figura 7. Concepto de estructuras de datos estáticas 38
Figura 8. Concepto de estructuras de datos dinámicas 39
Figura 9. Concepto de arreglos 42
Figura 10. Concepto de arreglo unidimensional vector 44
Figura 11. Arreglo v (10) de tipo entero 47
Figura 12. Arreglo v ( ) para búsqueda binaria 49
Figura 13. Resistencia eléctrica 94
Figura 14. Concepto de matriz 98
Figura 15. Clase String 122
Figura 16. Concepto de pila 178
Figura 17. Concepto de cola 198
Figura 18. Concepto de lista de doble enlace 214
Figura 19. Concepto de listas circulares sencillas 236
Figura 20. Concepto de árbol binario 250
Figura 21. Concepto de grafo 288
Figura 22. API de colecciones en Java 339
Figura 23. Concepto de colecciones 340
Figura 24. Concepto de colecciones 341

Prólogo

Las ciencias de la computación constituyen un área del conocimiento que integra una gran diversidad de temáticas relacionados con los sistemas y la informática. Para el caso, se toman aspectos tales como las estructuras de datos estáticas y dinámicas como medio para la organización y gestión de la información a partir de algoritmos que permiten la adición de datos, búsqueda, ordenamientos, actualización, borrado y recuperación de información almacenada en memoria, se abordan los fundamentos para la utilización y representación de grafos. Tópicos de vital importancia por su aplicabilidad en los procesos computacionales tales como organización de datos, inteligencia artificial, en la gestión de los directorios de archivos de varios niveles, como en el sistema operativo Unix, el cual trabaja con ellos por medio de una estructura en árbol de varios niveles, siendo de esta manera la organización óptima y la búsqueda más sencilla y rápida; así como también su aplicación en las bases de datos. Espacios académicos que son base y fundamentos esenciales en la formación de los futuros ingenieros de sistemas.

Las estructuras de datos estáticas son aquellas en las que el tamaño en memoria se define antes de correr el programa y este tamaño no puede actualizarse durante la ejecución del programa. En otras palabras, es una variable simple que hace referencia a un único valor a la vez; dentro de este grupo de datos, pueden ser valores numéricos, caracteres, cadenas, objetos, entre otros.

En las estructuras de datos dinámicas el tamaño y su forma es variable a lo largo de un programa, es decir, la memoria se reserva a tiempo de corrida del programa; este tipo de estructuras está conformada por nodos, los cuales tienen como mínimo dos campos: uno de información (clientes, pasajeros, cuenta bancaria, entre otros) y otro que contiene la referencia del siguiente nodo; los nodos se crean y destruyen en tiempo de ejecución. Esto permite dimensionar la estructura de datos de una forma precisa permitiendo la asignación de memoria en tiempo de ejecución según se va requiriendo.

Con el propósito de lograr lo anterior, esta obra se divide en catorce capítulos, cada uno distribuido de la siguiente manera: relación general de la temática a

desarrollar, una introducción al tema central donde se complementa con un mapa mental, el desarrollo de la temática respectiva incluyendo ejemplos, se recomiendan unas lecturas que posibilitan ampliar el tema, unas preguntas de revisión de conceptos y, finalmente, se propone una serie de ejercicios como apoyo complementario y fortalecimiento de los conceptos abordados.

En el capítulo 1, se tratan temas fundamentales para la validación de datos, como los son las excepciones como mecanismo de control de errores en tiempo de ejecución, siendo una forma de hacer que la aplicación continúe la ejecución si se produce un error, y las aserciones, que son una condición que se asume como verdadera y que el sistema se encarga de comprobarla.

En el capítulo 2, la recursividad, como técnica de programación que permite el llamado de métodos dentro de un mismo método, que además permite la división de problemas complejos, necesaria para facilitar ciertas operaciones en el desarrollo e implementación de algoritmos en algunas estructuras de datos. También presenta la distinción entre estructuras de datos estáticas y dinámicas.

En el capítulo 3, se estudian los arreglos unidimensionales o vectores; en Java estas estructuras de datos permiten almacenar un conjunto de datos de un mismo tipo. El tamaño del arreglo se declara en un primer momento y no puede cambiar luego durante la ejecución del programa.

En el capítulo 4, se trabaja con arreglos bidimensionales o matrices como estructuras de datos estáticas no lineales, útiles para cierto tipo de aplicaciones. En el capítulo 5, se estudia el tema relacionado con el manejo de cadenas en Java, incluyendo los métodos asociados a la clase String, StringTokenizer, StringReader, entre otras.

A partir del capítulo 6, se abordan las estructuras de datos dinámicas, las cuales corresponden a una colección de elementos llamados nodos, que son estructuras donde la dimensión puede crecer o decrecer durante la ejecución del programa. Este tipo de estructuras de datos pueden ser dinámicas lineales (como las listas enlazadas, las pilas, las colas, las listas circulares, las listas dobles) o estructuras de datos dinámicas no lineales (como los árboles y grafos). En este capítulo se da inicio con listas con referencias o enlaces sencillos.

El capítulo 7 hace alusión a las estructuras de datos dinámicas de tipo lista de acceso restringido, como son las pilas y las operaciones básicas para su implementación.

En el capítulo 8, se estudian otro tipo de listas de acceso restringido, como son las colas, tipo de lista que implementa la técnica de primeros en entrar, primeros en salir.

En el capítulo 9, se involucran los temas listas con enlaces dobles, pueden ser utilizadas cuando son necesarias varias operaciones de inserción o eliminación de elementos, facilitando su recorrido en ambas direcciones.

En el capítulo 10, se estudia la temática de listas circulares enlazadas, que son un tipo de listas donde el último nodo apunta al primero, lo cual la convierte en una lista sin fin.

En el capítulo 11, se estudia otro tipo de estructuras de datos dinámicas no lineales, como lo son los árboles, implementado los métodos para las operaciones que se pueden realizar, como la creación de un árbol, los recorridos que consiste en visitar cada uno de sus nodos del árbol una vez; los recorridos son en-orden, pre-orden y postorden.

En el capítulo 12, se continúa con el concepto de estructura de datos no lineales brindando una introducción a los grafos.

