GRADO EN INGENIERÍA INFORMÁTICA

Módulo de Formación Obligatoria

ESTRUCTURA DE DATOS Y ALGORITMOS

Dr. Oscar Antonio Palma Gamboa

viu Universidad Internacional de Valencia

Edita
Universidad Internacional de Valencia

Grado en

Ingeniería Informática

Estructura de datos y algoritmos

Módulo de Formación Obligatoria

6ECTS

Dr. Oscar Antonio Palma Gamboa

Índice

Leyenda

abc **Glosario**
Términos cuya definición correspondiente está en el apartado "Glosario".

Enlace de interés
Dirección de página web.

Tema 1.
Introducción: Estructuras de Datos y Algoritmos en el entorno *Java*

Este tema incorpora un repaso de las principales estructuras del entorno de programación *JAVA*, así como una introducción al análisis de algoritmos. Deberá de ponerse atención en los conceptos y ejemplos planteados para poder resolver los ejercicios de manera satisfactoria.

1.1 Estructuras Primitivas

1.1.1 Comprobando la versión de *Java*

Para iniciar con los contenidos del temario es necesario haber **instalado *Java*** en la computadora. A través del acceso a la dirección se podrá verificar si existe alguna versión instalada y la que actualmente está corriendo. En caso de existir una actualización se ofrece la alternativa de instalarla. Es importante que las versiones antiguas de *Java* sean **desinstaladas** con el propósito de no entrar en conflicto entre versiones.

Enlace 1

Verificación de *Java* Script

http://www.*java*.com/es/download/installed.jsp

Para verificar si la instalación ha sido correcta debe revisarse que existan dos carpetas en una ruta similar a "C:\Archivos de programa*Java*" ("C:\Program Files*Java*"), las carpetas tienen un nombre similar a *"jdk-10"* y *"jre-10"*. La primera corresponde al **compilador** e intérprete *Java*. La segunda carpeta, de nombre similar a *"jre-10",* incluye la **máquina virtual** *Java*. Si las carpetas existen, se ha finalizado con la instalación de *Java* en Windows.

Para usar *java* se debe configurar algunas **variables** de entorno, en este caso serán "*JAVA_HOME"* y *"PATH"*.

JAVA_HOME, es una **variable de entorno** del sistema que informa al sistema operativo sobre la ruta donde se encuentra instalado *Java*, por lo tanto, el valor de la variable deberá contener la ruta en hacia el archivo *"jdk"*.

PATH es una variable de entorno del sistema que permite al sistema operativo saber la ruta de distintos directorios esenciales para el funcionamiento de la computadora. Se requiere añadir en el contenido de la variable PATH el lugar donde se encuentran los archivos ejecutables de *Java* necesarios para su **ejecución**, como el compilador (*javac*.exe) y el intérprete (*java*.exe).

Para verificar la instalación escribiremos el siguiente programa en un **block de notas:**

```
/* Ejemplo Hola Mundo */
public class HolaMundo {
     public static void main(String[ ] arg) {
     System.out.println("Hola Mundo");
     }
}
```

Una vez guardado, se debe de ejecutar la **consola *"command"*** del sistema operativo y dirigirse a la carpeta en donde haya guardado el programa que acaba de ser creado. Una vez ahí, el programa debe ser compilado con la instrucción *javac HolaMundo.java*. Si no existen errores se puede continuar con la ejecución del programa a través de la instrucción *java* HolaMundo. Si es posible visualizar el resultado en pantalla, entonces la instalación ha sido exitosa.

1.1.2 Tipos primitivos, constantes, ciclos y métodos

Los **tipos primitivos en** *Java* **son 8:** *byte, short, int, long, float, double, char, boolean*. Estos representan ya sea un número, un *caracter* o un booleano (falso o verdadero) (Weiss,2013).

Las **constantes** pueden definirse en cualquiera de los siguientes sistemas de numeración: octal (0), hexadecimal (0xó 0X) o decimal. Las constantes usadas para caracteres deben escribirse entre una

pareja de comillas simples ('b'), una constante de cadena es una secuencia de caracteres encerrados por comillas cerradas ("Hola"). Existen secuencias de escape para representar tabulaciones, saltos de línea, etc.

Las variables deben declararse usando un tipo primitivo, además de su nombre, y opcionalmente un valor inicial. El nombre se conoce como identificador y puede estar compuesto de cualquier combinación de números, letras y caracteres; sin embargo, no pueden iniciar con un digito. *Java* hace diferencia entre mayúsculas y minúsculas por lo que pueden diferenciarse dos identificadores con el mismo nombre al hacer uso de una combinación de mayúsculas y minúsculas.

