Programación II

Apuntes de cátedra

Versión 1.3

Cuadrado Estrebou, María Fernanda
Trutner, Guillermo Hernán

Facultad de Ingeniería y Ciencias Exactas
Universidad Argentina de la Empresa

UADE

El objetivo de este apunte es el de ser una guía en el dictado de la materia, junto a las explicaciones de los docentes en las clases. Se encuentra en su primera versión y es por eso que agradeceremos todas las observaciones, comentarios y correciones que nos puedan hacer llegar los alumnos para ir mejorándolo.

Resumen

El objetivo del curso de Programación II es introducir el concepto de *tipos de datos abstractos* (TDA), sus especificaciones, implementaciones y aplicaciones. Se utilizará el lenguaje de programación Java como lenguaje de soporte, remarcando que no es el objetivo del curso el lenguaje en sí mismo, sino sólo un medio para aplicar los contenidos. Comenzaremos a trabajar con estructuras de datos conocidas por los alumnos de cursos previos, como son las pilas y colas entre otras y continuaremos con otras estructuras más complejas como son los árboles y grafos.

Índice General

Unidad 1

Introducción a Lenguaje *Java*

En esta unidad se introducirán los conceptos del lenguaje *Java* que se van a necesitar a lo largo del curso. No es éste un curso para aprender a programar en este lenguaje sino que es un curso en que se lo va a utilizar como herramienta para ejercitar todos los conceptos vistos.
Uno de los aspectos importantes a tener en cuenta antes de comenzar, es que el lenguaje es sensible a mayúsculas y minúsculas, es decir se deben utilizar las variables y los métodos tal cual fueron definidos.

1.1 Tipos de datos

Los tipos de datos básicos en *Java* que se van a utilizar a lo largo del curso, son los siguientes:

- `boolean`: los valores posibles que pueden tomar las variables de este tipo son `true` ó `false` únicamente
- `int`: valores enteros que tienen un tamaño de 32 bits
- `short`: valores enteros que tienen un tamaño de 16 bits
- `char`: representa a un carácter, y tiene un tamaño de 16 bits
- `byte`: representa a 8 bits
- `float`: es un valor numérico con decimales con un tamaño de 32 bits
- `long`: es un valor numérico con decimales con un tamaño de 64 bits
- `double`: es un valor numérico con decimales con un tamaño de 64 bits

1.2 Variables

Otro aspecto a considerar es cómo se lleva a cabo la declaración de variables en *Java*, y las mismas pueden ser de la siguiente manera:

- declaración de una variable sola:

  ```
  <tipo> <nombre_variable> ;
  ```

  por ejemplo:

  ```
  int cantidad;
  ```

- declaración de más de una variable del mismo tipo:

  ```
  <tipo> <nombre_variable_1>,<nombre_variable_2>,<nombre_variable_3>;
  ```

  por ejemplo:

  ```
  int cantidad, contador, resultado;
  ```

- declaración y asignación de valor a una variable:

  ```
  <tipo> <nombre_variable> = <valor>;
  ```

  por ejemplo:

  ```
  int contador = 0;
  ```

  o para el caso de variables del tipo char se debe usar para el valor entre ‘’:

  ```
  char c =‘f’;
  ```

Como se vió en algunos de los ejemplos anteriores para la asignación de valores a una variable se utiliza el operador `=` (por ejemplo: `variable = 5`).
El *alcance* de la variable está dado por el bloque en el que se declara. Una variable es visible dentro del bloque en el que fue declarada y dentro de todos los sub-bloques que se definan dentro de éste.

1.3 Operadores

1.3.1 Aritméticos

Las operaciones aritméticas básicas sobre números son las siguientes:

- Para indicar que un número es negativo se usa el símbolo `-` (por ejemplo `-5`). Este mismo símbolo es utilizado también para la operación de resta (por ejemplo `x = x - 8;`).
- `+`: operador de suma.
- `*`: operador de multiplicación.

- `/`: operador de división.
- `%`: operador de resto ó módulo, es decir, devuelve el resto de una división (por ejemplo `x % 2;` el resultado de esta operación es 1 si el número es impar y 0 si el número par).
- `++`: operador de incremento en uno (por ejemplo si tenemos `int x = 2; x++;`, entonces el valor que toma `x` es 3).
- `--`: operador de decremento en uno (por ejemplo si tenemos `int x = 2; x--;`, entonces el valor que toma `x` es 1).

1.3.2 Relacionales

Las operaciones de comparación básicas sobre números son las siguientes:

- `<`: operador menor, es decir, si tenemos `x < z`, la comparación devolverá `true` si `x` tiene un valor menor que el de `z`, y `false` en caso contrario.
- `>`: operador mayor, es decir, si tenemos `x > z`, la comparación devolverá `true` si `x` tiene un valor mayor que el de `z,` y `false` en caso contrario.
- `<=`: operador menor o igual, es decir, si tenemos `x <= z`, la comparación devolverá `true` si `x` tiene un valor menor o igual que el de `z,` y `false` en caso contrario.
- `>=`: operador mayor o igual, es decir, si tenemos `x >= z`, la comparación devolverá `true` si `x` tiene un valor mayor o igual que el de `z`, y `false` en caso contrario.
- `==`[1]: operador de igualdad, es decir, si tenemos `x == z`, la comparación devolverá `true` si `x` tiene un valor igual que el de `z`, y `false` en caso contrario.
- `!=`: operador distinto, es decir, si tenemos `x != z`, la comparación devolverá `true` si `x` tiene un valor distinto que el de `z,` y `false` en caso contrario.

1.3.3 Lógicos

Además, de las operaciones sobre números están las operaciones sobre valores lógicos:

- `!`: operador de negación, es decir, si tenemos `!valor`[2] , la comparación devolverá `true` si el valor que tiene `valor` es falso, y `false` en caso contrario.

[1] Tener especial cuidado de no omitir un signo `=,` ya que el lenguaje lo tomaría como una asignación y no daría error pero cambiaría el significado o comportamiento de la operación

[2] valor debe ser una expresión de tipo booleano

- `||`: operador O lógico.
- `&&`: operador Y lógico.

1.4 Estructuras de control

Dentro de las instrucciones básicas veremos la notación para secuencia, el condicional, condicional múltiple y algunos operadores de ciclos.

1.4.1 Bloques

En *Java*, los bloques de sentencias se delimitan con llaves ({ y }). Se comienza un bloque con una llave de apertura y se finaliza con una llave de cierre. En muchos de los casos, si el bloque contiene una única sentencia, se pueden omitir las llaves, aunque para evitar errores y malos entendidos, se recomienda utilizarlas siempre. La identación se utilizará a modo simbólico para facilitar la lectura del código, sin embargo el compilador lo ignora totalmente.

1.4.2 Secuencia

Para la separación de secuencias de instrucciones se utiliza `;`.

1.4.3 Condicional Simple

El condicional simple se utiliza para validar una condición, y según sea verdadera o falsa se ejecuta un bloque de sentencias.

```
if (<condición_boleana>){
    instrucción 1;
    instrucción 2;
    instrucción 3;
    ...
} else {
    instrucción 4;
    instrucción 5;
    instrucción 6;
    ...
}
```

El `else` en el condicional es opcional. En caso de no utilizar las llaves para armar un bloque de instrucciones, la primera instrucción será la que forme parte del `if` ó `else` y las siguientes quedan fuera del condicional.

1.4.4 Condicional múltiple

Para las evaluaciones de diferentes condiciones para un mismo valor se puede utilizar el `switch`. La condición de evaluación en un `switch` tiene que ser un valor entero (o una expresión cuya evaluación da como resultado un valor entero).

```
switch (<expresión_entera>){
    case valor1:
        instrucción 1;
        instrucción 2;
        ...
        break;
    case valor2:
        instrucción 3;
        instrucción 4;
        ...
        break;
    case valor3:
        instrucción 5;
        instrucción 6;
        ...
        break;
    default :
        instrucción 7;
        instrucción 8;
        ...
}
```

En cada `case` es aconsejable colocar un `break`, debido a que si no se coloca un `break` el lenguaje sigue evaluando las condiciones de los siguientes `case`, y en caso de que otro sea verdadero también lo ejecuta. Además, si la sentencia `switch` tiene el `default` y no se coloca un `break` en cada `case`, luego de ejecutar el case correspondiente siempre va a terminar ejecutando lo que esta definido dentro del `default`.

1.4.5 Ciclos

Para la ejecución de ciclos vamos a ver tres instrucciones, `while`, `for` y `do..while`. Las dos primeras se pueden ejecutar como mínimo cero veces, a diferencia del `do..while` que se ejecuta como mínimo una vez, debido a que primero hace la ejecución y luego evalúa la condición.

```
while (<condición_boleana>){
    expresión 1;
    expresión 2;
    ...
}
```

Si luego de la condición del `while` no se colocan las llaves, el lenguaje va a considerar como expresión de ejecución solamente la primera, y no las subsiguientes (este mismo comportamiento es válido para la sentencia `for` que describiremos a continuación).
Otra opción para un ciclo es el `for`:

```
for (<inicialización_variable> ; <condición_boleana> ; <
   variación_variable> ){
    expresión 1;
    expresión 2;
    ...
}
```

La variable que se inicializa y utiliza dentro del `for` puede ser declarada previo al `for` o dentro del mismo(como en el ejemplo que sigue). De acuerdo a dónde se declare será el alcance que tenga, es decir, si se declara en el mismo `for` sólo se podrá utilizar en las instrucciones dentro del ciclo del `for`. La variación de la variable puede ser un incremento o un decremento. Un ejemplo de uso del `for` podría ser para sumar los números del 1 al 10, con lo cual se haría:

```
int suma = 0;
for (int i = 1; i <= 10 ; i++){
    suma = suma + i;
}
```

En este caso tenemos declarada la variable `i` del `for` en el mismo, ya que no se requiere fuera del él.
Ahora supongamos que estamos buscando un valor en un arreglo de enteros de 20 elementos y queremos conocer no sólo si existe el valor, sino en qué posición del mismo, en cuyo caso haríamos lo siguiente:

```
int buscarValor(int[] a, int valor){
    boolean encontrado = false;
    int i;
    for (i=0; (i < 20 && !encontrado); i++){
        if (a[i]==valor){
            encontrado = true;
        }
    }
    if (encontrado){
        return i-1;
    }
    else{
        // Se devuelve un valor fuera del arreglo para indicar
           que no se encontró
        return -1 ;
    }
}
```

Por último, nos queda presentar el `do..while`:

```
do {
    instrucción 1;
    instrucción 2:
    ...
} while (<condición_boleana>)
```

El `do..while`, a diferencia del las otros dos sentencias de ciclos, lo que hace primero es ejecutar las instrucciones y luego evalúa la condición, con lo cual, al menos una vez se ejecutan todas las instrucciones independientemente del valor de la condición.

1.5 Arreglos

Luego de haber descripto las diferentes sentencias de ciclos vamos a ver como se declaran y usan los arreglos en *Java*.

Declaración
La forma de declarar un arreglo es la siguiente:
`<tipo_datos_arreglos>[] <nombre_arreglo>;`
por ejemplo, si queremos declarar un arreglo de enteros haríamos lo siguiente:
`int[] valores;.`

Dimensionamiento
Los arreglos, por tratarse de una estructura de tamaño fijo, antes de poder utilizarlos se le debe asignar la cantidad de elementos que va a contener, es decir, dimensionarlo:
`<nombre_arreglo> = new <tipo_datos_arreglos>[<dimension>]`
La palabra reservada `new`, se utilizará en el lenguaje, cada vez que se quiera reservar memoria para una variable que no sea simple. Para los enteros, reales y caracteres, como se ha visto, no es necesario hacer la reserva de memoria, ya que la declaración de la variable lleva inmersa la reserva de memoria.
Por ejemplo, si al arreglo `valores` utilizado para ejemplificar en la parte de declaración, le queremos decir que va a contener 100 elementos, entonces lo que haríamos es:
`valores = new int[100];`
Otra forma de dimensionar un arreglo, y a su vez inicializarlo con valores es la siguiente:
`int[] valores = {5,8,15,20};`
en este caso el arreglo toma una dimensión 4, con los valores 5, 8, 15 y 20 en las posiciones 0, 1, 2 y 3 respectivamente.

Utilización

Debemos tener en cuenta que los arreglos en *Java* tienen su primer elemento en la posición cero, es decir, si declaramos un arreglo de dimensión 100, lo vamos a poder recorrer desde 0 a 99. A continuación se ve un ejemplo de recorrido del arreglo valores para inicializarlo con valores:

```
for (int i = 0; i < 100;i++){
    Valores[i] = i + 2;
}
```

1.6 Paquetes y salida

El lenguaje provee conjuntos de librerías que pueden ser utilizadas dentro de nuestros programas. En *Java* las librerías se llaman *paquetes*, y se agrupan en sub-paquetes. Por ejemplo hay un paquete `net` y dentro del mismo hay un paquete `URL`. Para hacer referencia entonces a la URL, se utilizará la *notación de punto*, por ejemplo `util.URL` y si queremos utilizar una función de la librería, hay que invocar un método dentro de la librería, por ejemplo para recuperar el contenido de una URL se podría hacer así: `util.URL.getContents()`
Un paquete particular que vamos a utilizar es el que sirve para imprimir por pantalla. Es el paquete `System.out` y el método se llama `println`, entonces para imprimir algo, estos son ejemplos:

```
int n = 3;
char c = 'f';
System.out.println(n);
System.out.println(c);
System.out.println("Hola!");
```

1.7 Métodos

Por último, vamos a ver dos conceptos particulares de los lenguajes orientados a objetos como es el caso de *Java*, que son los **métodos** y **clases**. El concepto de métodos y clases que daremos es básico y no tiene por objetivo dar un conocimiento del paradigma de Orientación a Objetos, sino dar las herramientas básicas para poder trabajar a lo largo del curso.
En *Java* todas las operaciones se llevan a cabo a través de métodos (se puede hacer una similitud a funciones y procedimientos de los lenguajes estructurados) los cuales se declaran de la siguiente manera:

```
    <tipo_retorno_método><nombre_metodo>(<tipo_parametro><
       parametro1> ,...,<tipo_parametro> <parámetro_n>){
        instruccion1;
```

```
        instruccion2;
        ...
    }
```

En el caso de que el método no devuelva valor se lo declara del tipo **void**. Por otra parte, el método puede no contener parámetros, en cuyo caso es igualmente obligatorio colocar los paréntesis sin ningún parámetro, es decir, se colocaría:

```
<tipo_retorno_método><nombre_metodo>(){
    instruccion1;
    instruccion2;
    ...
}
```

A continuación daremos dos ejemplos de métodos, el primero devolverá un valor que es la suma de dos variables que se dan como parámetro, y el segundo será un método que no devuelve valor sino que imprime por pantalla los valores de un arreglo dado.