En el capítulo 13, se estudian las colecciones, que son una especie de arreglos de tamaño dinámico, conformadas por un conjunto de clases e interfaces del paquete java.util para gestionar colecciones de objetos.

En el capítulo 14 se aborda las estructuras de datos desde la genericidad.

A pesar de tratarse los principales elementos de las estructuras de datos y de emplearse buenas prácticas y elementos de la programación orientada a objetos en los ejemplos y en algunos proyectos y ejercicios propuestos, no es finalidad de este texto involucrarse con elementos de complejidad algorítmica y de análisis de algoritmos. El estudiante interesado en el tema, podría emplear otro lenguaje de programación de alto nivel diferente a Java.

Capítulo 1

Excepciones y aserciones

1.1 Temática a desarrollar

- Manejo de excepciones
- Tipos de excepciones
- Aserciones

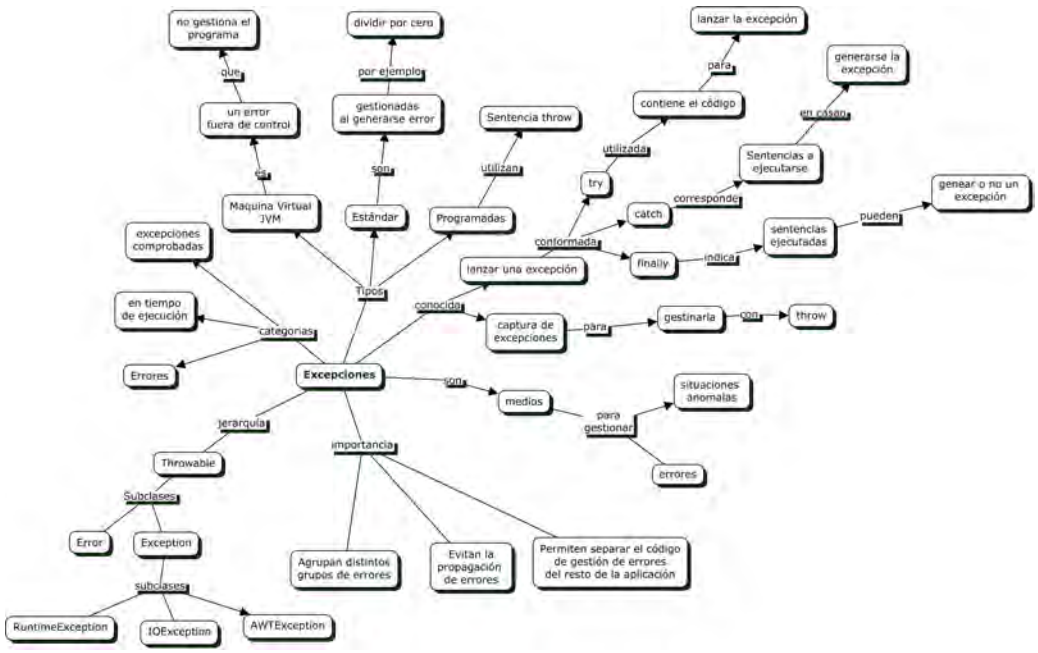
1.2 Introducción

Una excepción es un evento que ocurre durante la ejecución o corrida de un programa, que interrumpe el flujo normal de las sentencias, alterando la ejecución de la aplicación, donde el error se pude relanzar o capturar, permitiendo a los programas de Java que manejen los diferentes errores que puedan suceder.

Existen diversos tipos de errores que pueden generar excepciones, que van desde problemas de hardware, como daño de un disco duro, o los errores de programación, el tratar de acceder a un elemento en un arreglo fuera de su rango, el acceso a la información de un archivo que no existe o el nombre diferente.

Las excepciones son requeridas para la validación de datos en una aplicación de software. En la figura 1 se resume el concepto de excepciones.

Figura 1.
Concepto de excepción



1.3 Tipos de excepciones

La máquina virtual de Java (JMV) puede generar una excepción como producto de un error interno que está fuera de su control. Estas excepciones generalmente no pueden ser manejadas por el programa. Los dos tipos de excepciones principales son:

- **Runtime Exceptions:** excepciones generadas en el sistema de ejecución de Java. Por ejemplo, utilizar referencias en null, realizar una división por cero, acceder a una posición que no existe en un arreglo.

- **NonRuntime Exceptions:** excepciones producidas fuera del sistema de ejecución de Java. Por ejemplo, las excepciones que se producen por acceso a archivos (IOExceptions).

Se pueden construir una excepción utilizando la estructura throw. Sin tener en cuenta cómo se produjo la excepción, se maneja de la misma forma que las excepciones definidas en Java.

1.4 Sentencias try, catch, finally

try*:** corresponde a un grupo de sentencias que se ejecuta y lanza la excepción a la primera sentencia ***catch que gestione ese tipo de excepción.
***catch*:** sentencia que captura la excepción y realiza las acciones correspondientes.
finally*:** se ejecuta al final de ***try y después de ***catch***, además captura las excepciones no atrapadas por una sentencia ***catch*.**

Su sintaxis es:

```
try {
        // bloque de código en el que gestionamos la excepción
}
catch ( tipo_excepción1 objeto ) {
        // Gestión de tipo_excepción1
}
catch ( tipo_excepción2 objeto ) {
        // Gestión de tipo_excepción2
}
finally {
        // Gestión de otras excepciones
}
```

1.5 Implementación de las excepciones

Las excepciones se incorporan en Java con la capacidad de detectar y notificar errores. Al presentarse un error, se realizan las siguientes acciones:

- Se interrumpe la ejecución del código en curso.
- Se crea una excepción de tipo objeto que contiene información del problema. Esta acción es conocida como "*lanzar una excepción*". Existe un mecanismo estándar de gestión de error.
- Si hay un manejador de excepciones en el contexto actual, se le transfiere el control. En caso contrario, pasa la referencia de la excepción tipo objeto al contexto anterior en la pila de llamadas.
- Si no hay ningún manejador con capacidad de gestionar la excepción, el hilo que la generó es terminado.

1.6 La importancia de usar excepciones

Usar excepciones es de gran importancia en la medida de que:

- El código de tratamiento del error está separado del resto del programa. Esto aumenta la legibilidad y permite centrarse en cada tipo de código.
- Un mismo manejador puede gestionar las excepciones de varios ámbitos inferiores. Esto reduce la cantidad de código necesaria para gestionar los errores.