En cuanto a los **operadores** básicos, *Java* dispone de:

```
Operadores de asignación: =, +=, -=, *=, y /=
Operadores aritméticos binarios: +, -, *,/, y %
Operadores aritméticos unarios: -x, ++x, x++, --x, x—
```

La instrucción *if,* permite al programa ejecutar alguna parte del código mediante la **toma de decisiones,** es decir, cuando la expresión de evaluación es cierta (TRUE) entonces se ejecuta una serie de instrucciones, si la expresión es falsa (FALSE), se ejecuta un segundo bloque de instrucciones. Para definir estos dos bloques de instrucciones se utiliza la combinación *if-else,* esta misma combinación se puede utilizar de forma tal que aparece en **bloques anidados.**

```
if (expresión a evaluar)
     Ejecuta si es cierta ->Instrucción
Ejecuta de ser falsa
…

If (expresión a evaluar)
     {
     Bloque de instrucciones a ejecutar si es verdadera
     }
else
     {
     Bloque de instrucciones a ejecutar si es falsa
     }
```

Para el uso de los ciclos en *Java* deben definirse operadores relacionales y lógicos:

```
Operadores relacionales: ==, !=, <, <=, >=
Operadores lógicos: &&, ||,!
```

Los ciclos definidos en *Java* son**: *while, for, do.*** La mayoría de las situaciones que deben resolverse mediante ciclos pueden hacerse mediante la instrucción *while*, sin embargo, la solución de situaciones específicas puede hacerse más entendible mediante las otras dos instrucciones (Weiss,2013).

El **ciclo *while*** es utilizado para ejecutar un bloque de instrucciones mientras una expresión sea cierta:

```
while (expresión)
      {
           Instrucciones
      }
instrucción fuera del ciclo
```

El **ciclo *for*** se utiliza para realizar iteraciones ya establecidas, como en un contador.

```
for(inicialización; comprobación; actualización)
      {
           Instrucciones a iterar
      }
```

El campo de inicialización permite establecer el valor inicial de la variable que estará fungiendo como "contador". El campo de comprobación, verifica si el conteo ha llegado a un número determinado de iteraciones, mientras que el campo de actualización realiza el incremento o decremento del contador.

El **ciclo *do*,** permite ejecutar el bloque de instrucciones por lo menos una vez. Esto es debido a que la comprobación de ciclo se hace hasta el final de la ejecución del bloque de instrucciones.

```
do
      {
              Bloque de instrucciones
      } while (expresión de evaluación)
```

Todos los elementos que se han citado hasta el momento son integrados en funciones o procedimientos, como se les llamaba en la programación estructurada. En *Java,* que es un lenguaje para programación orientada a objetos el término para estos elementos es el **método.**

Un método consta de:

Una **cabecera de método:** contiene un nombre, una lista de parámetros y un valor de retorno.

Un **cuerpo del método:** bloque de instrucciones para ejecutar una tarea.

1.2 Tipos de referencia

Los tipos primitivos descritos en la sección anterior son los básicos dentro del entorno de *java*, todos los que son distintos a estos se conocen como tipos de **referencia**, como por ejemplo *las cadenas, las matrices y los flujos de archivos.*

1.2.1 Conceptos básicos

Una **referencia es una variable que almacena una dirección de memoria** en la que reside un objeto. En el entorno de *java*, los objetos son instancias que integran los tipos primitivos pero también variables de referencia. Las variables de referencia también pueden ser sujetas a operaciones.

Para declarar un objeto (variable de referencia) simplemente proporcionamos un nombre, aunque esta **declaración no crea** el objeto. Cuando no se crea el objeto no puede hacerse referencia a algún espacio de memoria, por lo que el valor almacenado en la variable es ***null.*** La única forma comúnmente empleada para crear un objeto es a través del uso de la **palabra clave new.** En *java*, cuando un objeto que ha sido construido ya no está referenciado por ninguna variable de objeto, el sistema reclama la memoria anteriormente ocupada automáticamente. Entre variables de referencia debe tenerse en cuenta que el signo = no relaciona valores como en los tipos primitivos, sino que relaciona valores de memoria, y como se mencionó anteriormente, estos apuntan hacia los objetos (Weiss,2013).