Ejemplo 1: Suma de dos valores dados

```
int sumar(int x, int y){
    int sum = 0;
    sum = x+y;
    return sum;
}
```

Esto se puede escribir de la siguiente manera que da el mismo resultado, es decir, son formas equivalentes de decir los mismo:

```
int sumar(int x, int y){
     return (x+y);
}
```

Ejemplo 2: impresión por pantalla de los elementos de un arreglo de dimensión 10.

```
void imprimirArreglo( int[] a) {
    for (int i = 0; i < 10;i++) {
          System.out.println (a[i]);
    }
}
```

1.8 Clases

Luego de haber hablado de los métodos, nos resta describir el concepto de clases. En los lenguajes Orientados a Objetos siempre vamos a estructurar nuestros programas o sistemas mediante clases. Las clases las podríamos comparar a las estructuras de los lenguajes estructurados, aunque la definición correcta de las clases es que son la definición de un objeto, en donde un objeto representa a alguna entidad de la vida real (por ejemplo, un objeto puede ser una persona, una casa, un auto, una formula, etc.). A continuación veremos como es la declaración y el uso de una clase.

Declaración
Vamos a declarar una clase con sus componentes, esos componentes son variables de la clase y sus métodos. Para indicar que es una clase se utiliza la palabra reservada del lenguaje class.

```
class <nombre_clase>{
    <tipo_variable> <nombre_variables>;  //variables de la clase
    <tipo_metodo> <nombre_metodo> (<parámetros_metodo>){
        instrucción 1;
        instrucción 2;
        ...
    }
}
```

Como ejemplo, vamos a declarar una clase punto, en donde tenemos dos componentes que representan las coordenadas `x` e `y` del punto:

```
class Punto{
    int x;
    int y;
}
```

Uso de clases como variables
El uso de las clases es similar al uso de los tipos primitivos, es decir, se debe declarar una variable del tipo de la clase:

```
<nombre_clase> <nombre_variable>;
```

Por ejemplo, si queremos declarar una variable `p` del tipo `Punto` debemos hacer:

```
Punto p;
```

Y luego antes de poder usar cualquier variable cuyo tipo sea una clase (es decir, no sea una variable de tipo primitivo) se le debe asignar memoria de la siguiente manera:

```
<nombre_variable> = new <nombre_clase>();
```

Continuando con el ejemplo de la variable `p` del tipo `Punto`, para asignar memoria haríamos lo siguiente:

```
p = new Punto();
```

Cabe aclarar que la declaración de una variable y su asignación de memoria se puede hacer en una misma instrucción de la siguiente manera:

```
<class_name> <nombre_variable> = new <nombre_clase>();
```

Utilizando lo mencionado anteriormente en nuestro ejemplo nos quedaría:

```
Punto p = new Punto();
```

Por último, nos resta ver como podemos acceder a los componentes de las clases, es decir, las variables[3] y métodos. Para ello se utiliza el punto "." de la siguiente manera:

```
<nombre_variable>.<nombre_componente>
```

Es decir, si queremos asignarles valores a las coordenadas `x` e `y` del punto `p` lo que haríamos es lo siguiente:

```
p.x = 5;  p.y = 10;
```

En lo referido a métodos y variables que hemos venido trabajando, existe el concepto de alcance o visibilidad de los métodos y variables, y es lo que nos permite indicar si una método o variable podrá ser utilizado por quién usa esa clase o no. Es decir, si un método o variables es privado solo se podrá utilizar dentro de la clase, en cambio si es público podrá ser utilizado tanto dentro de la clase como por quién usa esa clase. Para indicar que un método o variable es público se utiliza la palabra reservada del leguaje *public*, y para indicar si es privado la palabra *private* (por defecto si no se indica nada el lenguaje considera que la variable o método es privada). Supongamos el siguiente ejemplo para mostrar lo mencionado anteriormente:

```
class Persona{
    String nombre;
    String apellido;
    String calle;
    int numero;
    String ciudad;

    public void setearNombre(String n, String a){
        nombre = n;
        apellido = a;
    }

    public void setearDireccion(String c, int n, String ciu){
        Calle = c;
        Numero = n;
        Ciudad = ciu;
    }
}
```

[3]Debemos tener en cuenta que no hemos introducido el concepto de métodos privados y públicos hasta el momento, ya que sólo podríamos acceder desde afuera si el método es público. Este concepto se va a ver en los próximos párrafos. No se verá en el curso el concepto de variables o métodos protegidos

Tenemos una clase que representa una persona, que tiene un nombre, apellido, la calle, número y ciudad en donde vive. Para asignarle el nombre y el apellido a la persona tenemos un método público y para asignarle la calle, número y ciudad otro. Estos dos métodos serán públicos y se deberán ser utilizados por quién usa la clase, pero las variables en donde guardamos esos valores no podrán ser accedidas por quién usa la clase con lo cual serán privadas. Para el ejemplo se utilizará un tipo de dato provisto por *Java* que es `String` que simboliza una cadena de caracteres y se puede imprimir como vimos anteriormente.

Unidad 2

TDA: conceptos y especificación

En este capítulo vamos a introducir el concepto de TDA, qué son, cómo se definen y cómo se utilizan. Vamos a introducir también la notación en lenguaje de programación que vamos a utilizar. Para los ejemplos, vamos a utilizar los TDA Pila, Cola, Cola con Prioridad, Conjunto y Diccionario.

2.1 Tipos de Datos Abstractos: Definición

Ya hemos definido en cursos anteriores los conceptos de Tipo de Dato y de Estructura de Dato. Vamos a recordar sus definiciones para poder introducir el nuevo concepto, que será el eje de la materia.
Cuando hablamos del *tipo de dato* de una variable, nos referimos al conjunto de valores que puede tomar una variable, y el conjunto de operaciones que se pueden aplicar sobre el mismo. Por ejemplo si tomamos el tipo de dato `int` sabemos que el conjunto de valores que puede tomar son todos los enteros y el conjunto de operaciones serán la suma, resta, multiplicación, división y todas las demás que permita el lenguaje utilizado. Se los puede clasificar según su contenido en simples o estructurados, según contenga un único valor o un conjunto de valores. Ejemplos de tipos simples son enteros, reales, caracteres, etc. Ejemplos de estructurados pueden ser arreglos o las estructuras de C. Si son estructurados, se pueden clasificar en homogéneos o heterogéneos según sus elementos sean de un mismo tipo(por ejemplo arreglos) o de diferentes tipos (por ejemplo estructuras). También podríamos clasificarlos en predefinidos por el lenguaje o definidos por el usuario.
Una *estructura de datos* es una definición que se realiza utilizando los mecanismos propuestos por el lenguaje, que permite almacenar un conjunto de valores. Según la estructura definida será la forma de accederlas.
Un tipo de dato abstracto (TDA) es un tipo de dato según lo explicado más arriba, donde el concepto de abstracción está asociado al modo en que se definen los TDA. Este modelo estará formado por las operaciones que tiene y el comportamiento

asociado a esas operaciones, es decir, estará centrado en qué hace el TDA y no cómo lo hace, es decir, la abstracción significa el ocultamiento del cómo.

2.2 Especificación de un TDA

Como dijimos en la definición de TDA, vamos a separar qué hace un TDA de cómo lo hace. Y para entender esta diferencia, veremos cómo se pueden definir TDA y utilizarlos sin necesidad de conocer cómo se hace cada una de las operaciones. De hecho, para una misma definición de TDA puede haber diferentes maneras de realizar lo definido por ese TDA.
A la definición del TDA la llamaremos *especificación* y a la forma de llevar a cabo lo definido por un TDA lo denominaremos *implementación*. Por lo tanto, para un mismo TDA vamos a tener diferentes implementaciones.
La notación de *Java* que utilizaremos para especificar TDA es la notación de `interfaces`. Para especificar el TDA `Ejemplo`, notaremos:

```
public interface EjemploTDA {

}
```

Para especificar una operacion de un TDA utilizaremos un prototipo de un método en *Java*, que incluye el nombre de la operación, los parámetros que recibe con su tipo, y el tipo de dato que retornan. Además, se incluirá un comentario de lo que hace la operación, si no queda claro del nombre y las precondiciones de la misma. Por ejemplo:

```
public interface EjemploTDA {

// Descripción de la acción que tiene el método
    int metodo1(int a, int b);
}
```

2.3 TDA con estructura lineal

Vamos a comenzar a especificar TDA con comportamientos ya conocidos de cursos previos para entender el concepto, para luego avanzar con nuevas estructuras. Durante todo el curso, vamos a suponer que el conjunto de valores que se almacenarán en las diferentes estructuras serán enteros. Esto no es una restricción de los TDA ni tampoco del lenguaje, sino una forma de simplificar la codificación.

2.3.1 Pila

El primer TDA a especificar será el tipo de dato abstracto que describe el comportamiento de una estructura pila. La *pila* es una estructura que permite almacenar conjuntos de valores, eliminarlos y recuperarlos, con la particularidad de que el elemento que se recupera o elimina es el último que ingresó. Por lo tanto, para poder definir el comportamiento de la pila vamos a utilizar la siguiente lista de operaciones:

- Apilar: permite agregar un elemento a la pila. Se supone que la pila está inicializada.
- Desapilar: permite eliminar el último elemento agregado a la pila. Se supone como precondición que la pila no esté vacía.
- Tope: permite conocer cuál es el último elemento ingresado a la pila. Se supone que la pila no está vacía.
- PilaVacía: indica si la pila contiene elementos o no. Se supone que la pila está inicializada.
- InicializarPila: permite inicializar la estructura de la pila.

Luego de listar las operaciones del TDA pila vamos a describir el mismo con *Java*:

```
public interface PilaTDA {
    void InicializarPila();
    // siempre que la pila esté inicializada
    void Apilar(int  x);
    // siempre que la pila esté inicializada y no esté vacía
    void Desapilar();
    // siempre que la pila esté inicializada
    boolean PilaVacia();
    // siempre que la pila esté inicializada y no esté vacía
    int Tope();
}
```

A continuación vamos a realizar un ejemplo en donde utilizaremos la `PilaTDA` definida anteriormente. En el ejemplo, vamos a pasar los elementos de una pila origen a una pila destino, en donde se perderán los elementos de la pila origen y en la pila destino quedarán esos elementos en orden inverso a la origen.

```
public void PasarPila(PilaTDA origen, PilaTDA destino){
    while (!origen.PilaVacia()){
        destino.Apilar(origen.Tope());
        origen.Desapilar();
    }
}
```

Otro ejemplo de utilización de la `PilaTDA`, es si queremos escribir un método que nos devuelva la suma de los valores que contiene la pila. En éste caso el método va a retornar un valor del tipo int.

```
public int SumarElementosPila(PilaTDA p){
    int suma = 0;
    while (!p.PilaVacia()){
        suma = suma + p.Tope();
        p.Desapilar();
    }
    return suma;
}
```

2.3.2 Cola

Luego de haber introducido el concepto de TDA, y para ello utilizar la estructura pila, ahora vamos a ver otra estructura que es la cola. Una *cola* es una estructura que nos permite almacenar valores, recuperar y eliminar el primer valor ingresado, además de que nos permite conocer si la cola tiene o no elementos. Por lo tanto, si analizamos la diferencia en cuanto al comportamiento de una pila y una cola, esa diferencia esta dada en la forma en que recuperan y eliminan los valores en cada estructura.
A continuación vamos a listar las operaciones, junto a sus comportamientos, que nos permite especificar el TDA de una cola:

- Acolar: permite agregar un elemento a la cola. Se supone que la cola está inicializada.
- Desacolar: permite eliminar el primer elemento agregado a la cola. Se supone como precondición que la cola no esté vacía.
- Primero: permite conocer cuál es el primer elemento ingresado a la cola. Se supone que la cola no está vacía.
- ColaVacía: indica si la cola contiene elementos o no. Se supone que la cola está inicializada.
- InicializarCola: permite inicializar la estructura de la cola.

Es decir, el tipo de dato abstracto `ColaTDA` nos quedaría de la siguiente manera:

```
public interface ColaTDA {
    void InicializarCola();
    // siempre que la cola esté inicializada
    void Acolar(int x);
    // siempre que la cola esté inicializada y no esté vacía
```

```
    void Desacolar();
    // siempre que la cola esté inicializada
    boolean ColaVacia();
    // siempre que la cola esté inicializada y no esté vacía
    int Primero();
}
```

De la misma manera que utilizamos como ejemplo pasar los elementos de una pila a otra, ahora vamos a escribir un método que nos permita pasar los elementos de una cola a otra. En este caso particular, a diferencia de las pilas, por el comportamiento particular de las colas, los elementos en la cola destino quedarán en el mismo orden que en la cola origen.

```
public void pasarCola(ColaTDA origen, ColaTDA destino){
    while (!c.ColaVacia()){
        destino.Acolar(origen.Primero());
        destino.Desacolar();
    }
}
```

2.3.3 Cola con Prioridad

En las secciones anteriores definimos estructuras en donde los elementos se recuperan y eliminan de acuerdo al orden en que habían ingresado a la estructura, es decir, en el caso de una pila se podía recuperar o eliminar el último elemento ingresado, y en el caso de la cola el primero ingresado. En esta sección vamos a especificar una estructura que a cada elemento cuando ingresa a la misma se le va a asignar una prioridad, y luego los elementos van a ser recuperados o eliminados de la estructura de acuerdo a esa prioridad. A esta estructura la denominaremos *cola de prioridad*. Este tipo de estructura es utilizado, por ejemplo, en las colas para embarazadas o discapacitados de los supermercados, ó en las colas de espera en las guardias de los hospitales, en dónde en este último caso la prioridad de atención al paciente se le da de acuerdo con el síntoma de problema de salud que haya llegado al hospital.
A continuación vamos a listar las operaciones, junto a sus comportamientos, que nos permite especificar el TDA de una cola con prioridad:

- AcolarPrioridad: permite agregar un elemento a la cola con una cierta prioridad dada. Se supone que la cola está inicializada.
- Desacolar: permite eliminar el elemento con mayor prioridad en la cola (en caso de tener dos elementos con la misma prioridad sale el primero ingresado). Se supone como precondición que la cola no esté vacía.