1.7 Excepciones comunes

Entre las excepciones más comunes con que cuenta un programador están:

IOException: se genera por falla de entrada o salida, tal como la inhabilidad de leer desde un archivo.
NullPoint erException: se hace referencia a un objeto NULL.
NumberFormatException: se digita un carácter en vez de un número, o por una conversión que no se pudo realizar de Strings a números.
ArithmeticException: errores aritméticos, por realizar una división por cero.
ArrayIndexOutOfBoundsException: se accede a una posición inexistente en el arreglo para almacenar información o el tipo de índice erróneo.
FileNotFoundException: se accede a un archivo inexistente.

Ejemplo:

1. Método que realiza la división de dos números; cuando el divisor es cero, se atrapa la excepción.

```
public String obtenerDivision(int  a,int  b) {
        int   c=0;
        String resultado="";
        try {
                c=a/b;
                resultado=""+c;
        }catch (ArithmeticException ex) {
                resultado="Error ha intentado dividir por cero";
        }
        return  resultado;
}
```

2. Aplicación que eleva un número al cuadrado en un rango entre 0 y 299 y en el caso de exceder este rango, lanza la excepción.

```
import java.util.Scanner;

public class ExcepcionNumero {
        private int numero;

        public ExcepcionNumero() {
                this.numero = 0;
        }


        public void leerNumero(){
                Scanner in = new Scanner(System.in);
                System.out.print ("Digite número a elevar al cuadrado menor a 300:
                ");
                numero=in.nextInt ();
                        try{
                                int numero = leerUnNumero();
                                longcuadrado=(long)Math.pow(numero, 2);
                                System.out.println("El cuadrado del número es: " +
                                cuadrado);
                        }catch( RangoNumericoExcedido e ) {
                                System.out.println("Rango excedido ( 0 ...300 )");
                        }catch( NumberFormatException e ){
                                System.out.println("Error en el número digitado.");
                        }

        }


        public int leerUnNumero() throws RangoNumericoExcedido
        {

                if ( numero< 300 ){
                        return numero;
                }
                else{
                        thrownew RangoNumericoExcedido(numero);
                }
        }

        public static void main(String[] args) {
                ExcepcionNumero unaExcepcion = new ExcepcionNumero( );
                unaExcepcion.leerNumero( );
```

```
        }

}


class RangoNumericoExcedido extends Exception
{
        public RangoNumericoExcedido(int numero)
        {
                super(" " + numero);
        }
}
```

3. Completar el siguiente programa que ingresa una cadena por teclado y la convierte a entero; para el caso que se ingrese un carácter o un símbolo, se captura la excepción.

```
/**
 * Método que recibe como parámetro una cadena y retorna la cadena
 * convertida en entero
 * @param a
 * @return
*/
        public int  convertirNumero(String a)
        {
                int n=-1;
                try{
                        n=Int eger.parseInt (a);
                        System.out.print ("Conversión satisfactoria ");

                }catch(NumberFormatException exc){
                        System.out.println("Error " +exc.toString());
                }
                return n;
        }
```

1.8 Ejercicios propuestos

1. Consultar en la API de Java:

   Clase Throwable
   Las subclases inmediatas:
   - Error
   - Exception

2. Consultar el proceso de creación de una excepción propia.

3. Construir la clase Cita para controlar la agenda de un médico. La clase tendrá los siguientes atributos y sus respectivas propiedades:
   - Día de la semana (ejemplo: lunes)
   - Hora (entero entre 0 y 23)
   - Minuto (entero entre 0 y 59)
   - Descripción de la cita (cadena)

Implemente un método constructor que reciba como parámetros todos los datos de la cita. Si el día, hora o minuto no son válidos, se debe lanzar una excepción.

1.9 Aserciones

Es un mecanismo de control que permite realizar comprobaciones durante la ejecución de un programa. Estas comprobaciones son expresiones booleanas. Si devuelve falso, se genera una excepción AssertionError.

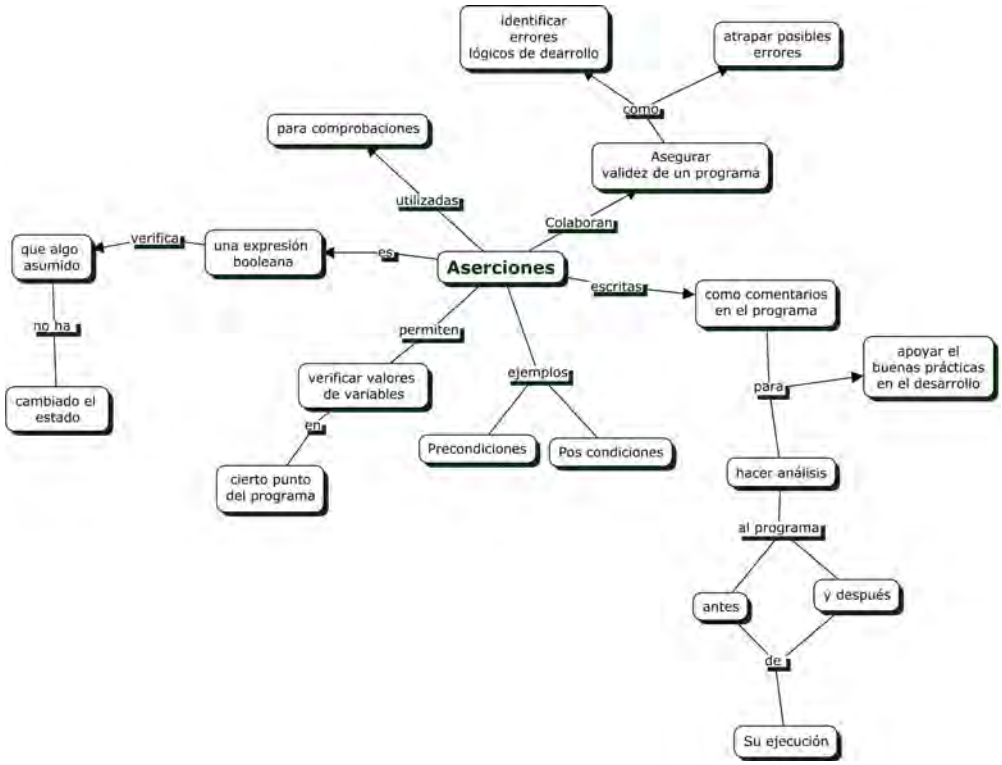
Las aserciones se utilizan durante la fase de desarrollo y depuración de una aplicación para verificar ciertas suposiciones asumidas por el programa, evitando la utilización innecesaria de instrucciones de impresión o de captura de excepciones.