Para comparar dos variables de referencia se utiliza el operador ==, el resultado de este será verdadero si hace referencia al mismo objeto almacenado, o si ambos son *null.*

1.2.2 Cadenas

En *java* las cadenas son variables de referencia tipo *string,* aunque *java* las hace parecer como estructuras primitivas. Las reglas fundamentales para el manejo de este tipo de objetos son:

1. El manejo debe hacerse teniendo en cuenta que son objetos.
2. Un *objeto string* es inmutable. Es decir, una vez construido el objeto, su contenido no puede modificarse.

Para **concatenar** cadenas pueden utilizarse los operandos +, y +=, aunque no sean tipos primitivos. Para comparar la igualdad de dos objetos *string* se usa el método *equals*, aunque se puede llevar a cabo una comprobación más general mediante el método *compareTo*. Para obtener la longitud de una cadena de caracteres se puede emplear el método *lenght* (una cadena vacía tiene longitud cero). Con el método *charAt* se puede obtener un carácter único en una cadena especificando la posición (la posición cero es la primera). Con el método *substring* se puede obtener la referencia de un objeto *string* cuando es de nueva construcción.

Con la concatenación de cadenas se puede convertir un valor primitivo en un objeto string, sin embargo existen métodos para hacer esto directamente. La operación descrita anteriormente puede hacerse mediante el método *toString*. Para obtener la conversión de un valor entero (int) se puede invocar al método *Integer.parseInt*. Se puede aplicar un concepto similar para un valor de tipo double *(Double.parseDouble)* (Weiss,2013).

1.2.3 Matrices

Una matriz almacena una **colección de entidades de tipo idéntico.** La matriz no es un tipo primitivo en *java*, por lo que su comportamiento es el de un objeto. Se puede tener acceso a cada entidad de la matriz mediante el operador de **indexación** de matriz [ ]. Este operador especifica a qué objeto hay que acceder. Las matrices se indexan empezando por el cero y puede obtenerse el número de elementos que pueden almacenar mediante *a.length*. La expansión dinámica de matrices permite construir matrices de tamaño arbitrario y hacerlas más grandes en caso necesario. Existe también un tipo de matriz, llamado matriz multidimensional. A este último tipo de objeto se accede mediante más de un **índice** (Weiss,2013).

1.2.4 *ArrayList*

Dado que la técnica de expansión dinámica es muy común, la librería de *java* contiene un tipo *ArrayList*. La idea es que se mantenga no solo un tamaño sino la cantidad de memoria reservada. El método *add* incrementa el tamaño en una unidad y añade a la matriz un nuevo elemento en la posición apropiada. Los objetos *ArrayList* se inicializan con un tamaño igual a cero. La indexación mediante *[ ]* está reservada solo para las matrices, por lo que para acceder a los elementos de este objeto se puede emplear el método *get*. Este método devuelve el objeto en el índice concreto y empleando el método set puede modificarse el valor de una referencia en un índice determinado. Es importante mencionar que a un *ArrayList* solo pueden añadirse objetos.

1.2.5 Tratamiento de excepciones

Las *excepciones* se utilizan para manejar situaciones excepcionales como, por ejemplo, los errores. Para manejar las excepciones se puede encerrar en un bloque *try* el bloque de instrucciones que podrían generar la excepción. Después de este bloque se define un bloque *catch* que procesa la excepción. Algunos objetos dentro de un bloque *try* podrán requerir tareas de limpieza, en este caso se utiliza la cláusula *finally*, que puede incluirse después del último bloque *catch* (o del bloque *try* si no ha bloques *catch*). La cláusula *finally* está compuesta por la palabra clave y seguida del bloque correspondiente. Existen tres casos básicos:

1. Si *try* se ejecuta sin excepción, el control pasa al bloque *finally*.
2. Si una excepción no es capturada dentro del bloque *try*, el control pasa al bloque *finally*. Después, tras ejecutar el bloque *finally*, la excepción se propaga.
3. Si una excepción es capturada en el bloque *try*, el control pasa al bloque *catch* apropiado. Después, tras ejecutar el bloque *catch*, se ejecuta el bloque *finally* (Weiss, 2013).

En *java* existen tipos de excepciones estándar. Estas excepciones en tiempo de ejecución incluyen sucesos tales como la **división entera por cero y el acceso ilegal a una matriz.** Los errores son excepciones no recuperables, es decir, problemas de la máquina virtual. El error más común es *OutOfMemoryError*. Otros errores comunes son *InternalError* y *UnknowError*, en este último la máquina

virtual tiene problemas y no quiere continuar. Mediante las cláusulas *throw* y *throws* el programador puede generar una excepción.