- Primero: permite conocer cuál es el elemento de mayor prioridad ingresado a la cola (en caso de tener dos elementos con la misma prioridad devuelve el primero ingresado). Se supone que la cola no está vacía.
- ColaVacía: indica si la cola contiene elementos o no. Se supone que la cola está inicializada.
- Prioridad: permite conocer la prioridad del elemento con mayor prioridad de la cola.
- InicializarCola: permite inicializar la estructura de la cola.

Es decir, el tipo de dato abstracto ColaPrioridadTDA nos quedaría de la siguiente manera:

```
public interface ColaPrioridadTDA {
    void InicializarCola();
    // siempre que la cola esté inicializada
    void AcolarPrioridad(int x, int prioridad);
    // siempre que la cola esté inicializada y no esté vacía
    void Desacolar();
    // siempre que la cola esté inicializada y no esté vacía
    int Primero();
    // siempre que la cola esté inicializada
    boolean ColaVacia();
    // siempre que la cola esté inicializada y no esté vacía
    int Prioridad();
}
```

Podemos establecer las siguientes conclusiones. Si acolamos todos los elementos con la misma prioridad, la cola con prioridad se comporta como una cola común. Si acolamos todos los elementos con la prioridad igual a su valor, la cola con prioridad puede ordenar automáticamente los elementos.

2.3.4 Conjunto

El siguiente tipo de dato abstracto que vamos a definir será el conjunto. El *conjunto* es una estructura que nos permite guardar elementos sin que los mismos se repitan y en donde no se tiene un orden, y además nos permite conocer si un elemento dado se encuentra o pertenece a la estructura. Debido a que la estructura no tiene un orden, cuando recuperamos un dato la estructura nos devuelve uno cualquiera que pertenece a ella, y por otro lado cuando queremos eliminar una valor debemos indicarle cuál es. A partir de esta descripción de comportamiento del conjunto, podemos decir que la lista de operaciones que define el TDA de conjunto es la siguiente:

- Agregar: permite agregar un elemento al conjunto. Se supone que el conjunto está inicializado.

- Sacar: permite eliminar del conjunto un elemento dado. Se supone como precondición que el conjunto no está vacío.
- Elegir: devuelve un elemento cualquiera del conjunto. Se supone como precondición que el conjunto no está vacío.
- Pertenece: permite conocer si un elemento dado se encuentra en el conjunto. Se supone que el conjunto está inicializado.
- ConjuntoVacío: indica si el conjunto contiene elementos o no. Se supone que el conjunto está inicializado.
- InicializarConjunto: permite inicializar la estructura del conjunto.

Es decir, el tipo de dato abstracto ConjuntoTDA nos quedaría de la siguiente manera:

```
public interface ConjuntoTDA {
    void InicializarConjunto();
    // siempre que el conjunto esté inicializado
    boolean ConjuntoVacio();
    // siempre que el conjunto esté inicializado
    void Agregar(int x);
    // siempre que el conjunto esté inicializado y no esté vacío
    int Elegir();
    // siempre que el conjunto esté inicializado
    void Sacar(int x);
    // siempre que el conjunto esté inicializado
    boolean Pertenece(int x);
}
```

Como ejemplo de uso del ConjuntoTDA, supongamos que queremos indicar si dos conjuntos son iguales, entendiendo que dos conjuntos son iguales si tienen los mismos elementos.

```
    public boolean SonConjuntosIguales(ConjuntoTDA c1,
       ConjuntoTDA c2){
        int elemento;
        boolean sonIguales = true;
        while (!c1.ConjuntoVacio() && !c2.ConjuntoVacio() &&
           sonIguales){
            elemento = c1.Elegir();
            if (!c2.Pertenece(elemento)){
                sonIguales = false;
            }else{
                c1.Sacar(elemento);
                c2.Sacar(elemento);
            }
        }
```

```
        if (c1.ConjuntoVacio() && c2.ConjuntoVacio()){
            return true;
        }else{
            return false;
        }
    }
```

2.3.5 Diccionario

El último tipo de dato abstracto que vamos a definir en esta unidad será el diccionario. La estructura de datos diccionario se caracteriza porque cada valor ingresa a la estructura asociado a una clave, y estás claves existen siempre que tengan valor asociado y son únicas. En el presente curso se van a analizar dos comportamientos diferentes de la estructura diccionario, el primero de ellos será en donde cada clave puede tener asociado un único valor, estructura que denominaremos *DiccionarioSimple*, y el segundo en donde cada clave tiene asociado un conjunto de valores, estructura que denominaremos *DiccionarioMultiple*.

Diccionario Simple

A continuación vamos a listar las operaciones, junto a sus comportamientos, que nos permite especificar el TDA de un diccionario simple:

- Agregar: dada una clave y un valor, agrega al diccionario el valor quedando asociado a la clave. Si ya existe la misma clave con otro valor se sobreescribe el valor, dejando el nuevo ingresado. Para poder agregar un par clave/valor la estructura debe estar inicializada.
- Eliminar: dada una clave elimina el valor asociado a la clave, y por consiguiente la clave ya que el diccionario no puede contener claves sin valores asociados. Si la clave no existe no se hace nada. Se supone que el diccionario esta inicializado.
- Recuperar: dada una clave devuelve el valor asociado a la clave. La clave dada debe pertenecer al diccionario. Se supone que el diccionario esta inicializado.
- Claves: devuelve el conjunto de todas las claves definidas en el diccionario. Se supone que el diccionario esta inicializado.
- InicializarDiccionario: permite inicializar la estructura del diccionario.

Es decir, el tipo de dato abstracto DiccionarioSimpleTDA nos quedaría de la siguiente manera:

```
public interface DiccionarioSimpleTDA {
        // siempre que el diccionario esté inicializado
   void InicializarDiccionario();
        // siempre que el diccionario esté inicializado
   void Agregar(int clave, int valor);
        // siempre que el diccionario esté inicializado
   void Eliminar(int clave);
        // siempre que el diccionario esté inicializado y la
           clave exista en el mismo
   int Recuperar(int clave);
        // siempre que el diccionario esté inicializado
   ConjuntoTDA Claves();
}
```

Diccionario Múltiple

El diccionario múltiple tiene como característica principal que cada clave del diccionario puede tener asociado un conjunto de valores.
A continuación vamos a listar las operaciones, junto a sus comportamientos, que nos permite especificar el TDA de un diccionario múltiple:

- Agregar: dada una clave y un valor, agrega al diccionario el valor quedando asociado a la clave. Una misma clave puede tener asociada un conjunto de valores, pero esos valores no se pueden repetir. Para poder agregar un par clave/valor la estructura debe estar inicializada.

- Eliminar: dada una clave elimina todos los valores asociados a la clave, y por consiguiente la clave ya que el diccionario no puede contener claves sin valores asociados. Si la clave no existe no se hace nada. Se supone que el diccionario esta inicializado.

- EliminarValor: dada una clave y un valor se elimina el valor asociado a la clave, y en caso de que la clave o el valor no existan no se hace nada. Si al eliminar el valor, la clave no tiene otros valores asociados se debe eliminar la misma. Se supone que el diccionario esta inicializado.

- Recuperar: dada una clave devuelve el conjunto de valores asociados a la misma. Si la clave dada no pertenece al diccionario, se debe devolver un conjunto vacío. Se supone que el diccionario esta inicializado.

- Claves: devuelve el conjunto de todas las claves definidas en el diccionario. Se supone que el diccionario esta inicializado.

- InicializarDiccionario: permite inicializar la estructura del diccionario.

Es decir, el tipo de dato abstracto DiccionarioMultipleTDA nos quedaría de la siguiente manera:

```
public interface DiccionarioMultipleTDA {
   void InicializarDiccionario();
        // siempre que el diccionario esté inicializado
   void Agregar(int clave, int valor);
        // siempre que el diccionario esté inicializado
   void Eliminar(int clave);
        // siempre que el diccionario esté inicializado
   void EliminarValor(int clave, int valor);
        // siempre que el diccionario esté inicializado
   ConjuntoTDA Recuperar(int clave);
        // siempre que el diccionario esté inicializado
   ConjuntoTDA Claves();
}
```

2.4 Resumen

Durante esta unidad hemos visto el concepto de tipo de dato abstracto, y la definición y uso de cinco tipos de datos abstractos como son Pila, Cola, Cola con Prioridad, Conjunto y Diccionario (en sus dos variantes, simple y múltiple). Como mencionamos cuando definimos un TDA, en donde dijimos que el TDA especifica el comportamiento de una estructura independientemente de cómo sea la implementación, y que por otra parte, íbamos a tener diferentes implementaciones para un mismo TDA, estas diferentes implementaciones es lo que vamos a ver en las unidades siguientes.

Unidad 3

Implementación

El objetivo de esta unidad es entender el concepto de implementación y su clara diferencia con la especificación de un TDA. Para eso vamos a ver diferentes implementaciones de los TDA definidos anteriormente, utilizando como estrategia de implementación estructuras de tamaño fijo, es decir, arreglos. Luego de haber realizado las diferentes implementaciones veremos el costo asociado a cada una de las operaciones de cada TDA de acuerdo a la implementación realizada.

3.1 Implementación de TDAs

Durante esta sección veremos diferentes implementaciones de PilaTDA, ColaTDA, ColaPrioridadTDA y ConjuntoTDA.
Cuando hablamos de implementación nos estamos refiriendo a que debemos definir los siguientes puntos:

- En dónde se va a guardar la información
- De qué manera se va guardar esa información, es decir, definir el criterio con que se va a guardar y recuperar los datos
- Escribir / implementar cada una de las operaciones del TDA que se esta implementando

En esta sección las estructuras utilizada para guardar la información van a ser arreglos, lo cual va a tener una restricción que es que tienen tamaño fijo. En secciones posteriores veremos otra forma de guardar la información.
Para implementar un TDA en *Java*, vamos a escribir una clase, a la que le indicaremos que es la implementación de un TDA con la palabra reservada `implements`. Luego definiremos la estructura de datos, y por último, codificaremos cada uno de los métodos especificados. Si fuera necesario, se podrán agregar métodos auxiliares, que serán internos a la clase.

```
public class CLASE implements claseTDA{

}
```

3.1.1 Implementación de PilaTDA

Como hemos expresado en la unidad anterior, para una misma especificación de TDA pueden existir diferentes implementaciones, y de hecho, existen muchas. Por ese motivo para PilaTDA vamos a ver algunas posibles estrategias.

Estrategia 1: Implementación en donde se guardan los datos en un arreglo y además se tiene una variable que indica la cantidad de elementos que se tienen guardados en la pila. Cuando agregamos un nuevo elemento a la pila, el mismo se guarda en la posición indicada por la variable que me indica la cantidad de elementos. Cuando se tiene que desapilar un elemento de la pila, solo es necesario decrementar en una unidad la variable que me indica la cantidad de elementos. Se puede ver el esquema en la figura 3.1.

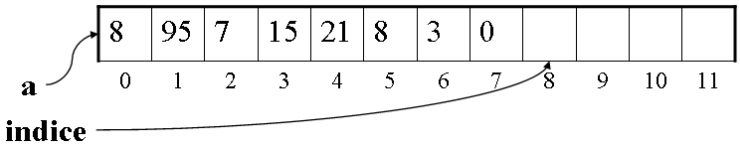

Figura 3.1: Implementación PilaTDA con Tope al Final

A continuación se muestra el código de esta implementación:

```
public class PilaTF implements PilaTDA{

    int[] a;  //arreglo en donde se guarda la información
    int indice; //variable entera en donde se guarda la cantidad
       de elementos que se tienen guardados

    public void InicializarPila(){
        a = new int[100];
        indice = 0;
    }

    public void Apilar(int x){
        a[indice] = x;
        indice++;
    }

    public void Desapilar(){
        indice--;
    }
```

```
    public boolean PilaVacia(){
        return (indice == 0);
    }

    public int Tope(){
        return a[indice-1];
    }
}
```

Estrategia 2: Otra variante de implementación es utilizar también un arreglo para almacenar los datos y tener una variable que indica la cantidad de elementos que se tienen guardados en la pila, pero cuando agregamos un nuevo elemento a la pila, en vez de hacerlo en la posición señalada por la variable que me indica la cantidad de elementos, se guarda en la primera posición (es decir, la posición 0 del arreglo). Esto tiene como desventaja que cuando se agrega un elemento se debe hacer un corrimiento del resto de los elementos hacia la derecha, y cuando se desapila un elemento de la pila nuevamente se tiene que hacer un corrimiento de elementos pero esta vez hacia la izquierda.
Se puede observar que la variante entre esta implementación y la anterior no esta dada en la estructura utilizada para guardar los elementos sino en el criterio utilizado para almacenar esa información.
A continuación se muestra el código de esta implementación:

```
public class PilaTI implements PilaTDA{
    int[] a;  //arreglo en donde se guarda la información
    int indice; //variable entera en donde se guarda la cantidad
       de elementos que se tienen guardados

    public void InicializarPila(){
        a = new int[100];
        indice = 0;
    }

    public void Apilar(int x){
        for (int i = indice-1; i >=0; i--)
            a[i+1] = a[i];
        a[0] = x;
        indice++;
    }

    public void Desapilar(){
        for (int i = 0; i <indice; i++)
            a[i] = a[i+1];
        indice--;
    }

    public boolean PilaVacia(){
```

```
            return (indice == 0);
    }
    public int Tope(){
        return a[0];
    }
}
```

Estrategia 3: Podemos mencionar otra estrategia de implementación en donde variamos la estructura utilizada para guardar la información, es decir, también utilizamos un arreglo pero para guardar la cantidad de elementos que tenemos en vez de hacerlo en una variable utilizamos la posición 0 del arreglo.

3.1.2 Implementación de ColaTDA

De la misma manera que para la estructura de Pila vamos a presentar diferentes estrategias de implementación para la estructura Cola. Todas las estrategias de implementación se basarán por el momento en estructuras de arreglos.