Cuando una variable toma un valor erróneo, en definitiva, las consecuencias se pueden apreciar inmediatamente o puede que el programa arrastre el error durante un tiempo antes de que se aprecie un fallo de ejecución o entregue un resultado final equivocado. Las aserciones se utilizan para la validación de datos en una aplicación de software. En la figura 2 se detalla el concepto de aserción.

La sintaxis de una aserción es:

- assert(condición) donde condición es una expresión cuyo resultado debe ser de tipo boolean (assert expresion_boolean). La condición siempre se espera que sea verdadera (true); si es así, el programa continúa ejecutándose normalmente, pero si la condición es falsa, el programa se interrumpirá lanzando un AssertionError que no deberá ser capturado.

Figura 2.
Concepto de aserción



Ejemplo:

El método evaluar que recibe con parámetro entero de visibilidad privada evalúa el dato, el cual será siempre un número positivo; se utiliza una aserción para depurar esta condición.

```
private void evaluar (int  numero)
{
assert(numero>0);
}
```

- Existe una segunda forma de utilizar aserciones que permite enviar información adicional cuando se produce el error de aserción assert(condicion):expresión; donde expresión es cualquier expresión que devuelve un valor.

```
private void evaluarNumero (int  numero) {
        assert(numero>0) : numero + " no es positivo";
}
```

Ejemplo:

Programa que implementa una aserción que evalúa si la edad ingresada es un valor negativo, en caso contrario, el programa se ejecutaría normalmente.

```
class EdadAssert {
        private int edad;

        public EdadAssert(int  edad) {
                        this.edad = edad;
        }

        private boolean  verificarEdad() {
                assert (edad> 0):"ingresa una edad mayor que cero";
                /* si edad es valida (p.e. edad>0) */
                if (edad>= 18) {
                        return true;

                }
                return  false;
        }

        public static void main (String[] args){
                int  edad = 20;
        //edad con los datos correctos.
        EdadAssert unaEdad = new EdadAssert(edad);
        if (unaEdad.verificarEdad()){
             System.out.println("Ya puede votar");
           }
        //Se crea una edad que harán saltar el aserto.
           edad = -17;
           EdadAssert otraEdad = new EdadAssert(edad);
        if (otraEdad.verificarEdad()){
             System.out.println("error");
           }
         }
}
```

1.10 Ejercicios propuestos

1. Consultar cómo se habilitan las aserciones en los entornos integrados de desarrollo Eclipse y Netbeans.

2. Construir un programa que, a partir de los tres segmentos de recta, determine si se puede formar un triángulo e implementar una aserción para el caso en que las medidas de los lados no permiten formar la figura.

Capítulo 2

Recursividad y estructuras de datos

2.1 Temáticas a desarrollar

- Recursividad
- Estructuras de datos estáticas
- Estructuras de datos dinámicas

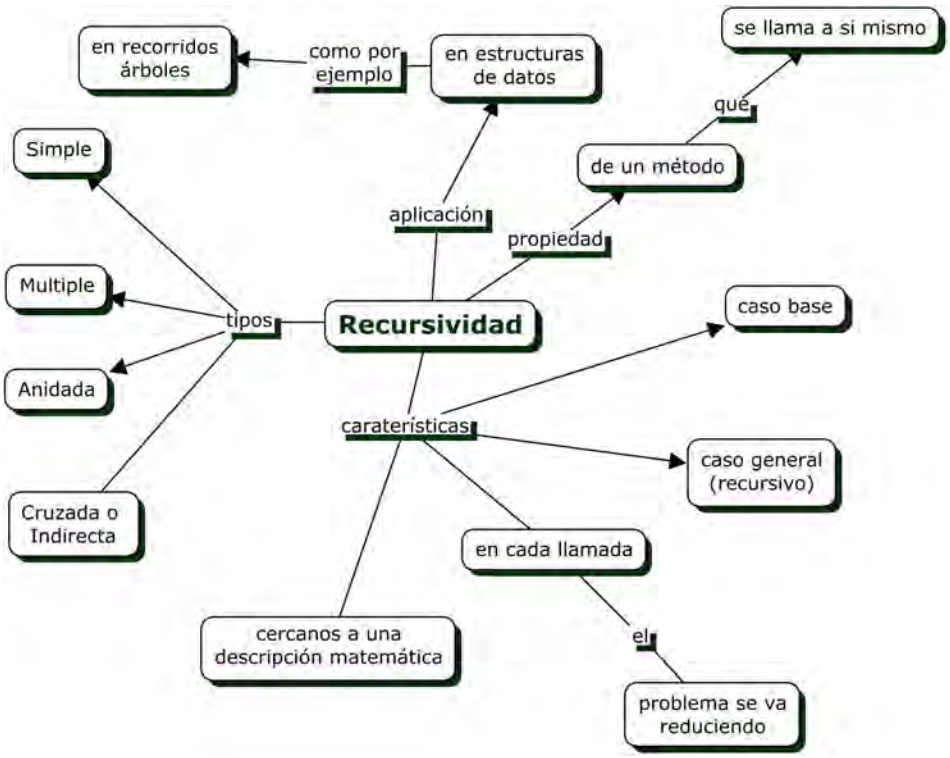
2.2 Introducción

La recursividad en programación consiste en realizar el llamado de un método dentro del mimo método. La recursividad es una técnica de programación que puede utilizarse en cambio de la iteración para resolver determinados tipos de problemas, que puede expresarse en términos de la resolución de un problema de igual naturaleza aunque de menor complejidad; en otras palabras, un problema complejo se divide en otros problemas más sencillos del mismo tipo denominado *caso base*, haciendo posible que se tenga que recurrir al *caso base* definido.