1.2.6 Entrada y salida

Para la entrada y salida *java* utiliza el **paquete** *java.io*. La **directiva** *import* permite no tener que utilizar nombres completos. Esto es similar al paquete *java.lang* utilizado en tipos *String* y *Math*. Para realizar la entrada o salida hacia o desde el terminal, un archivo o a través de Internet, el programador crea un grupo de datos asociado.

Existen tres grupos predefinidos para la entrada/salida de terminal:

1. *System.in,* entrada estándar;
2. *System.out,* la salida estándar; y
3. *System.err,* la salida estándar de error (Weiss,2013).

La salida en *java* se realiza casi exclusivamente mediante la concatenación de objetos *String* sin ningún formateo predefinido.

El método más simple de leer una entrada con formato consiste en utilizar un *Scanner*. Un *Scanner* permite al usuario leer líneas de una en una mediante *nextLine*, leer objetos *String* de uno en uno utilizando *next* o leer tipos primitivos de uno en uno utilizando métodos como *nextInt* y *nextDouble*. Cuando se emplea un *Scanner* es costumbre proporcionar la directiva de importación import *java.util.Scanner.*

Una regla básica de *java* es que lo que funciona para la entrada/salida a través de una terminal, **también funciona para los archivos.** Para manejar un archivo, construimos un objeto *FileReader*. Cuando terminemos con el archivo debemos cerrarlo con la instrucción *Close*.

1.3 Objetos y clases

La programación orientada a objetos surgió como **paradigma** a mediados de la década de los 90's. En el núcleo se encuentra lo que se conoce como **objeto**, es decir, un tipo de dato que tiene una **estructura y un estado**, además de operaciones que permiten acceder a él o manipularlo. Lo consideramos como una unidad atómica pues sus partes no pueden ser diseccionadas. A esto se le conoce con el nombre de **ocultación de la información,** porque el usuario no dispone de acceso directo al objeto ni a sus implementaciones; solo se puede acceder a él indirectamente mediante **métodos suministrados junto con el propio objeto** (Weiss,2013).

La *encapsulación* es la agrupación de datos y de operaciones que se aplican a los objetos, para formar un agregado, que al mismo tiempo se oculta.

Un objetivo de la programación orientada a objetos es la reutilización de los mismos, a través de mecanismos como por ejemplo el uso de código genérico. El mecanismo de herencia permite ampliar la funcionalidad de un objeto, es decir, que podemos crear nuevos tipos a partir del tipo original.

El *polimorfismo* es otro principio de la orientación a objetos. Este concepto establece que un tipo de referencia puede hacerlo hacia varios tipos de objetos distintos. Esto es parte del mecanismo de herencia en *java*.

Se puede definir en *java* al objeto como una instancia de una clase. Cada clase está compuesta por campos que almacenan datos y por métodos que se aplican a las instancias de esa clase.

Por ejemplo, el código que a continuación se muestra permite definir algunos conceptos:

```
// clase IntCell
// int read ( ) →Devuelve el valor almacenado
// void write( int x)  →Se almacena x

public class IntCell
{
      //Métodos públicos
      public  int read( )
           {
                return storedValue;
      }
      public void write( int x)
           {
                storedValue=x;
      }
      // Representación privada interna de los datos
      private int storedValue;
}
```

Este código representa la declaración de clase para un objeto *IntCell*, está compuesta de dos partes, una pública y una privada. La parte pública es visible para el usuario del objeto. En este código se dispone de métodos que leen y escriben en el objeto (*read y write*). Los datos de la sección privada (*storedValue*) son invisibles para el usuario del objeto. El acceso a ellos es a través de los métodos mencionados anteriormente; *main* no puede acceder directamente a él (Weiss,2013).