Estrategia 1: *Último elemento ingresado a la cola en la posición inicial del arreglo*: Implementación en donde se guardan los datos en un arreglo y además se tiene una variable que indica la cantidad de elementos que se tienen guardados en la cola. Cuando agregamos un nuevo elemento a la cola, el mismo se guarda en la posición cero del arreglo, por lo cual se requiere previo a la inserción un corrimiento a derecha de los elementos que se encuentran en la cola. Cuando se tiene que desacolar un elemento de la cola, solo es necesario decrementar en una unidad la variable que me indica la cantidad de elementos. A continuación se muestra el código de esta implementación:

```
public class ColaPU implements ColaTDA{
    int[] a;
    int indice;

    public void InicializarCola(){
        a = new int[100];
        indice = 0;
    }

    public void Acolar(int x){
        for (int i = indice-1; i >=0; i--)
            a[i+1] = a[i];
        a[0] = x;
        indice++;
    }

    public void Desacolar(){
        indice--;
    }
```

```
    public boolean ColaVacia(){
       return (indice == 0);
    }
    public int Primero(){
       return a[indice-1];
    }
}
```

Estrategia 2: *Primer elemento ingresado a la cola en la posición inicial del arreglo*: Implementación en donde se guardan los datos en un arreglo y además se tiene una variable que indica la cantidad de elementos que se tienen guardados en la cola. Cuando agregamos un nuevo elemento a la cola, el mismo se guarda en la posición indicada por la variable que guarda la cantidad de elementos que tiene la cola hasta el momento. Cuando se tiene que desacolar un elemento de la cola se requiere un corrimiento a izquierda en una posición de los elementos que se encuentran en la cola. A continuación se muestra el código de esta implementación:

```
public class ColaPI implements ColaTDA{

    int[] a;
    int indice;

    public void InicializarCola(){
        a = new int[100];
        indice = 0;
    }

    public void Acolar(int x){
        a[indice] = x;
        indice++;
    }

    public void Desacolar(){
        for (int i = 0; i < indice-1; i++)
            a[i] = a[i+1];
        indice--;
    }

    public boolean ColaVacia(){
       return (indice == 0);
    }

    public int Primero(){
       return a[0];
    }
}
```

Estrategia 3: De la misma manera que para Pilas, podemos mencionar otra estrategia de implementación en donde variamos la estructura utilizada para guardar la información, es decir, también utilizamos un arreglo pero para guardar la cantidad de elementos que tenemos en vez de hacerlo en una variable utilizamos la posición 0 del arreglo.

3.1.3 Implementación de ColaPrioridadTDA

Dentro de las posibles implementaciones para la cola de prioridad veremos las siguientes:

Estrategia 1: Consideramos como estructuras dos arreglos, en uno de ellos tendremos los elementos y en el otro la prioridad de cada uno de esos elementos. Como veremos se tendrán que mantener ambos arreglos sincronizados en cuanto a que para una posición dada se tendrá en un arreglo el valor del elemento y en el otro la prioridad que le corresponde a ese elemento. Además, como en todos los casos que hemos trabajado con arreglos tendremos una variable en donde llevemos la cantidad de elementos que se tienen. Se puede ver un esquema de esta estrategia en la figura 3.2. El elemento con mayor prioridad estará en la posición anterior a la marcada por `indice`.

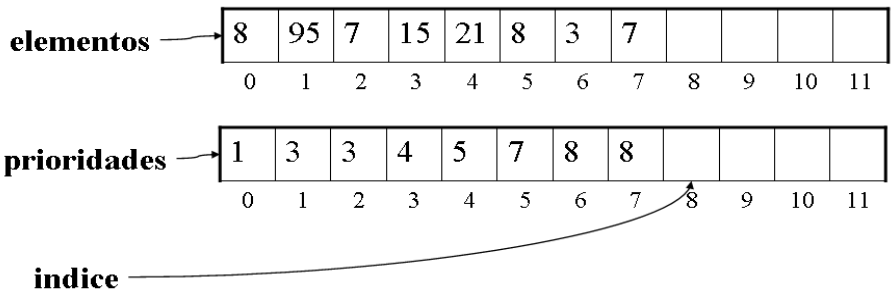

Figura 3.2: Implementación ColaPrioridadTDA con estrategia 1

```
public class ColaPrioridadDA implements ColaPrioridadTDA{

    int[] elementos;
    int[] prioridades;
    int indice;

    public void InicializarCola(){
        indice = 0;
        elementos = new int[100];
        prioridades = new int[100];
    }

    public void AcolarPrioridad(int x, int prioridad){
```

```
        //desplaza a derecha los elementos de la cola mientras
        //estos tengan mayor o igual prioridad que la de x
        int j = indice;
        for ( ; j>0 && prioridades[j-1]>= prioridad; j-- ){
            elementos[j] = elementos[j-1];
            prioridades[j] = prioridades[j-1];
        }
        elementos[j] = x;
        prioridades[j] = prioridad;
        indice++;
    }

    public void Desacolar(){
        indice--;
    }

    public int Primero(){
        return elementos[indice-1];
    }

    public boolean ColaVacia(){
        return (indice == 0);
    }

    public int Prioridad(){
        return prioridades[indice-1];
    }
}
```

Estrategia 2: Otra opción de implementación de las colas de prioridades es considerar un arreglo cuyos elementos representan una estructura (por ser en *Java* una clase) que contiene el valor y la prioridad asociada a ese valor. Se puede ver un esquema de esta implementación en la figura 3.3.

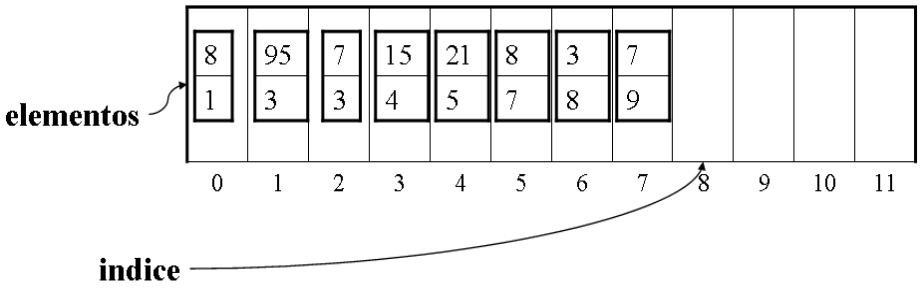

Figura 3.3: Implementación de ColaPrioridadTDA con estrategia 2

A continuación se muestra el código de ésta implementación.

```
public class ColaPrioridadAO implements ColaPrioridadTDA {

    class Elemento{
        int valor;
        int prioridad;
    }

    Elemento[] elementos;
    int indice;

    public void InicializarCola(){
        indice = 0;
        elementos = new Elemento[100];
    }

    public void AcolarPrioridad(int x, int prioridad){
        int j = indice;

        //desplaza a derecha los elementos de la cola mientras
        //estos tengan mayor o igual prioridad que la de x
        for (; j>0 && elementos[j-1].prioridad>=prioridad; j--){
            elementos[j] = elementos[j-1];
        }
        elementos[j]= new Elemento();
        elementos[j].valor=x;
        elementos[j].prioridad = prioridad;
        indice++;
    }

    public void Desacolar(){
        elementos[indice - 1]= null;
        indice--;
    }

    public int Primero(){
        return elementos[indice-1].valor;
    }

    public boolean ColaVacia(){
        return (indice == 0);
    }

    public int Prioridad(){
        return elementos[indice-1].prioridad;
    }
}
```

3.1.4 Implementación de ConjuntoTDA

A continuación vamos a llevar a cabo la implementación del TDA de conjuntos, a partir de la utilización de un arreglo. En el caso del conjunto debido a que los elementos no tienen un orden, cuando se elimina una elemento se puede colocar el último elemento que se encuentra en el arreglo en la posición del elemento a eliminar, y de esta manera evitar el desplazamiento de todos los elementos. Para implementar el método `Elegir`, dado que la especificación no indica cuál es el elemento a recuperar, vamos a recuperar el elemento que está en la primera posición del arreglo.

```
public class ConjuntoTA implements ConjuntoTDA{
    int[] a;
    int cant;

    public void Agregar(int x) {
        if (!this.Pertenece(x)){
            a[cant] = x;
            cant++;
        }
    }

    public boolean ConjuntoVacio() {
        return cant == 0;
    }

    public int Elegir() {
        return a[cant - 1];
    }

    public void InicializarConjunto() {
        a = new int[100];
        cant = 0;
    }

    public boolean Pertenece(int x) {
        int i = 0;
        while (i<cant && a[i]!=x)
            i++;
        return ( i < cant);

    }

    public void Sacar(int x) {
        int i = 0;
        while (i<cant && a[i]!=x)
            i++;

        if (i < cant){
            a[i] = a[cant-1];
            cant--;
```

```
        }
    }
}
```

Una implementación particular de conjuntos es la que denominamos conjunto de universo acotado, en donde tenemos un arreglo cuyas posiciones representan la presencia o ausencia de un valor en el conjunto. Es decir, si consideramos un conjunto de 15 elementos (del 0 al 14), tendremos un arreglo de 15 posiciones, en donde tendremos por ejemplo un 1 en la posición 5 si es que el elemento 5 pertenece al conjunto, y 0 en caso contrario. Esta implementación tiene limitaciones en cuanto a los valores que puede tomar el conjunto, pero puede ser útil para aquellos casos en donde esta limitación no sea un problema ya que tenemos una manera más eficiente de agregar o eliminar un valor del arreglo, así como de conocer si un cierto valor pertenece al mismo. Todas las operaciones sobre este tipo de implementación para el conjunto son de costos constantes.

3.1.5 Implementación de Diccionario

A continuación vamos a llevar a cabo la implementación de los TDA de diccionarios, a partir de la utilización de arreglos. En el caso del diccionarioSimple se emplea un único arreglo de elementos que se definen por una clave y un valor. mientras que en el diccionarioMultiple vamos a emplear un arreglo de elementos donde cada uno de estos contiene una clave y un arreglo de valores. Dado que las claves y valores de un diccionario no tienen que mantener un criterio de ordenaniento particular, cuando se elimina una elemento se puede colocar (al igual que en la implentación del conjunto) el último elemento que se encuentra en el arreglo en la posición del elemento a eliminar, y de esta manera evitar el desplazamiento de todos los elementos.

Diccionario Simple

```
public class DicSimpleA implements DiccionarioSimpleTDA {

    class Elemento{
        int clave;
        int valor;
    }

    Elemento[] elementos;
    int cant;

    public void InicializarDiccionario(){
        cant = 0;
        elementos = new Elemento[100];
    }
```

```
    public void Agregar(int clave, int valor){
        int pos = Clave2Indice(clave);
        if(pos==-1) {
            pos=cant;
            elementos[pos]= new Elemento();
            elementos[pos].clave = clave;
            cant++;
        }
        elementos[pos].valor = valor;

    }

    private int Clave2Indice(int clave){
        int  i=cant-1;
        while(i>=0 && elementos[i].clave!=clave)
            i--;
        return i;
    }

    public void Eliminar(int clave) {
        int pos = Clave2Indice(clave);
        if(pos!=-1) {
            elementos[pos] = elementos[cant-1];
            cant--;
        }
    }

    public int Recuperar(int clave) {
        int pos = Clave2Indice(clave);
        return elementos[pos].valor;
    }

    public ConjuntoTDA Claves() {
        ConjuntoTDA c=new ConjuntoLD();
        c.InicializarConjunto();

        for(int i=0; i<cant; i++){
            c.Agregar(elementos[i].clave);
        }
        return c;
    }
}
```

Diccionario Mutiple

```
public class DicMultipleA implements DiccionarioMultipleTDA {

    class Elemento{
```

```
        int clave;
        int[] valores;
        int cantValores;
    }

    Elemento[] elementos;
    int cantClaves;

    public void InicializarDiccionario(){
        elementos = new Elemento[100];
        cantClaves = 0;
    }

    public void Agregar(int clave, int valor){
        int posC = Clave2Indice(clave);
        if(posC==-1) {
            posC = cantClaves;
            elementos[posC]= new Elemento();
            elementos[posC].clave = clave;
            elementos[posC].cantValores = 0;
            elementos[posC].valores = new int[100];
            cantClaves++;
        }

        Elemento e =  elementos[posC];
        int posV = Valor2Indice(e, valor);
        if(posV==-1) {
            e.valores[e.cantValores] = valor;
            e.cantValores++;
        }
    }

    private int Clave2Indice(int clave){
        int  i = cantClaves-1;
        while(i>=0 && elementos[i].clave!=clave)
            i--;
        return i;
    }

    public void Eliminar(int clave) {
        int pos = Clave2Indice(clave);
        if(pos!=-1) {
            elementos[pos] = elementos[cantClaves-1];
            cantClaves--;
        }
    }

    public void EliminarValor(int clave, int valor) {
        int posC = Clave2Indice(clave);
        if(posC!=-1) {
            Elemento e = elementos[posC];
            int posV = Valor2Indice(e, valor);
```

```
            if(posV!=-1) {
                e.valores[posV] = e.valores[e.cantValores-1];
                e.cantValores--;
                if (e.cantValores==0) {
                    Eliminar(clave);
                }
            }
        }
    }

    private int Valor2Indice(Elemento e, int valor) {
        int  i = e.cantValores-1;
        while(i>=0 && e.valores[i]!=valor)
            i--;
        return i;
    }

    public ConjuntoTDA Recuperar(int clave) {
        ConjuntoTDA c=new ConjuntoLD();
        c.InicializarConjunto();

        int pos = Clave2Indice(clave);
        if (pos!=-1) {
            Elemento e= elementos[pos];
            for(int i=0; i<e.cantValores; i++){
                c.Agregar(e.valores[i]);
            }
        }
        return c;
    }

    public ConjuntoTDA Claves() {
        ConjuntoTDA c=new ConjuntoLD();
        c.InicializarConjunto();

        for(int i=0; i<cantClaves; i++){
            c.Agregar(elementos[i].clave);
        }
        return c;
    }
```

3.2 Análisis de Costos

Cuando queremos analizar o comparar diferentes implementaciones necesitamos algún parámetro o variables en la cual basar esa comparación, esas parámetros pueden ser:

- Cantidad de operaciones llevadas a cabo o utilizadas

- Cantidad de memoria utilizada

En el primer caso hablamos de análisis de complejidad temporal, y en el segundo de complejidad de espacio. En este curso haremos el análisis de complejidad temporal, y para ello realizaremos un análisis simplificado, sin entrar en conceptos demostrativos de complejidad temporal.
Para el análisis de costos de complejidad temporal se consideran los siguientes tres criterios a través de los cuales clasificaremos el costo de ejecución de un método:

- Costo constante (C): el costo no depende de la cantidad de elementos de entrada. Es el mejor de los costos posibles.
- Costo lineal (L): el costo depende linealmente de la cantidad de elementos de entrada, es decir que por cada elemento de entrada hay una cantidad constante de operaciones que se ejecutan. Es un costo bueno, aunque siempre depende del contexto.
- Costo polinómico (P): el costo es mayor a lineal. Un ejemplo de esta categoría sería el costo cuadrático. Es un costo no tan bueno, pero la interpretación depende del contexto.