Los métodos recursivos o recurrentes se pueden clasificar en recursividad directa o recursividad indirecta, donde la recursividad directa se presenta cuando el método se manda llamar a sí mismo dentro de su propio cuerpo de instrucciones y la recursividad indirecta se muestra cuando un primer método llama a otro y dentro del segundo se manda llamar al primero.

La aplicación de la recursividad es fundamental en la implementación de algunos métodos u operaciones con las estructuras de datos. En la figura 3 se detalla el concepto de recursividad.

Figura 3.
Concepto de recursividad



2.3 Características de la recursividad

- La recursividad es una herramienta poderosa de la programación, en la cual un método se invoca a sí mismo, concepto fundamental en matemáticas y en computación.
- Es una alternativa diferente para implementar estructuras de repetición (ciclos). Los módulos se hacen llamadas recursivas.
- Se puede usar en toda situación en la cual la solución pueda ser expresada como una secuencia de movimientos, pasos o transformaciones administradas por un conjunto de reglas no ambiguas.
- Cuando un método se llama a sí mismo dentro de su definición (dentro del método), se dice que es directamente recursivo.
- Si un método x contiene una invocación a otro método, y que a su vez contiene una invocación al método x, se dice que el método x es indirectamente recursivo.

- Un método recursivo debe incluir una o varias instrucciones selectivas para establecer la condición o condiciones de salida.
- Al escribir métodos recursivos, se debe tener una sentencia if para obligar al método a volver sin que la llamada recursiva sea ejecutada.

Los métodos recursivos están estructurados por:

- **Caso base:** es el caso más simple de un método recursivo, que devuelve un resultado; el caso base se puede considerar como una salida no recursiva.

- **Caso general:** relaciona el resultado del algoritmo con resultados de casos más simples. Dado que cada caso de problema aparenta o se ve similar al problema original, la función o método llama una copia nueva de sí misma, para que empiece a trabajar sobre el problema más pequeño y esto se conoce como una llamada recursiva y también se llama el paso de recursión.

2.4 Tipos de recursividad

Recursividad simple: aquella en cuya definición solo aparece una llamada recursiva. Se puede transformar con facilidad en algoritmos iterativos. Como por ejemplo, el factorial para cualquier entero positivo N, que se expresa N! y es el producto de todos los enteros menores al número, donde el factorial de cada número incluye el factorial de todos los números anteriores a este valor, por ejemplo, 5! es 4! * 5; 3! es 2! * 3; como se observa en la siguiente tabla, la definición recursiva para el factorial:
n! = n * (n-1)! , donde n >= 1

Definición recursiva	Factorial de un número
0! = 1	0! = 1
1! = 1	1! = 0! x 1 = 1
2! = 1! x 2	2! = 1! x 2 = 2
3! = 2! x 3	3! = 1 x 2 x 3 = 6
4! = 3! x 4	4! = 1 x 2 x 3 x 4 = 24
5! = 4! x 5	5! = 1 x 2 x 3 x 4 x 5 = 120
6! = 5! x 6	6! = 1 x 2 x 3 x 4 x 5 x 6 = 720
7! = 6! x 7	7!= 1 x 2 x 3 x 4 x 5 x 6 x 7= 5.040

El siguiente código presenta la clase Factorial en la cual se implementa el método obtenerFactorial, el cual recibe como parámetro un número entero y retorna el valor correspondiente al factorial de dicho número.

```
public class Factorial {

....

        public long  obtenerFactorial(int  n)          {
                //condición para detenerse
        If(n==0) {
        return  1;
           } else {
                // Método recursivo que se llama a sí mismo
                long fact=n*obtenerFactorial(n-1);
        return  fact;
          }
        }
```

En general, la definición recursiva para el factorial de un entero N es:

$$N! = \begin{cases} 1, & \text{si } n = 0 \text{ (base)} \\ n * (n-1)!, & \text{si } n > 0 \text{ (recursión)} \end{cases}$$

La base no es recursiva y es el punto tanto de partida como de terminación de la definición.

Tener en cuenta que:

- Se debe asegurar que existe una condición (por ejemplo n==0) de salida, que no genera llamadas recursivas. Es importante determinar un caso base, es decir, un punto en el cual existe una condición por la cual no se requiera volver a llamar al mismo método.

```
if(n==0) {
return  1;
        }
```

- Asegurar que se cubren todos los posibles casos entre el valor inicial (por ejemplo, n = 0) y los no iniciales (ejecución de sentencias).

```
else {
    long fact=n*obtenerFactorial(n-1);
}
```

- Cada llamada, en el caso no inicial, conduce a problemas cada vez más pequeños que terminarán en el valor inicial.

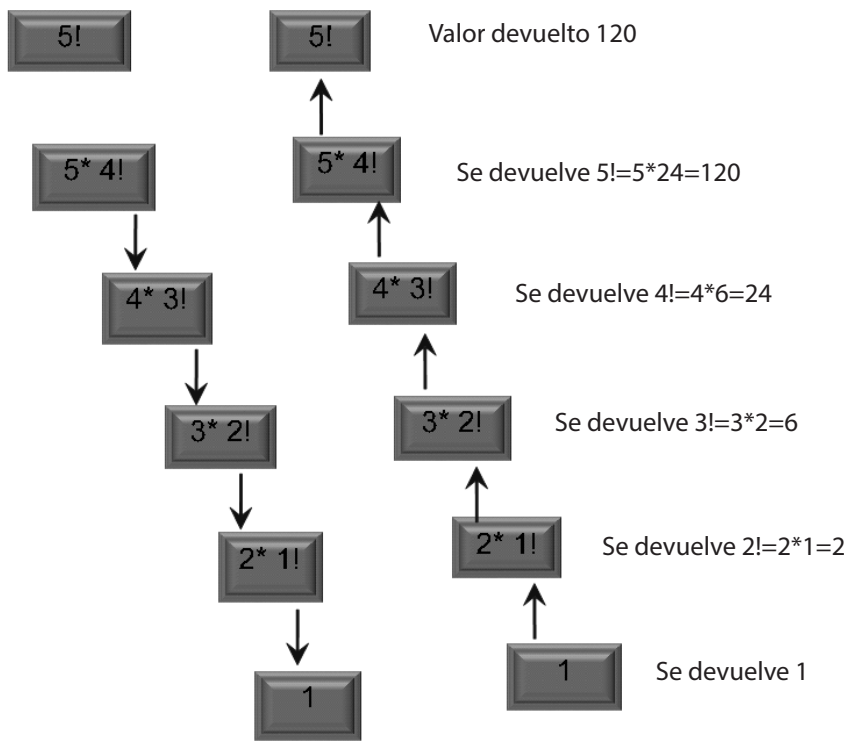
```
obtenerFactorial(n-1);
```

En la siguiente tabla se explica el comportamiento recursivo para calcular el factorial de 5.