Podemos concluir que la clase define **miembros** que pueden ser campos (datos) o métodos (funciones). Los métodos pueden actuar sobre los campos y pueden invocar otros métodos. La declaración *public* significa que se puede acceder mediante el operador punto, mientras que *private* significa que a ese miembro solo pueden acceder otros miembros de esta clase.

```
// Ejemplo de uso de la clase IntCell

public class TestIntCell
{
      public static void main( String [ ] args)
      {
            IntCell m= new IntCell( );

            m.write(5);
            System.out.println("Cell contents: "+m.read( );

            // La siguiente línea seria illegal si no
               estuviera desactivada por un
            // comentario, porque storedValue es un miembro
               privado
            // m.storedValue=0;
      }
}
```

Cuando se diseña una clase, se dice que se **especifica**, es decir, describe lo que puede hacerse con un objeto. La implementación representa los detalles internos acerca de cómo se satisfacen las especificaciones. En muchos lenguajes la implementación y la especificación se mantienen en archivos fuentes separados, por ejemplo, C++ dispone de los archivos .h y .cpp. En *java*, el programa ***javadoc*** puede ejecutarse para generar automáticamente documentación para las clases; la salida de *javadoc* es un conjunto de archivos html que pueden visualizarse o imprimirse (Weiss,2013).

1.3.1 Métodos básicos

Algunos métodos son comunes a todas las clases, entre ellos se encuentran los mutadores, accesores y tres métodos especiales (los constructores, *toString* y *equals*).

En *java* el método que controla cómo se crea e inicializa un objeto es el **constructor**. Si no se proporciona ningún constructor, se genera uno predeterminado que inicializa cada dato miembro utilizando los valores predeterminados normales. Para escribir un constructor tenemos que proporcionar un método que tenga el mismo nombre que la clase y ningún tipo de retorno (Weiss,2013).

Algunas veces se desea acceder a los campos en una declaración *private*, esto podría hacerse cambiando la declaración a *public*, sin embargo esto violaría el principio de ocultamiento. En lugar de ello se define un método que examina pero no modifica el estado de un objeto (accesor); si lo que se desea es modificar el estado entonces el método se denomina mutador. Los accesores pueden tener nombres que inicien con *get*, mientras de los mutadores pueden tener nombres que inicien con *set*.

Con método constructor *toString* se puede mostrar el estado de un objeto. El método *equals* se utiliza para comprobar si dos objetos representan el mismo valor.

Cuando es ejecutado el comando *java*, se invoca al método *main* del archivo de clase referenciado por el comando *java*; cada clase puede disponer de su propio método *main* por lo que es posible probar la funcionalidad básica de las clases individuales, aunque esto proporcione más visibilidad de la que se desea permitir. Otros constructores adicionales son *this*, *instanceof* y *static*.

El constructor *this* es una referencia al objeto actual. Puede utilizarse para enviar el objeto actual como una unidad a algún otro método. Podemos utilizar *this* dentro de un constructor para invocar a uno de los otros constructores de la clase.

El operador *instanceof* se utiliza para comprobar si una expresión es una instancia de alguna clase determinada.

Los campos y métodos declarados con la palabra clave *static* son miembros estáticos, mientras que si se declaran sin esta palabra se dice que son miembros de instancia. Un método estático es un método que no necesita un objeto que lo controle, y que por tanto se suele llamar proporcionando un nombre de clase en lugar del nombre del objeto controlador. El método estático más común es *main*. Los campos estáticos se utilizan cuando tenemos una variable que tiene que ser compartida por todos los miembros de una cierta clase. Son esencialmente variables globales cuyo ámbito es el de una clase. Los campos estáticos se inicializan en el momento de cargar la clase.

1.3.2 Paquetes

Los paquetes se utilizan para organizar clases similares. *Java* proporciona varios paquetes predefinidos, incluyendo *java.io, java.lang* y *java.util*. El paquete *java.lang* incluye entre otras, las clases *Integer, Math, String* y *System*. Algunas de las clases del paquete *java.util* son *Date, Random* y *Scanner*. El paquete *java*.io se utiliza para la entrada-salida e incluye diversas clases para el flujo de datos (Weiss,2013).

La directiva import se utiliza para proporcionar una abreviatura para un nombre de clase completamente cualificado.

Para indicar que una clase forma parte de un paquete, debemos hacer dos cosas, primero incluir la instrucción *package* en la primera línea de código, antes de la definición de clase, luego debemos colocar el código en un subdirectorio apropiado. Los paquetes se buscan en las ubicaciones designadas en la variable *classpath*. La variable classpath especifica los archivos y directorios que hay que explorar para encontrar las clases. Los paquetes tienen varias reglas de visibilidad. En primer lugar, si no se especifica ningún modificador de visibilidad para un campo, tendrá visibilidad de paquete, es decir, será visible únicamente para las restantes clases del mismo paquete. En segundo lugar, fuera del paquete solo se pueden utilizar las clases públicas de ese paquete. Todas las clases que no forman parte de un paquete pero que son alcanzables a través de la variable *classpath*, se consideran parte del mismo paquete predeterminado (Weiss,2013).