Pila

	PilaTI	PilaTF
InicializarPila	C	C
Apilar	L	C
Desapilar	L	C
Tope	C	C
PilaVacia	C	C

Cola

	ColaPU	ColaPI
InicializarCola	C	C
Acolar	L	C
Desacolar	C	L
Primero	C	C
ColaVacia	C	C

Cola con Prioridad

	ColaPrioridadDA	ColaPrioridadAO
InicializarCola	C	C
AcolarPrioridad	L (2 n)	L (2 n)
Desacolar	C	C
Primero	C	C
ColaVacia	C	C
Prioridad	C	C

Conjunto

	Conjunto
InicializarConjunto	C
Agregar	L
Sacar	L
Pertenece	L
Elegir	C

Diccionarios

	DiccionarioSimpleA	DiccionarioMutipleA
InicializarDiccionario	C	C
Agregar	L	L
Eliminar	L	L
EliminarValor	-	L
Recuperar	L	L
Claves	L	L

Unidad 4

Implementación con estructuras dinámicas

En esta unidad vamos a implementar los TDA vistos en unidades anteriores utilizando ahora estructuras dinámicas. Una vez implementados, vamos a comparar los costos de estas implementaciones contra las anteriores. Al final de la unidad vamos a resolver problemas combinando estos TDA y más de una implementación de un mismo TDA.

4.1 Implementaciones dinámicas

Las implementaciones dinámicas se basan en el concepto de utilizar memoria dinámica, es decir que a diferencia de los arreglos, para los cuales definimos y reservamos la cantidad de memoria al momento de su creación, en las estructuras dinámicas, vamos reservando memoria a medida que se necesita. Asimismo la memoria que ya no se necesita se devuelve al sistema. Por las características de *Java*, la idea de devolver memoria es implícita, es decir, la memoria que no se utiliza, automáticamente queda disponible. Este mecanismo se denomina *Garbage Collection*.
La base de una estructura dinámica es lo que llamamos *célula* o *nodo*. La idea entonces es especificar un nodo que tenga la capacidad de almacenar un valor. Luego, se van a encadenar tantos nodos como sean necesarios.
Cabe recordar aquí que esta implementación no afectará en absoluto el comportamiento del TDA, es decir que todo aquel que haya usado el TDA con una implementación estática, lo podrá seguir usando de la misma forma, con una implementación dinámica sin modificar nada.
Volviendo a los nodos y la forma de encadenarlos, en cada nodo se definirá el espacio para almacenar el valor y para guardar una referencia a otro nodo. Así con un nodo, tenemos el valor y la referencia a otro nodo. Esto da lugar a una cadena de nodos potencialmente infinita. Un nodo que no apunta a ningún otro

nodo, tendra un valor especial en esa posición: `null`. La estructura del nodo se puede ver en la figura 4.1.

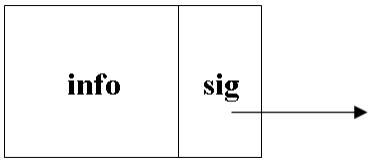

Figura 4.1: Nodo de la estructura dinámica lineal

Así, dos nodos encadenados quedan como en la figura 4.2.

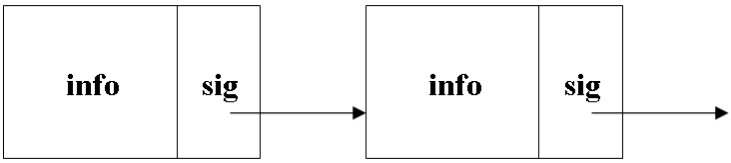

Figura 4.2: Dos nodos encadenados de la estructura dinámica lineal

Con esto, se puede generalizar a N nodos.
Entonces, en *Java*, la estructura del nodo sería:

```
class Nodo{
   int info;
   Nodo sig;
}
```

4.1.1 Pila

Vamos ahora a implementar el PilaTDA con una estructura dinámica. Para ello, recordemos la especificación del PilaTDA:

```
public interface PilaTDA {
    void InicializarPila();
    void Apilar(int  x);
    void Desapilar();
    boolean PilaVacia();
    int Tope();
}
```

La estrategia entonces será mantener una cadena de nodos, manteniendo un puntero al tope de la pila, este apuntará al siguiente, y así sucesivamente. Si la lista está vacía, será un punterno `null`. Para agregar un nodo, se lo crea, y se agrega al comienzo de la cadena. Para eliminarlo, simplemente se elimina el primer nodo, dejando el puntero apuntando al segundo si es que existe.

```
public class PilaLD implements PilaTDA{

    Nodo primero;

    public void InicializarPila(){
        primero = null;
    }

    public void Apilar(int x){
        Nodo aux = new Nodo();
        aux.info = x;
        aux.sig = primero;
        primero = aux;
    }

    public void Desapilar(){
        primero = primero.sig;
    }

    public boolean PilaVacia(){
        return (primero == null);
    }

    public int Tope(){
        return primero.info;
    }
}
```

4.1.2 Cola

Recordemos la especificación del ColaTDA:

```
public interface ColaTDA {
    void InicializarCola();
    void Acolar(int x);
    void Desacolar();
    boolean ColaVacia();
    int Primero();
}
```

Con la misma estrategia que para la pila, ahora vamos implementar la cola, esta vez, manteniendo un puntero al primer elemento y uno al último ya que por el comportamiento de este TDA, se deberá agregar por un extremo y eliminar por el otro. El orden en que estarán los elementos será el primero apunta al segundo, el segundo al tercero, así siguiendo, llegando al último elemento de la cadena que será el último elemento agregado a la Cola.

```
public class ColaLD implements ColaTDA{

    //Primer elemento en la cola
    Nodo primero;

    //Último elemento en la cola, es decir, el último agregado
    Nodo ultimo;

    public void InicializarCola(){
        primero = null;
        ultimo = null;
    }

    public void Acolar(int x){

        Nodo aux = new Nodo();
        aux.info = x;
        aux.sig = null;
        //Si la cola no está vacía
        if(ultimo != null)
            ultimo.sig = aux;
        ultimo = aux;

        // Si la cola estaba vacía
        if(primero == null)
            primero = ultimo;
    }

    public void Desacolar(){
        primero = primero.sig;

        // Si la cola queda vacía
        if(primero == null)
            ultimo = null;
    }

    public boolean ColaVacia(){
        return (ultimo == null);
    }

    public int Primero(){
        return primero.info;
    }
}
```

4.1.3 Cola con Prioridad

Recordemos la especificación del TDA ColaPrioridad:

```
public interface ColaPrioridadTDA {
```

```
    void InicializarCola();
    void AcolarPrioridad(int x, int prioridad);
    void Desacolar();
    int Primero();
    boolean ColaVacia();
    int Prioridad();
}
```

Aquí la estrategia será modificar el nodo, para que además del valor tenga la prioridad incluida:

```
class NodoPrioridad{
    int info;
    int prioridad;
    NodoPrioridad sig;
}
```

Y luego implementar el TDA manteniendo la cadena de nodos ordenada primero por prioridad y luego por orden de entrada, manteniendo un puntero al elemento con mayor prioridad, que es el primero que se deberá eliminar.

```
public class ColaPrioridadLD implements ColaPrioridadTDA {

    NodoPrioridad mayorPrioridad;

    public void InicializarCola(){
        mayorPrioridad = null;
    }

    public void AcolarPrioridad(int x, int prioridad){

        // Creo el nuevo nodo que voy a acolar
        NodoPrioridad nuevo = new NodoPrioridad();
        nuevo.info = x;
        nuevo.prioridad = prioridad;

        // Si la cola está vacía o bien es más prioritario que
        // el primero hay que agregarlo al principio
        if(mayorPrioridad == null ||
           prioridad > mayorPrioridad.prioridad) {
            nuevo.sig = mayorPrioridad;
            mayorPrioridad = nuevo;
        }
        else {
            //Sabemos que mayorPrioridad no es null
            NodoPrioridad aux = mayorPrioridad;

            while(aux.sig!=null && aux.sig.prioridad>=prioridad){
                aux = aux.sig;
            }
```

```
            nuevo.sig = aux.sig;
            aux.sig = nuevo;
        }
    }

    public void Desacolar(){
        mayorPrioridad = mayorPrioridad.sig;
    }

    public int Primero(){
        return mayorPrioridad.info;
    }

    public boolean ColaVacia(){
        return (mayorPrioridad == null);
    }

    public int Prioridad(){
        return mayorPrioridad.prioridad;
    }
}
```

4.1.4 Conjuntos

Para implementar el conjunto, a diferencia de las otras estructuras, no hay orden, por lo tanto la estrategia es simplemente mantener la cadena de nodos y siempre se deberá recorrerla para buscar un elemento o eliminarlo. Aquí presentamos una implementación, existen otras que mantienen la cadena ordenada. Con esto se mejoran algunos costos y se empeoran otros.
Recordemos entonces la especificación del ConjuntoTDA:

```
public interface ConjuntoTDA {
    void InicializarConjunto();
    boolean ConjuntoVacio();
    void Agregar(int x);
    int Elegir();
    void Sacar(int x);
    boolean Pertenece(int x);
}
```

Y la implementación propuesta es la siguiente:

```
public class ConjuntoLD implements ConjuntoTDA {

    Nodo c;

    public void InicializarConjunto() {
        c = null;
```

```
    }

    public boolean ConjuntoVacio() {
        return (c == null);
    }

    public void Agregar(int x) {
        /* Verifica que x no este en el conjunto */
        if (!this.Pertenece(x)){
            Nodo aux = new Nodo();
            aux.info = x;
            aux.sig = c;
            c = aux;
        }
    }

    public int Elegir() {
        return c.info;
    }

    public void Sacar(int x) {
        if(c!=null){
            // si es el primer elemento de la lista
            if (c.info == x) {
                c = c.sig;
            }else{
                Nodo aux = c;
                while (aux.sig!=null && aux.sig.info!=x)
                    aux = aux.sig;
                if (aux.sig != null)
                    aux.sig = aux.sig.sig;
            }
        }
    }

    public boolean Pertenece(int x) {
        Nodo aux = c;
        while ((aux != null) && (aux.info != x)){
            aux = aux.sig;
        }
        return (aux != null);
    }
}
```

4.1.5 Diccionarios

Para implementar los diccionarios, como en el caso del conjunto no hay necesidad mentener un orden, por lo tanto la estrategia es simplemente mantener la cadena de nodos y siempre se deberá recorrerla para buscar un elemento o eliminarlo.

Diccionario Simple

La implementación propuesta es la siguiente:

```
public class DicSimpleL implements DiccionarioSimpleTDA {

    class NodoClave{
        int clave;
        int valor;
        NodoClave sigClave;
    }

    NodoClave origen;

    public void InicializarDiccionario(){
       origen = null;
    }


    public void Agregar(int clave, int valor){
        NodoClave nc = Clave2NodoClave( clave);
        if (nc==null) {
            nc = new NodoClave();
            nc.clave = clave;
            nc.sigClave = origen;
            origen = nc;
        }
        nc.valor = valor;
    }

    private NodoClave Clave2NodoClave(int clave){
        NodoClave aux = origen;
        while (aux!=null && aux.clave!=clave){
            aux = aux.sigClave;
        }
        return aux;
    }

   public void Eliminar(int clave) {
        if(origen!=null) {
            if(origen.clave == clave) {
               origen = origen.sigClave;
            }
            else {
                NodoClave aux = origen;
                while (aux.sigClave != null && aux.sigClave.clave
                   !=clave){
                    aux = aux.sigClave;
                }
                if(aux.sigClave!=null) {
                    aux.sigClave= aux.sigClave.sigClave;
                }
```

```
            }
        }
    }

    public int Recuperar(int clave){
        NodoClave n = Clave2NodoClave(clave);
        return n.valor;
    }


    public ConjuntoTDA Claves(){
        ConjuntoTDA c = new ConjuntoLD();
        c.InicializarConjunto();

        NodoClave aux = origen;
        while (aux != null){
            c.Agregar(aux.clave);
            aux = aux.sigClave;
        }
        return c;
    }
}
```

Diccionario Multiple

La implementación propuesta es la siguiente:

```
public class DicMultipleL implements DiccionarioMultipleTDA {

    class NodoClave{
        int clave;
        NodoValor valores;
        NodoClave sigClave;
    }

    class NodoValor{
        int valor;
        NodoValor sigValor;
    }

    NodoClave origen;

    public void InicializarDiccionario(){
       origen = null;
    }


    public void Agregar(int clave, int valor){
        NodoClave nc = Clave2NodoClave( clave);
        if (nc==null) {
```

```
            nc= new NodoClave();
            nc.clave=clave;
            nc.sigClave= origen;
            origen = nc;
        }

        NodoValor aux = nc.valores;
        while (aux!=null && aux.valor!=valor) {
            aux= aux.sigValor;
        }
        if (aux==null) {
            NodoValor nv = new NodoValor();
            nv.valor = valor;
            nv.sigValor = nc.valores;
            nc.valores = nv;
        }
    }

    private NodoClave Clave2NodoClave(int clave){
        NodoClave aux = origen;
        while (aux!=null && aux.clave!=clave){
            aux = aux.sigClave;
        }
        return aux;
    }

    public void EliminarValor(int clave, int valor){

        if(origen!=null) {
            if(origen.clave == clave) {
                EliminarValorEnNodo(origen, valor);
                if(origen.valores==null) {
                    origen = origen.sigClave;
                }
            }
            else {
                NodoClave aux = origen;
                while (aux.sigClave != null && aux.sigClave.clave
                   !=clave){
                    aux = aux.sigClave;
                }
                if(aux.sigClave!=null) {
                    EliminarValorEnNodo(aux.sigClave, valor);
                    if(aux.sigClave.valores==null) {
                        aux.sigClave= aux.sigClave.sigClave;
                    }
                }
            }
        }
    }

    private void EliminarValorEnNodo(NodoClave nodo, int valor) {
```

```
        if(nodo.valores!=null) {
            if(nodo.valores.valor == valor) {
                nodo.valores = nodo.valores.sigValor;
            }
            else {
                NodoValor aux = nodo.valores;
                while (aux.sigValor != null && aux.sigValor.valor
                    !=valor){
                    aux = aux.sigValor;
                }
                if(aux.sigValor!=null) {
                    aux.sigValor= aux.sigValor.sigValor;
                }
            }
        }
    }

    public void Eliminar(int clave) {
        if(origen!=null) {
            if(origen.clave == clave) {
               origen = origen.sigClave;
            }
            else {
                NodoClave aux = origen;
                while (aux.sigClave != null && aux.sigClave.clave
                    !=clave){
                    aux = aux.sigClave;
                }
                if(aux.sigClave!=null) {
                    aux.sigClave= aux.sigClave.sigClave;
                }
            }
        }
    }

    public ConjuntoTDA Recuperar(int clave){
        NodoClave n = Clave2NodoClave(clave);

        ConjuntoTDA c = new ConjuntoLD();
        c.InicializarConjunto();
        if(n!=null) {
            NodoValor aux = n.valores;
            while (aux != null){
                c.Agregar(aux.valor);
                aux = aux.sigValor;
            }
        }
        return c;
    }


    public ConjuntoTDA Claves(){
```

```
        ConjuntoTDA c = new ConjuntoLD();
        c.InicializarConjunto();

        NodoClave aux = origen;
        while (aux != null){
            c.Agregar(aux.clave);
            aux = aux.sigClave;
        }
        return c;
    }
}
```

4.2 Comparación de costos

Vamos a comparar los costos de cada implementación respecto a la propia con estructuras estáticas.