Factorial	Llamado de la recursividad	Retorno del invoque recursivo
5! = 5 * 4!	Fact = 5 * obtenerFactorial(4)	1 * obtenerFactorial(0) return 1
4! = 4 * 3!	4 * obtenerFactorial(3)	1 * obtenerFactorial(1) return 1
3!= 3 * 2!	3 * obtenerFactorial(2)	1 * obtenerFactorial(2) return 2
2! = 2 * 1!	2 * obtenerFactorial(1)	2 * obtenerFactorial(3) return 6
1! = 1 * 0!	1 * obtenerFactorial(0)	6 * obtenerFactorial(4) return 24
0! = 1		24 * obtenerFactorial(5) return 120

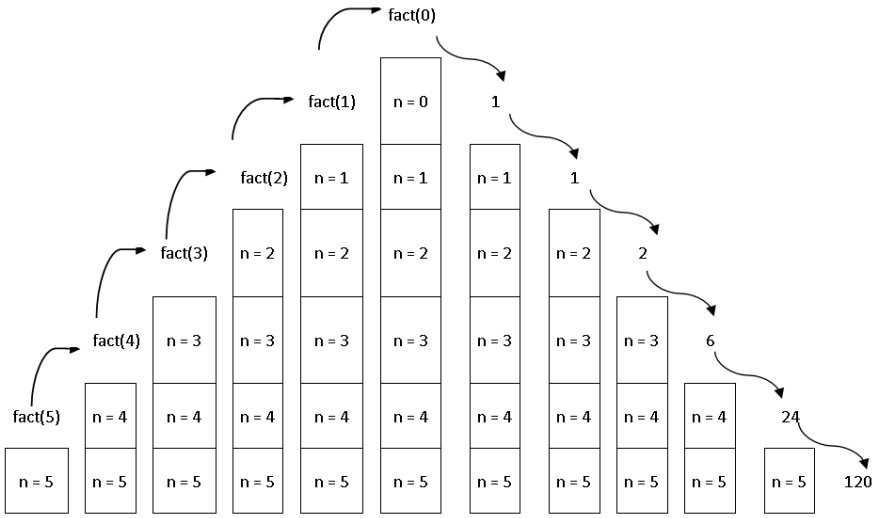
La figura 4 presenta el comportamiento del método recursivo.

Figura 4.
Comportamiento recursivo



Otra manera de representar gráficamente el comportamiento del método recursivo para hallar el factorial de 5 se muestra en la figura 5.

Figura 5.
Representación gráfica del método recursivo



El factorial de un número también se puede realizar utilizando instrucciones repetitivas como, por ejemplo, con un ciclowhile, como se presenta en el método getFactorial.

```
public int  getFactorial(int x){
        while (x!=0){
                return x * getFactorial(x-1);
        }
        return  1;
}
```

Recursividad múltiple: se da cuando hay más de una llamada a sí misma dentro del cuerpo del método, resultando más difícil de hacer de forma iterativa, como, por ejemplo, la serie de Fibonacci (0, 1, 1, 2, 3, 5, 8, 13,…).

```
public long  obtenerNumero(long n){
        System.out.print  (" "+n);
        if ( n == 0 ){
                return  0;
        }else if  (n == 1){
                return  1;
        }else{
        return  obtenerNumero(n - 1) + obtenerNumero(n - 2);
        }
}
```

Recursividad anidada: en algunos de los argumentos de la llamada recursiva hay una nueva llamada a sí misma, por ejemplo, la función Ackermann. En teoría de la computación, función de Ackermann es una función matemática recursiva encontrada en 1926 por Wilhelm Ackermann. Esta función toma dos números naturales como argumentos y devuelve un único número natural. Como norma general se define así:

$$A(m,n) = \begin{cases} n + 1 & \text{si } m = 0; \\ A(m - 1, 1), & \text{si } m > 0 \text{ y } n = 0; \\ A(m - 1,(m,n - 1)), & \text{si } m > o \text{ y } n > 0; \end{cases}$$

Recursividad cruzada o indirecta: son algoritmos donde un método realiza una llamada a sí mismo de forma indirecta a través de otros métodos. Es decir, es aquel en la que un método es llamado por otro y este a su vez llama al método que lo llamó, tal como se muestra a continuación:

```
public class Recursividad {
   …….
        int  par(int nump) {
                if (nump==0)
                        return (1);
                return ( impar(nump-1));
        }
        int  impar (int  numi) {
                if(numi==0)
                        return (0);
                return ( par(numi-1));
        }

}
```

Ejemplos:

1. Aplicación que implementa un método recursivo que permite hacer la división por restas sucesivas de dos números enteros.

```
        import java.util.Scanner;

public class Division {
        public int  obtenerDivision (int x, int y){
                if(y>x)
```

```
                return 0;
        else
                return obtenerDivision(x-y, y)+1;
    }

    public static void main(String[] args) {
        int x = 0;
        int y = 0;
        Scanner lea = new Scanner(System.in);
        Division unaDivision = new Division();
        System.out.print ("Digite Dividendo: ");
        x = lea.nextInt ();

        System.out.print ("Digite Divisor: ");
        y = lea.nextInt ();
        System.out.println(x+" / "+y+" = "+unaDivision.obtenerDivision(x,
        y));

    }

}
```

2. Programa que calcula la potencia de dos números enteros, mediante la implementación del método recursivo obtenerPotencia, que recibe como parámetro los números correspondientes a la base y el exponente.