Pila

	PilaTI	PilaTF	PilaLD
InicializarPila	C	C	C
Apilar	L	C	C
Desapilar	L	C	C
Tope	C	C	C
PilaVacia	C	C	C

Cola

	ColaPU	ColaPI	ColaLD
InicializarCola	C	C	C
Acolar	L	C	C
Desacolar	C	L	C
Primero	C	C	C
ColaVacia	C	C	C

Cola con Prioridad

	ColaPrioridadDA	ColaPrioridadAO	ColaPrioridadLD
InicializarCola	C	C	C
AcolarPrioridad	L (2 n)	L (2 n)	L (n)
Desacolar	C	C	C
Primero	C	C	C
ColaVacia	C	C	C
Prioridad	C	C	C

Conjunto

	ConjuntoA	ConjuntoLD
InicializarConjunto	C	C
Agregar	L	L
Sacar	L	L
Pertenece	L	L
Elegir	C	C

Diccionarios

	DicSimpleA	DicMutipleA	DicSimpleL	DicMutipleL
InicializarDiccionario	C	C	C	C
Agregar	L	L	L	L
Eliminar	L	L	L	L
EliminarValor	-	L	-	L
Recuperar	L	L	L	L
Claves	L	L	L	L

4.3 Resolución de problemas combinando TDAs e implementaciones

Ejemplo 1: Determinar si una Cola es capicúa.

```
public boolean esColaCapicua(ColaTDA c){
        ColaTDA aux1 = new ColaLD();
        aux1.InicializarCola();
        int cant = 0;
        PilaTDA p = new PilaLD();
        p.InicializarPila();

        while (!c.ColaVacia()){
            aux1.Acolar(c.Primero());
            cant++;
            c.Desacolar();
        }

        //Devuelve la parte entera de una división
        int mitad = cant/2;

        while (!aux1.ColaVacia() && mitad > 0){
            p.Apilar(aux1.Primero());
            aux1.Desacolar();
            mitad--;
        }

        //Retorna 0 si el número es par y 1 si es impar
        if ((cant%2) != 0){
            aux1.Desacolar();
        }
```

```
        while (!p.PilaVacia() && !aux1.ColaVacia()){
            if (p.Tope() != aux1.Primero()){
                return false;
            }
            p.Desapilar();
            aux1.Desacolar();
        }
        return (p.PilaVacia() && aux1.ColaVacia())
    }
```

Ejemplo 2: Eliminar los elementos repetidos de una Pila.

```
    public void eliminarRepetidos(PilaTDA p){
        PilaTDA aux = new PilaLD();
        aux.InicializarPila();
        ConjuntoTDA c = new ConjuntoLD();
        while (!p.PilaVacia()){
            int tope = p.Tope();
            if (!c.Pertenece(tope)){
                aux.Apilar(tope);
                c.Agregar(tope);
            }
            p.Desapilar();
        }

        while (!aux.PilaVacia()){
            p.Apilar(aux.Tope());
            aux.Desapilar();
        }
    }
```

Ejemplo 3: Calcular la diferencia simétrica entre dos conjuntos.

```
public ConjuntoTDA diferenciaSimetricaSinOperaciones(ConjuntoTDA
   c1, ConjuntoTDA c2){
        ConjuntoTDA resultado = new ConjuntoLD();

        while (!c1.ConjuntoVacio()){
            int elemento = c1.Elegir();
            if (!c2.Pertenece(elemento)){
                resultado.Agregar(elemento);
            }
            else{
                c2.Sacar(elemento);
            }
            c1.Sacar(elemento);
        }

        while (!c2.ConjuntoVacio()){
            int elemento = c2.Elegir();
```

```
            resultado.Agregar(elemento);
            c2.Sacar(elemento);
        }
        return resultado;
    }
```

Unidad 5

Arboles

Durante esta sección vamos a describir el concepto de la estructura de árboles y definiremos en particular el tipo de dato abstracto de los árboles binarios de búsqueda (ABB). Se presentarán los algoritmos de recorridos de árboles. Para la implementación y resolución de ejercicios con estas estructuras, será necesario introducir el tema de recursividad.

5.1 Conceptos

Un árbol representa una estructura jerárquica sobre una colección de elementos. Se puede definir como una colección de elementos llamados *nodos* sobre la que se define una relación de paternidad que impone una estructura jerárquica sobre estos nodos. Esta relación entre dos elementos define dos roles, un nodo es el *padre* y el otro es el *hijo*. En un árbol todo nodo tiene uno y sólo un padre, excepto un nodo particular al que se denomina la *raíz* del arbol y que no tiene ningún padre.
Más formalmente podemos decir que un árbol es vacío o es un nodo raíz y un conjunto de árboles que dependen de esa raíz. Cada uno de estos árboles, se denomina *subárbol*. Más adelante, veremos que esta definición es especialmente útil al momento de resolver problemas con árboles.
Se pueden definir más términos que se utilizarán con los árboles. Un nodo sin hijos se denomina *hoja*. Todos los nodos que no son ni la raíz ni las hojas se denominan *nodos internos*. Se denominan *descendientes* de un nodo al conjunto de elementos de todos sus subárboles. Se denominan *ancestros* de un nodo al conjunto de elementos formados por su padre, el padre de su padre y así siguiendo hasta llegar a la raíz del árbol.
La *profundidad* de un nodo es la longitud del camino único desde la raíz a ese nodo. La *altura* del árbol es la mayor de las profundidades de sus nodos.
Un solo nodo es por si mismo un árbol, y ese nodo será a su vez la raíz del árbol. También el árbol nulo, es decir, aquel que no contiene nodos es considerado un árbol.

Por ejemplo, una estructura de directorios en una PC puede ser representada por un árbol como se ve en la figura 5.1.

Figura 5.1: Estructura de árbol para un directorio

Siguiendo con la metodología de toda la materia, se trabajará con árboles de enteros. Por ejemplo, un árbol de enteros como el que se ve en la figura 5.2.

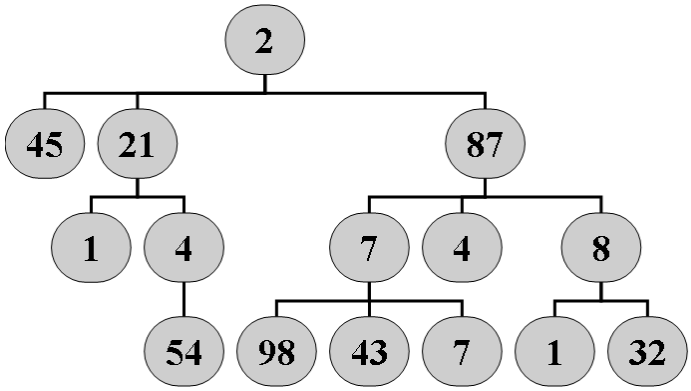

Figura 5.2: Arbol de enteros

En el árbol de ejemplo, 2 es la raíz. Las hojas son 45, 1, 54, 98, 43, 7, 1 y 32. Los descendientes de 21 son 1, 4, y 54. Los ancestros de 43 son 7, 87 y 2. La profundidad de 8 es 2. La altura del árbol es 3.

5.2 TDA Arbol

Para definir el TDA del árbol, utilizaremos los conceptos de *hijo mayor* y *hermano siguiente* que se pueden ver en la figura 5.3. Dado un nodo de un árbol, el hijo

mayor es el subárbol que está más a la izquierda. En cambio, el hermano siguiente de un nodo, es un subárbol del mismo padre que el nodo, que sigue a la derecha del mismo.

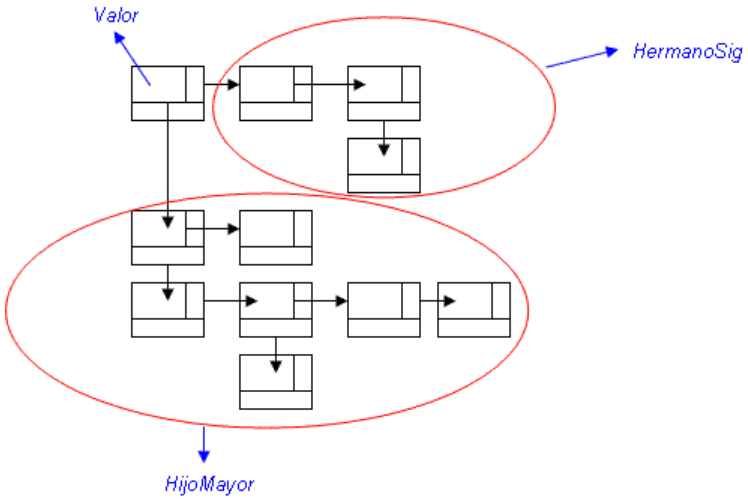

Figura 5.3: Arbol n-ario

De este TDA sólo veremos la especificación a modo ilustrativo. No veremos la implementación.
Las operaciones necesarias son:

- Valor: Recupera el valor de la raíz del árbol.
- HijoMayor: Recupera el subárbol del hijo más a la izquierda.
- HermanoSig: Recupera el subárbol del hermano siguiente.
- ArbolVacio: Indica si el árbol está vacío.
- InicializarArbol: Inicializa el árbol.
- ElimHijoMConDesc: Elimina el subárbol más a la izquierda del árbol dado con todos sus descendientes.
- ElimHermSConDesc: Elimina el hermano siguiente del árbol dado con todos sus descendientes.
- CrearArbol: Asigna el valor como raíz del árbol.
- AgregarHijoM: Agrega el valor dado como raíz del subárbol del árbol dado.

La especificación del TDA es la siguiente:

```
public interface ANaTDA {
    int Valor();
    ANaTDA HijoMayor();
    ANaTDA HermanoSig();
    boolean ArbolVacio();
    void InicializarArbol();
    void ElimHijoMConDesc(int x);
    void ElimHermSConDesc(int x);
    void CrearArbol(int r);
    void AgregarHijoM(int p, int h);
}
```

5.3 Arboles binarios

Un caso particular de árboles son los árboles binarios, los cuales cumplen con todas las definiciones dadas previamente, sólo que cada nodo puede tener a lo sumo dos hijos. Es decir cada nodo tiene cero, uno o dos hijos.
En estos árboles podemos hablar de hijo izquierdo, hijo derecho, subárbol izquierdo y subárbol derecho respectivamente.
Se puede ver un árbol binario en la figura 5.4.

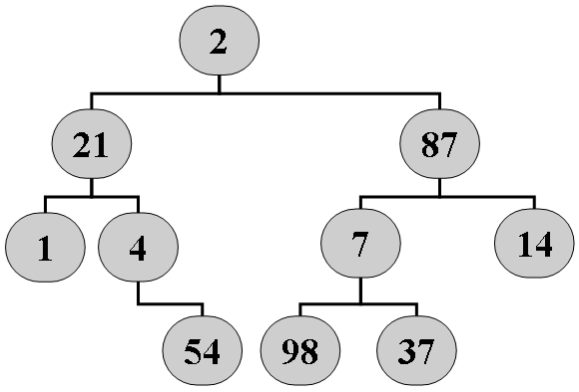

Figura 5.4: Arbol binario de enteros

5.4 Recursividad

Debido a la forma en que vamos a representar los árboles, se requiere para la mayoría de sus operaciones la utilización de operaciones recursivas. Vamos a recordar el concepto de recursividad.
Cuando hablamos de un método recursivo hablamos de un método que se define en función de sí mismo. Siempre nos vamos a encontrar que dentro de ese método

va a haber una llamada o invocación a sí mismo. Esto, a simple vista llevaría a un método que nunca acabaría su ejecución, por lo tanto, vamos a tener que buscar una forma de cortar esta recursión. Entonces, en todo método recursivo vamos a tener:

1. Un caso base o condición de corte, en donde se finalice la recursividad
2. Un paso recursivo, o invocación al método en sí mismo.

Debemos tener especial cuidado en que la ejecución del algoritmo alcance siempre la condición de corte, entonces cuando se invoca al método en forma recursiva se debe hacer con los parámetros que tiendan a alcanzar en un punto al caso base que corte la recursividad.
Un ejemplo simple de recursividad es el método que calcula el factorial de un número:

```
public int factorial(int n){

    if (n == 0){ //caso base
        return 1;
    }else{
        //Paso recursivo en donde se decrementa n para llegar al
           caso base
        return n * factorial(n-1);
    }
}
```

Como vemos, en este ejemplo, la llamada recursiva se hace con un número menos al ingresado, con lo cual, suponiendo que el número era positivo, en algún momento se va a alcanzar la condición de corte, cuando el número sea 0.

5.5 Árboles Binarios de Búsqueda (ABB)

Los árboles binarios de búsqueda (ABB) son árboles binarios que tienen la particularidad que sus elementos están ordenados de izquierda a derecha, es decir, dado un nodo X todos los nodos que pertenecen al subárbol izquierdo de ese nodo X tienen valores menores a él, y todos los nodos que son hijos derechos tienen valores mayores al nodo X. Por otra parte, un árbol ABB no tiene elementos repetidos.
Un ejemplo de ABB es el que se ve en la figura 5.5.

5.5.1 Especificación

Las operaciones necesarias para utilizar un ABB son:

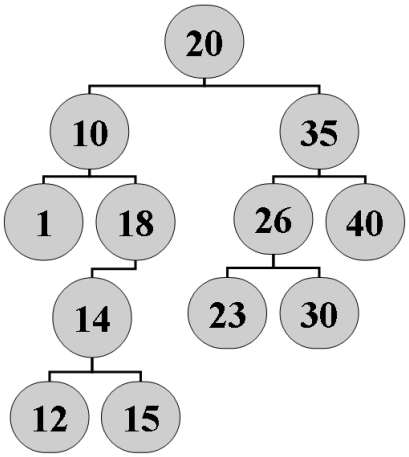

Figura 5.5: ABB

- Raiz: recupera el valor de la raíz de un ABB siempre que el mismo esté inicializado y no sea vacío.
- HijoIzq: devuelve el subárbol izquierdo siempre que el árbol esté inicializado y no esté vacío.
- HijoDer: devuelve el subárbol derecho siempre que el árbol esté inicializado y no esté vacío.
- ArbolVacio: indica si el árbol está vacío o no siempre que el árbol esté inicializado.
- InicializarABB: inicializa el árbol.
- AgregarElem: agrega el elemento dado manteniendo la propiedad de ABB, siempre que el árbol esté inicializado.
- EliminarElem: elimina el elemento dado manteniendo la propiedad de ABB, siempre que el árbol esté inicializado.

A continuación vamos a presentar la definición del TDA para los árboles ABB:

```
public interface ABBTDA {
    // siempre que el árbol esté inicializado y no esté vacío
    int Raiz();
    // siempre que el árbol esté inicializado y no esté vacío
    ABBTDA HijoIzq();
    // siempre que el árbol esté inicializado y no esté vacío
    ABBTDA HijoDer();
    // siempre que el árbol esté inicializado
    boolean ArbolVacio();
```

```
    void InicializarArbol();
    // siempre que el árbol esté inicializado
    void AgregarElem(int x);
    // siempre que el árbol esté inicializado
    void EliminarElem(int x);
}
```

5.5.2 Implementación

Existen diversas formas de implementar los ABB. Vamos a presentar una con estructura dinámica, pero se debe saber que existen otras implementaciones, incluso algunas con estructuras estáticas.
Para esta implementación, vamos a definir un nodo, como un valor y sus dos subárboles. El nodo queda como sigue:

```
class NodoABB{
    int info;
    ABBTDA hijoIzq;
    ABBTDA hijoDer;
}
```

Para la implementación, vamos a utilizar la última definición de árbol que dimos, adaptada a los ABB. La definición es que un árbol es un conjunto vacío de nodos o un nodo raíz y sus dos subárboles. Es decir, que para nosotros un árbol con un solo nodo, será un nodo y dos subárboles vacíos. Con esta definición, vamos a implementar el `ABBTDA`:

```
public class ABB implements ABBTDA{
    NodoABB raiz;

    public int Raiz(){
        return raiz.info;
    }

    public boolean ArbolVacio(){
        return (raiz == null);
    }

    public void InicializarArbol(){
        raiz = null;
    }

    public ABBTDA HijoDer(){
        return raiz.hijoDer;
    }

    public ABBTDA HijoIzq(){
        return raiz.hijoIzq;
```

```
    }

    public void AgregarElem(int x){
        if (raiz == null){
            raiz = new NodoABB();
            raiz.info = x;
            raiz.hijoIzq = new ABB();
            raiz.hijoIzq.InicializarArbol();
            raiz.hijoDer = new ABB();
            raiz.hijoDer.InicializarArbol();
        }
        else if (raiz.info > x )
            raiz.hijoIzq.AgregarElem(x);
        else if (raiz.info < x)
            raiz.hijoDer.AgregarElem(x);
    }

    public void EliminarElem(int x){
        if (raiz != null) {
            if (raiz.info == x && raiz.hijoIzq.ArbolVacio() &&
               raiz.hijoDer.ArbolVacio()) {
                raiz = null;
            }
            else if (raiz.info == x && !raiz.hijoIzq.ArbolVacio()
               ) {
                raiz.info = this.mayor(raiz.hijoIzq);
                raiz.hijoIzq.EliminarElem(raiz.info);
            }
            else if (raiz.info == x && raiz.hijoIzq.ArbolVacio())
                {
                raiz.info = this.menor(raiz.hijoDer);
                raiz.hijoDer.EliminarElem(raiz.info);
            }
            else if (raiz.info < x){
                raiz.hijoDer.EliminarElem(x);
            }
            else{
                raiz.hijoIzq.EliminarElem(x);
            }
        }
    }

    private int mayor(ABBTDA a){
        if (a.HijoDer().ArbolVacio())
            return a.Raiz();
        else
            return mayor(a.HijoDer());
    }

    private int menor(ABBTDA a){
        if (a.HijoIzq().ArbolVacio())
            return a.Raiz();
```

```
        else
            return menor(a.HijoIzq());
    }
}
```

5.5.3 Recorridos

Existen tres recorridos básicos de un árbol, que se conocen como pre-order, in-order y post-order. Los tres recorridos son recursivos por definición. Presentaremos la idea de cada uno y el algoritmo para resolverlo. Es importante tener presente qué realiza cada uno para poder utilizarlos ya que son la base para resolver muchos de los problemas con árboles.

Pre-order
Este recorrido propone visitar primero los padres y luego cada hijo en el mismo orden, es decir se visita el nodo raíz, y luego cada subárbol en pre-order. En esta definición, visitar puede significar mostrar o cualquier otra acción que se desee realizar. El algoritmo que hace este recorrido es el siguiente:

```
public void preOrder(ABBTDA a){
    if(!a.ArbolVacio()){
        System.out.println(a.raiz());
        preOrder(a.HijoIzq());
        preOrder(a.HijoDer());
    }
}
```

El resultado de recorrer en pre-order el árbol de ejemplo de la figura 5.5 sería: 20, 10, 1, 18, 14, 12, 15, 35, 26, 23, 30, 40.

In-order
Este recorrido propone visitar in-order el subárbol izquierdo, luego la raíz y por último in-order el subárbol derecho. En esta definición, visitar puede significar mostrar o cualquier otra acción que se desee realizar. Este recorrido es el que en un ABB muestra los elementos ordenados de menor a mayor. El algoritmo que hace este recorrido es el siguiente:

```
public void inOrder(ABBTDA a){
    if(!a.ArbolVacio()){
        inOrder(a.HijoIzq());
        System.out.println(a.raiz());
        inOrder(a.HijoDer());
    }
}
```

El resultado de recorrer in-order el árbol de ejemplo de la figura 5.5 sería: 1, 10, 12, 14, 15, 18, 20, 23, 26, 30, 35, 40.

Post-order
Este recorrido propone visitar primero los hijos en post-orden y luego la raiz. En esta definición, visitar puede significar mostrar o cualquier otra acción que se desee realizar. El algoritmo que hace este recorrido es el siguiente:

```
public void postOrder(ABBTDA a){
    if(!a.ArbolVacio()){
        postOrder(a.HijoIzq());
        postOrder(a.HijoDer());
        System.out.println(a.raiz());
    }
}
```

El resultado de recorrer en post-order el árbol de ejemplo de la figura 5.5 sería: 1, 12, 15, 14, 18, 10, 23, 30, 26, 40, 35, 20.

5.5.4 Utilización

Vamos a resolver algunos ejercicios en donde se trabaja con árboles binarios de búsqueda.

Ejercicio 1: Contar la cantidad de nodos de un ABB.

```
public int Contar(ABBTDA a){
    if(a.ArbolVacio()){
        return 0;
    }
    else{
        return(1 + Contar(a.HijoIzq()) + Contar(a.HijoDer()));
    }
}
```

Ejercicio 2: Dado un elemento y un ABB, calcular su profundidad en el árbol.

```
    public int calcularProfundidad(ABBTDA t, int x){
        if (t.ArbolVacio()){
            return 0;
        }
        else if (t.Raiz() == x){
            return 0;
        }
        else if (t.Raiz() > x){
            return (1+this.calcularProfundidad(t.HijoIzq(), x));
        }
```

```
        else{
            return (1+this.calcularProfundidad(t.HijoDer(), x));
        }
    }
```

Ejercicio 3: Dado un elemento, determinar si está o no en un ABB.

```
public boolean existeElementoEnABB(ABBTDA t, int x){
    if (t.ArbolVacio()){
        return false;
    }
    else if (t.Raiz() == x){
        return true;
    }
    else if (t.Raiz() > x){
        return this.existeElementoEnABB(t.HijoIzq(), x);
    }
    else{
        return this.existeElementoEnABB(t.HijoDer(), x);
    }
}
```

Ejercicio 4: Dado un árbol ABBTDA a, devolver todos los nodos pares del mismo.

```
    public ConjuntoTDA nodosPares(ABBTDA a){

        ConjuntoTDA r= new ConjuntoLD();
        r.InicializarConjunto();

        if (!a.ArbolVacio()){
            if (a.raiz() % 2 == 0){
                r.Agregar(a.raiz());
            }
            ConjuntoTDA rI = nodosPares(a.HijoIzq());
            ConjuntoTDA rD = nodosPares(a.HijoDer());
            while (!rI.ConjuntoVacio()){
                int x = rI.Elegir();
                r.Agregar(x);
                rI.Sacar(x);
            }
            while (!rD.ConjuntoVacio()){
                int x = rD.Elegir();
                r.Agregar(x);
                rD.Sacar(x);
            }
        }
        return r;
    }
```

Unidad 6

Grafos

En esta unidad se introducirán los conceptos principales de grafos. Se especificará el TDA grafo y sus posibles implementaciones.

6.1 Conceptos básicos

Un grafo es un conjunto de de elementos llamados *nodos* o *vértices* y un conjunto de pares de nodos llamados *aristas*. Cada arista tiene un valor entero asociado al que llamaremos *peso*. Un grafo sirve para representar una red. Un ejemplo podría ser una red de computadoras (figura 6.1) o un mapa de rutas que unen ciudades(figura 6.2).
Un caso particular de grafos, son aquellos en los que las aristas tienen un sentido y no son de doble sentido, y se denominan *grafos dirigidos*. En ese caso, la arista tiene un *nodo origen* y un *nodo destino*.
Se dice que un vértice tiene *aristas entrantes*, cuando existe al menos una arista que tiene al vértice en cuestión como destino. Se dice que un vértice tiene *aristas salientes*, cuando existe al menos una arista que tiene al vértice en cuestión como origen. Un vértice que no tiene aristas entrantes ni salientes es un *vértice aislado*.
La cantidad de aristas salientes se llama *grado positivo* de un vértice. La cantidad de aristas entrantes se llama *grado negativo* de un vértice.
Se dice que un vértice `b` es *adyacente* de un vértice a, si existe una arista entre a y b.
Un *camino*, es una sucesión ordenada de nodos $v_1, v_2, v_3, \cdots, v_n$ en donde existe una arista entre v_1 y v_2, otra entre v_2 y v_3, y así siguiendo hasta v_n. El *costo* del camino es la suma de los pesos de cada arista.
Un *ciclo* es un camino que comienza y termina en un mismo vértice. Es de especial importancia tener en cuenta este concepto, ya que al momento de comenzar a recorrer los grafos si nos econtramos con un ciclo podríamos entrar en un camino sin salida si no tenemos en cuenta esta situación y la tratamos.
Se dice que un vértice es *alcanzable* desde otro si existe un camino que una al segundo con el primero.

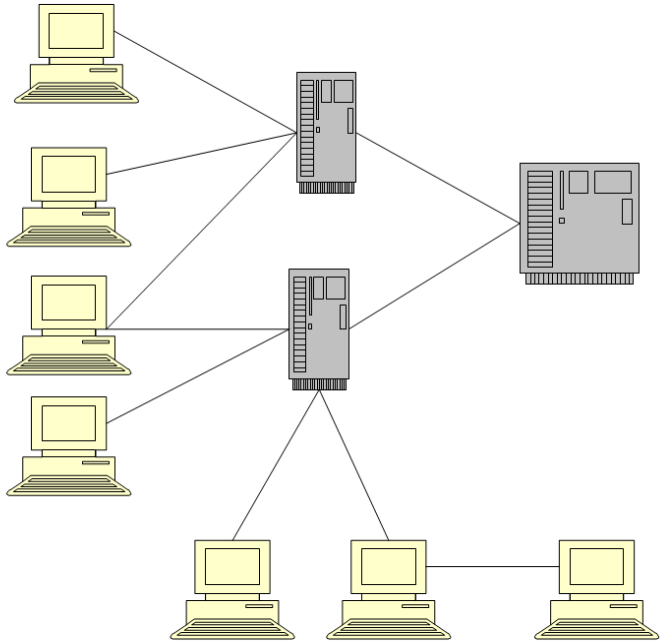

Figura 6.1: Red de computadores

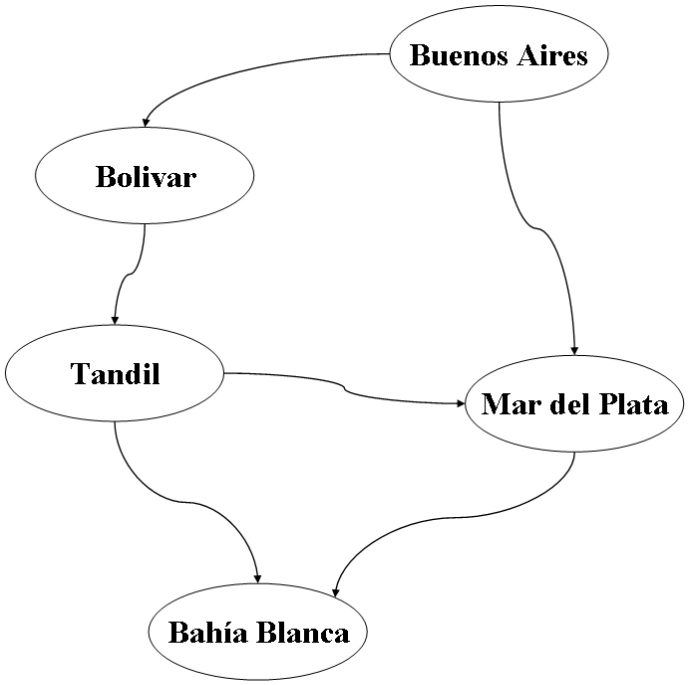

Figura 6.2: Mapa de ciudades

Se entiende por *aristas paralelas*, a un par de aristas con mismo origen y mismo destino. Se define como *bucle*, una arista que tiene igual origen y destino.