   Revisar tamaño de la viñeta 2 respecto a la 1

```
import java.util.Scanner;

public class Potencia {

        public int obtenerPotencia(int base, int expo){
            if ( expo == 0 )
                return 1;
            else
                return (base * obtenerPotencia(base, expo - 1));
        }

        public static void main(String[] args) {
            int x = 0;
            int y = 0;
            Scanner lea = new Scanner(System.in);
            Potencia unaPotencia = new Potencia();
```

```
                    System.out.print ("Digite base: ");
                    x = lea.nextInt ();

                    System.out.print ("Digite exponente: ");
                    y = lea.nextInt ();
                    System.out.println(x+" ^ "+y+" =
                    "+unaPotencia.obtenerPotencia(x, y));
            }

}
```

3. Aplicación que permite hallar el máximo común divisor de dos números, el cual corresponde al valor más grande posible que divide a esos números.

```
import java.util.Scanner;

public class MaximoComunDivisor {
        public int  obtenerMCD(int a, int b) {
                int resultado;
                System.out.println("inicio de mcd("+a+","+b+")");
                if (b==0)
                resultado= a;
                else
                resultado= obtenerMCD(b, a% b);
                return resultado;
        }



        /**
         * Método que lee dos números y los envía como parámetro
         * para obtener el mcd
         */

        public void leerNumeros(){
                int a = 0;
                int b= 0;
                Scanner in = new Scanner(System.in);
                System.out.print ("Digite el primer número: ");
                a=in.nextInt ();
                System.out.print ("Digite el segundo número: ");
                b=in.nextInt ();
                System.out.println("El MCD de "+a+" y "+b+ " es:
                "+obtenerMCD(a,b));
```

```
        }
        public static void main(String[] args) {
                MaximoComunDivisor app=new  MaximoComunDivisor();
                app.leerNumeros();

        }
}
```

2.5 Ejercicios propuestos

1. Construir un método recursivo que permita calcular la sumatoria para los n primeros números cuadrados, donde se reciba como parámetro un número entero positivo, menor o igual a 50.

Entrada	Salida
3	14
5	55

   - Si la entrada es 3, la salida será 14, porque la sumatoria de los tres primeros números al cuadrado son (1 + 4 + 9 = 14).
   - Determinar el número de iteraciones que realice el método anterior antes de finalizar su proceso, teniendo en cuenta el tamaño de la entrada.

2. Se requiere construir una aplicación conformada por dos clases, la clase PruebaFactorial, donde se realiza la lectura del valor, se crea un objeto de la clase Factorial y se invoca el método getFactorial presentado en el punto anterior.

3. Implementar un método que defina la función de Ackermann, utilizando recursividad anidada.

2.6 Estructuras de datos

Se trata de un conjunto de variables de un determinado tipo, agrupadas y organizadas de alguna manera para representar un comportamiento. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. En algunos casos, la dificultad para resolver un

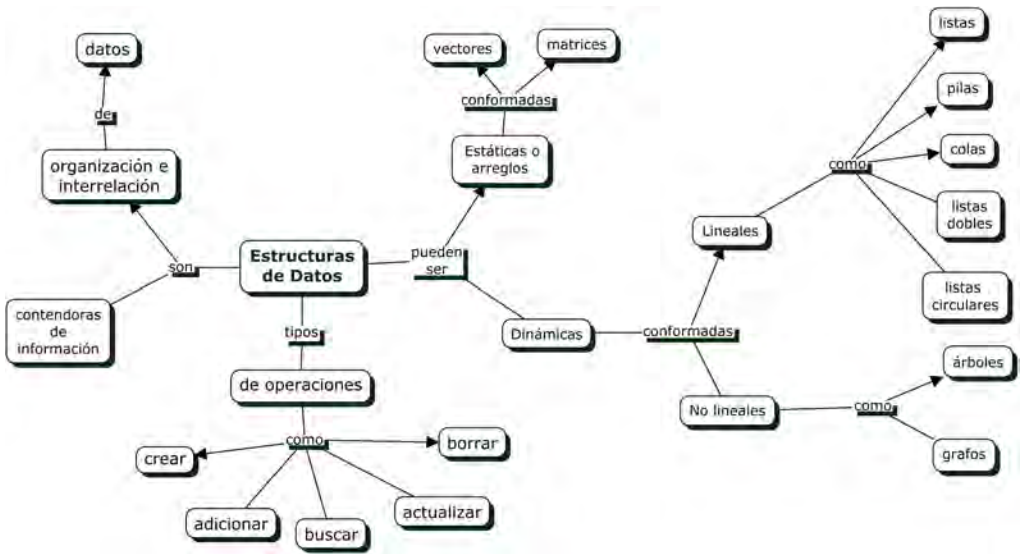
problema radica en escoger la estructura de datos adecuada. Y, en general, la elección del algoritmo y de las estructuras de datos que manipulará estará muy relacionada.

Según su comportamiento durante la ejecución de un programa y en general, se distinguen dos tipos de estructuras de datos:

- **Estáticas:** donde su tamaño en memoria es fijo. Por ejemplo: los arreglos.
- **Dinámicas:** su tamaño en memoria es variable. Por ejemplo: las listas enlazadas por medio de referencias, archivos, etc.

Las estructuras de datos que se abordan en los capítulos siguientes son los arreglos, las pilas y las colas, los árboles y algunas variantes de estas estructuras. La figura 6 muestra un concepto amplio de lo que son las estructuras de datos en general.