A los efectos del curso, vamos a trabajar con: grafos dirigidos, sin aristas paralelas ni bucles, en donde los nodos son números enteros, los pesos son enteros positivos.

Dado el grafo de la figura 6.3 los vértices son 1, 2, 3, 4, 5, 6, 7, 8 y 10. Las aristas son (1,2), (2,1), (1,3), (3,4), (3,5), (4,6), (5,6), (6,5) y (8,10). El peso de cada una es 12, 10, 21, 32, 9, 10, 10, 12, 87 y 10 respectivamente. El nodo 3 tiene una arista entrante y dos salientes. El nodo 7 no tiene ninguna arista entrante ni saliente, por ende es un nodo aislado. Existe un camino entre el nodo 1 y el 4, su costo es 53. Además el nodo 4 es alcanzable desde 1.

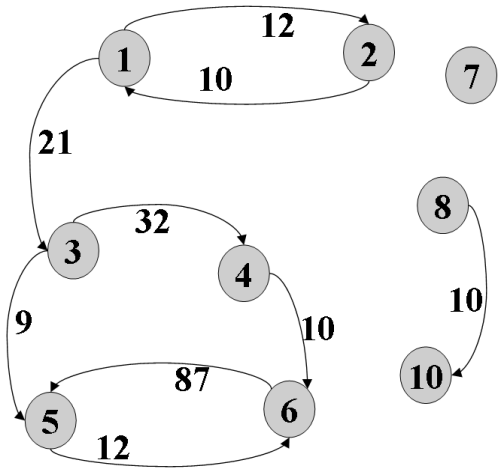

Figura 6.3: Ejemplo de grafo

6.2 Especificación del TDA

Para utilizar un grafo, definiremos las siguientes operaciones:

- InicializarGrafo: inicializa el grafo.
- AgregarVertice: agrega un nuevo vértice al grafo, siempre que el grafo esté inicializado y el vértice no exista.
- EliminarVertice: elimina el vértice dado siempre que el vértice exista.
- Vertices: devuelve el conjunto de vértices de un grafo, siempre que el grafo esté inicializado.

- AgregarArista: agrega una arista al grafo entre los vértices dados y con el peso dado, siempre que existan ambos vértices pero no exista una arista entre ambos.
- EliminarArista: elimina la arista entre los vértices dados, siempre que esta arista exista.
- ExisteArista: indica si el grafo contiene una arista entre los vértices dados, siempre que los vértices existan.
- PesoArista: devuelve el peso de la arista entre los vértices dados, siempre que la arista exista.

Entonces queda la siguiente especificación del GrafoTDA:

```
public interface GrafoTDA {
    void InicializarGrafo();
    // siempre que el grafo esté inicializado y no exista el nodo
    void AgregarVertice(int v);
    // siempre que el grafo esté inicializado y exista el nodo
    void EliminarVertice(int v);
    // siempre que el grafo esté inicializado
    ConjuntoTDA Vertices();
    // siempre que el grafo esté inicializado, no exista la
       arista, pero existan ambos nodos
    void AgregarArista(int v1, int v2, int peso);
    // siempre que el grafo esté inicializado y exista la arista
    void EliminarArista(int v1, int v2);
    // siempre que el grafo esté inicializado y existan los nodos
    boolean ExisteArista(int v1, int v2);
    // siempre que el grafo esté inicializado y exista la arista
    int PesoArista(int v1, int v2);
}
```

6.3 Implementaciones estáticas y dinámicas

Vamos a proponer dos implementaciones del TDA, una estática y otra dinámica. La implementación estática es una implementación con *matrices de adyacencia*. Una matriz de adyacencia, es una matriz cuadrada donde la cantidad de filas y columnas está dada por la cantidad de nodos del grafo. Además el contenido de cada celda es el peso de la arista con origen en el número de fila y destino en el número de columna.
Ahora bien, en *Java*, los índices de las matrices van de cero en adelante, en forma consecutiva. No es el caso de los nodos, que pueden tener valores no contínuos. Por lo tanto, vamos a tener que utilizar un mapeo que nos indique en qué índice

de la matriz está el nodo que buscamos. Entonces, al nombre del nodo lo vamos a llamar internamente etiqueta.
Por ejemplo para el grafo dado de la figura 6.3, la matriz sería la que se ve en la figura 6.4 y el mapeo entre etiquetas e índices se puede ver en la figura 6.5

	0	1	2	3	4	5	6	7	8
0	0	12	21	0	0	0	0	0	0
1	10	0	0	0	0	0	0	0	0
2	0	0	0	32	9	0	0	0	0
3	0	0	0	0	0	10	0	0	0
4	0	0	0	0	0	12	0	0	0
5	0	0	0	0	87	0	0	0	0
6	0	0	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	10
8	0	0	0	0	0	0	0	0	0

Figura 6.4: Matriz de adyacencia

1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**10**
0	1	2	3	4	5	6	7	8

Figura 6.5: Etiquetas de mapeo entre vértices e índices de la matriz

```
public class GrafoMA implements GrafoTDA {

    static int n = 100;
    int [][] MAdy;
    int [] Etiqs;
    int cantNodos;

    public void InicializarGrafo() {
        MAdy = new int[n][n];
        Etiqs = new int[n];
        cantNodos = 0;
    }

    public void AgregarVertice(int v) {
        Etiqs[cantNodos] = v;
        for(int i=0; i<=cantNodos; i++){
```

```
            MAdy[cantNodos][i] = 0;
            MAdy[i][cantNodos] = 0;
        }
        cantNodos++;
    }

    public void EliminarVertice(int v) {
        int ind = Vert2Indice(v);

        for(int k=0; k<cantNodos; k++)
            MAdy[k][ind] = MAdy[k][cantNodos-1];
        for(int k=0; k<cantNodos; k++)
            MAdy[ind][k] = MAdy[cantNodos-1][k];

        Etiqs[ind] = Etiqs[cantNodos-1];
        cantNodos--;
    }

    private int Vert2Indice(int v){
        int  i=cantNodos-1;
        while(i>=0 && Etiqs[i]!=v)
            i--;
        return i;
    }

    public ConjuntoTDA Vertices() {
        ConjuntoTDA Vert=new ConjuntoLD();
        Vert.InicializarConjunto();

        for(int i=0; i<cantNodos; i++){
            Vert.Agregar(Etiqs[i]);
        }
        return Vert;
    }

    public void AgregarArista(int v1, int v2, int peso) {
        int o = Vert2Indice(v1);
        int d = Vert2Indice(v2);
        MAdy[o][d] = peso;
    }

    public void EliminarArista(int v1, int v2) {
        int o = Vert2Indice(v1);
        int d = Vert2Indice(v2);
        MAdy[o][d]=0;
    }

    public boolean ExisteArista(int v1, int v2) {
        int o = Vert2Indice(v1);
        int d = Vert2Indice(v2);
        return MAdy[o][d]!=0;
    }
```

```
    public int PesoArista(int v1, int v2) {
        int o = Vert2Indice(v1);
        int d = Vert2Indice(v2);
        return MAdy[o][d];
    }
}
```

Para el caso de la implementación dinámica, llamaremos a la implementación *lista de adyacencia*. La idea es tener una lista encadenada de nodos (a los que llamaremos `NodoGrafo`). Cada nodo tiene su valor, una referencia al próximo nodo y además una referencia a la lista de aristas que tienen a ese nodo como origen. Una arista (a la que llamaremos `NodoArista`), tiene una referencia al nodo destino de esta arista, el peso y una referencia a la próxima arista de esa lista.
Entonces para el grafo del ejemplo de la figura 6.3 tenemos la la representación de la figura 6.6 donde los círculos son los `NodoGrafo` y los rectángulos los `NodoArista`.

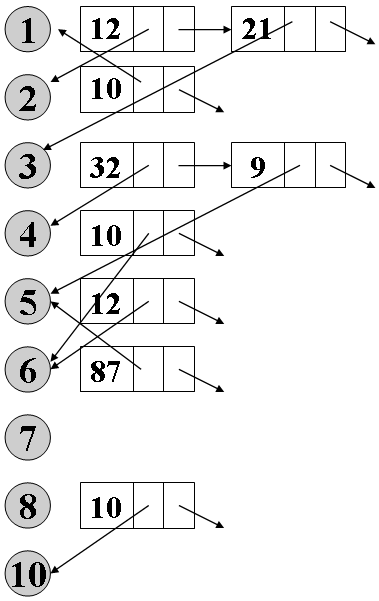

Figura 6.6: Lista de adyacencia

La representación de los nodos y las aristas es la siguiente:

```
class NodoGrafo{
    int nodo;
    NodoArista arista;
    NodoGrafo sigNodo;
}
```

```
class NodoArista{
    int etiqueta;
    NodoGrafo nodoDestino;
    NodoArista sigArista;
}
```

Entonces, la implementación con listas de adyacencia es la siguiente:

```
public class GrafoLA implements GrafoTDA{

    NodoGrafo origen;

    public void InicializarGrafo(){
       origen = null;
    }

    public void AgregarVertice(int v){
        //El vertice se inserta al inicio de la lista de nodos
        NodoGrafo aux = new NodoGrafo();
        aux.nodo = v;
        aux.arista = null;
        aux.sigNodo = origen;
        origen = aux;
    }

    /*
     * Para agregar una nueva arista al grafo, primero se deben
     * buscar los nodos entre los cuales se va agregar la arista,
     * y luego se inserta sobre la lista de adyacentes del nodo
     * origen (en este caso nombrado como v1)
     */
    public void AgregarArista(int v1, int v2, int peso){
        NodoGrafo n1 = Vert2Nodo(v1);
        NodoGrafo n2 = Vert2Nodo(v2);

        //La nueva arista se inserta al inicio de la lista
        //de nodos adyacentes del nodo origen
        NodoArista aux = new NodoArista();
        aux.etiqueta = peso;
        aux.nodoDestino = n2;
        aux.sigArista = n1.arista;
        n1.arista = aux;
    }

    private NodoGrafo Vert2Nodo(int v){
        NodoGrafo aux = origen;
        while (aux!=null && aux.nodo!=v){
            aux = aux.sigNodo;
        }
        return aux;
    }
```

```
public void EliminarVertice(int v){
    //Se recorre la lista de vértices para remover el nodo v
    //y las aristas con este vértice.

    //Distingue el caso que sea el primer nodo
    if(origen.nodo == v) {
       origen = origen.sigNodo;
    }

    NodoGrafo aux = origen;
    while (aux != null){
        //remueve de aux todas las aristas hacia v
        this.EliminarAristaNodo(aux, v);

        if (aux.sigNodo!=null && aux.sigNodo.nodo == v) {
            //Si el siguiente nodo de aux es v, lo elimina
            aux.sigNodo = aux.sigNodo.sigNodo;
        }
        aux = aux.sigNodo;
    }
}

/*
 * Si en las aristas del nodo existe
 * una arista hacia v, la elimina
 */
private void EliminarAristaNodo(NodoGrafo nodo, int v){
    NodoArista aux = nodo.arista;

    if (aux!=null) {
        //Si la arista a eliminar es la primera en
        //la lista de nodos adyacentes
        if (aux.nodoDestino.nodo == v){
            nodo.arista = aux.sigArista;
        }
        else {
            while (aux.sigArista!=null && aux.sigArista.
               nodoDestino.nodo != v){
                aux = aux.sigArista;
            }
            if(aux.sigArista!=null) {
                //Quita la referencia a la arista hacia v
                aux.sigArista = aux.sigArista.sigArista;
            }
        }
    }
}

public ConjuntoTDA Vertices(){
    ConjuntoTDA c = new ConjuntoLD();
    c.InicializarConjunto();
```

```
        NodoGrafo aux = origen;
        while (aux != null){
            c.Agregar(aux.nodo);
            aux = aux.sigNodo;
        }
        return c;
    }

    /*
     * Se elimina la arista que tiene como origen al vértice v1
     * y destino al vértice v2
     */
    public void EliminarArista(int v1, int v2){
        NodoGrafo n1 = Vert2Nodo(v1);
        EliminarAristaNodo(n1,v2);
    }

    public boolean ExisteArista(int v1, int v2){
        NodoGrafo n1 = Vert2Nodo(v1);

        NodoArista aux = n1.arista;
        while (aux!=null && aux.nodoDestino.nodo!=v2){
            aux = aux.sigArista;
        }
        //Solo si se encontro la arista buscada, aux no es null
        return aux!=null;
    }

    public int PesoArista(int v1, int v2){
        NodoGrafo n1 = Vert2Nodo(v1);

        NodoArista aux = n1.arista;
        while (aux.nodoDestino.nodo != v2){
            aux = aux.sigArista;
        }
        //Se encontró la arista entre los dos nodos
        return aux.etiqueta;
    }
}
```

Bibliografía

[AA98] J. Hopcroft Y J. Ullman A. Aho. *Estructuras de Datos y Algoritmos*. Addison Wesley, Ubicación en biblioteca UADE: 681.3.01/A292, 1998.

[Wei04] Mark Allen Weiss. *Estructuras de datos en JAVA*. Addison Wesley, Ubicación en biblioteca UADE: 004.43 WEI est [1a] 2004, 2004.

[yJC06] J. Lewis y J. Chase. *Estructuras de Datos con Java. Diseño de Estructuras y Algoritmos*. Addison Wesley Iberoamericana, 2006.

[yRT02] M. Goodrich y R. Tamassia. *Estructura de Datos y Algoritmos en Java*. Editorial Continental, 2002.