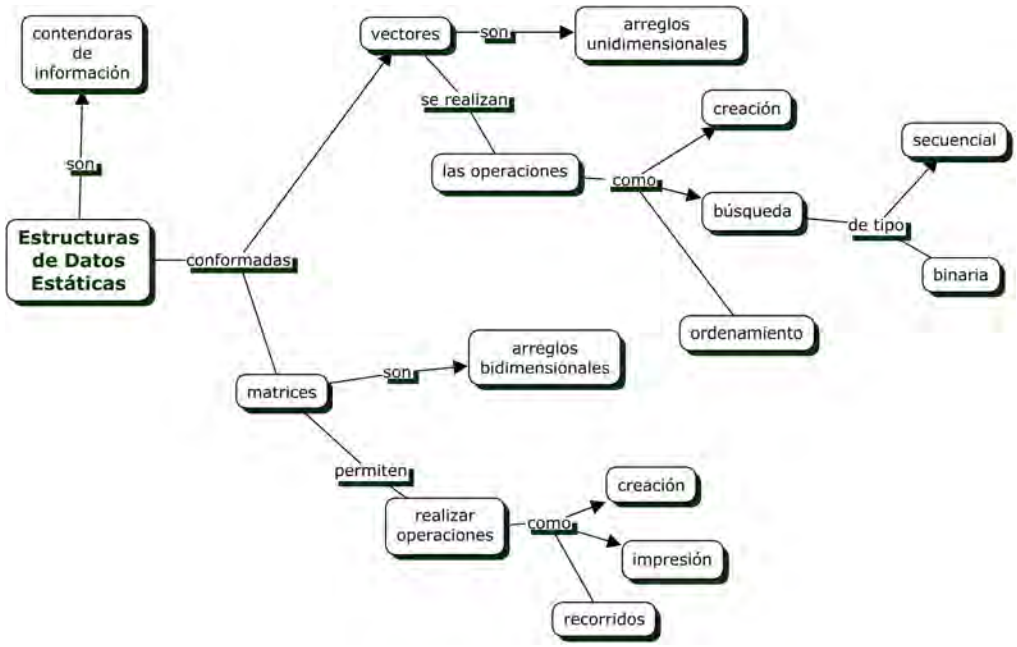
Figura 6.
Concepto de estructuras de datos



2.6.1 Estructuras de datos estáticas

Las estructuras de datos estáticas conformadas por vectores y matrices agrupan a un conjunto de datos de un tamaño fijo que se encuentran consecutivos en la memoria principal del computador y en la que es posible el acceso al elemento que se requiere simplemente con indicar su posición o ubicación. En la figura

Figura 7.
Concepto de estructuras de datos estáticas



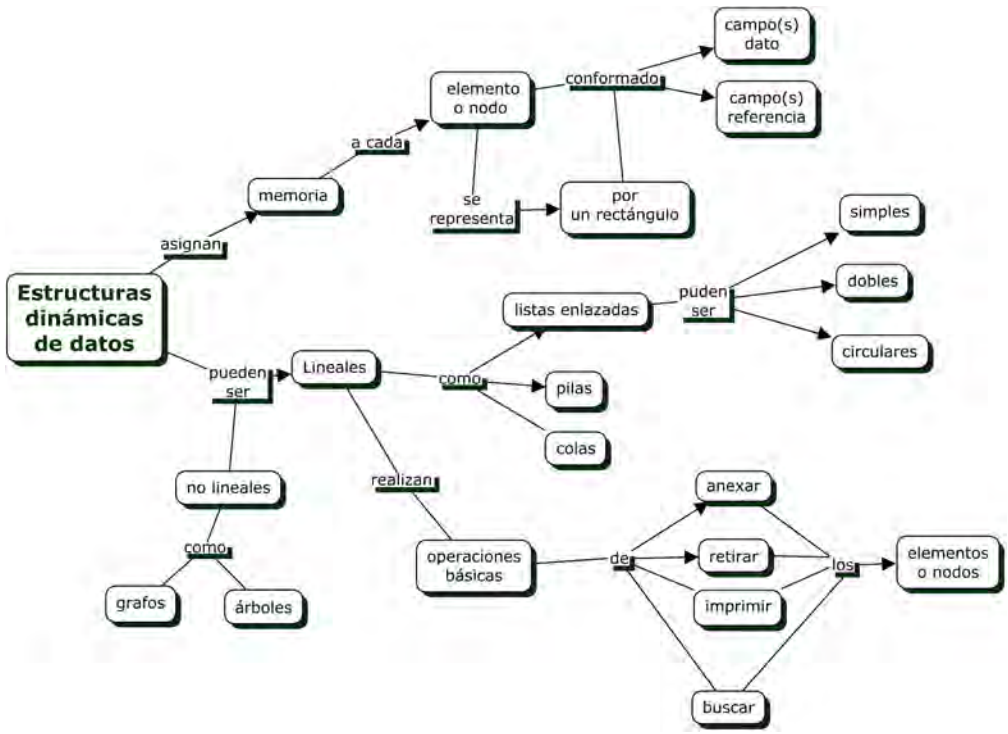
Este tipo de estructura de datos se estudia en los capítulos 3 y 4.

2.6.2 Estructuras de datos dinámicas

Las estructuras dinámicas son aquellas cuyo tamaño (longitud o número de elementos) varía en tiempo de ejecución del programa, como, por ejemplo, las listas simplemente enlazadas; al contrario de un arreglo, que tiene un tamaño fijo para almacenar sus elementos.

La figura 8 amplía el concepto de estructuras dinámicas de datos.

Figura 8.
Concepto de estructuras de datos dinámicas



Estas estructuras se abordan desde diversos puntos de vista: diseño de estructuras en respuesta a necesidades específicas, encapsulamiento de los tipos de datos usándolos con base en su especificación. En el capítulo 6 se empiezan a estudiar las estructuras de datos dinámicas relacionadas con enlace sencillo, al igual que en los capítulos 7 y 8, y en los capítulos 9 y 10 se continúa con las estructuras lineales con doble enlace.

Capítulo 3

Arreglos unidimensionales o vectores

3.1 Temática a desarrollar

- Vectores

3.2 Introducción

Los arreglos unidimensionales o vectores forman parte de las estructuras de datos estáticas, donde se almacenan uno o más datos de un mismo tipo y en una misma dimensión y es posible acceder a cada elemento de la estructura solamente con indicar su posición o ubicación donde este se encuentra.

Entre las principales ventajas de estas estructuras de datos están:

- Posibilita el uso de menor cantidad de nombres de variables, dado que con un solo nombre genérico se almacenan n datos, mientras que con el uso de datos simples se necesitarían n nombres de variables distintas.
- Menor tamaño del algoritmo cuando se utiliza gran cantidad de variables al mismo tiempo en memoria.
- A partir de un archivo desordenado, los arreglos permiten generar los mismos datos con un cierto orden.
- Se tiene todo el conjunto de datos en memoria sin necesidad de acceder al dispositivo.
- La facilidad de acceso a la información, ya sea a un elemento en particular o a un grupo con ciertas condiciones, y, por otro lado, los distintos sentidos para el recorrido del mismo, pueden ser desde el inicio al fin o viceversa.
- Los arreglos permiten el uso de parámetros en la definición del tamaño, con lo cual se logra mayor flexibilidad en los algoritmos.