1.4 Integración de conceptos

En esta sección se analizarán una serie de ejemplos que tienen que ver con los conceptos establecidos en los anteriores apartados, la idea es que se puedan integrar en programas que desarrollen esos contenidos de forma práctica.

1.4.1 Primer programa

```
1.  // Programa uno
2.  // clase principal

3.  public class ProgramaUno {
      public static void main( String [ ] args )
      {
           System.out.println( "Saludos…este es mi primer
           programa" );
      }
4.  }
```

En el código anterior se puede ver el uso de los comentarios escribiendo las dobles diagonales. Es recomendable que durante todo el desarrollo de un programa sean empleados los comentarios, con la finalidad de hacer más fácil la ubicación de secciones que puedan contener elementos esenciales del código principal.

En la línea tres se nombra a la clase con el mismo nombre del archivo con el que se guardará en el directorio establecido para ello, esto es, deberá existir un archivo con el nombre ProgramaUno. *java*. Es importante notar que el nombre de archivo debe ser idéntico al nombre de la clase. La línea cinco establece la clase principal o ***main***, esta es la clase que buscará el comando *java* cuando sea ejecutado el programa. Por el momento es necesario escribir la línea tal cual aparece, no omitiendo los caracteres de llave **"{", "}",** ya que indican el principio y fin de una clase. Las clases son públicas por lo que pueden ser accesadas por el usuario y este puede conocer como efectúan la **función** (Weiss,2013).

En la línea siete se emplean la clase *System* (inicial en mayúscula) con su atributo *out* (la variable definida con la que se efectuarán las acciones), y el método o acción *println*. Con esta clase mostramos en pantalla el texto que está entre comillas. Es necesario que copie este programa y lo ejecute a fin de familiarizarse con el lenguaje. Los pasos son:

1. Escriba en un block de notas el texto del programa, es recomendable el **Notepad++** (enlace 2) ya que puede reconocer el texto basado en *Java*.

2. Guarde el programa con el mismo nombre de la clase: *ProgramaUno.java,* la extensión *.java* puede ubicarla en el campo "*tipo*" del *Notepad++*.

3. Ejecute ***javac* ProgramaUno.*java*,** esto permitirá al compilador incluido en el JDK (*Java Development Kit*) aceptar el código fuente y transformarlo en *bytecode* para ser ejecutado por la máquina virtual de *Java*, dando lugar al archivo *ProgramaUno.class*.

4. Ejecute *java ProgramaUno*, y observe el resultado. Si todo ha salido bien debe observar el mensaje **Saludos…este es mi primer programa.**

Enlace 2

About

https://notepad-plus-plus.org/

1.4.2 Prueba de los operadores unarios y de asignación

Los **operadores** unarios en *Java* son aquellos que solo requieren un operando para funcionar. El siguiente ejemplo permite conocer su funcionamiento (Weiss,2013).

```
1. public class PruebaUnarios
2. {
3. // Programa para mostrar los operadores unarios
4. // La salida esperada es como sigue:
5. // 12 8 6
6. // 6 8 6
7. // 6 8 14
8. // 22 8 14
9. // 24 10 33
10. public static void main( String [ ] args )
11. {
12. int a = 12, b = 8, c = 6;
13. System.out.println( a + " " + b + " " +  c ); //12 8 6
14. a = c;
15. System.out.println( a + " " + b + " " +  c );//6 8 6
16. c += b;
17. System.out.println( a + " " + b + " " +  c );//6 8 14
18. a = b + c;
19. System.out.println( a + " " + b + " " +  c );//22 8 14
20. a++;//23
21. ++b;//9
22. c = a++ + ++b;//23+10
23. System.out.println( a + " " + b + " " +  c );//24 10 33
24. }
25. }
```

En la línea doce definimos tres variables de tipo entero (int) que como se recordará son **tipos primitivos.** En la línea trece se muestran los valores de a,b,c sin haber efectuado operación alguna. En la línea catorce aplicamos el *operador de asignación* (=), y el resultado es asignar el valor de *c* a la variable *a*.

La línea dieciséis presenta una contracción de la operación c=c+b, para las líneas 20, 21 y 22 se presentan los resultados en la siguiente tabla.

Operación	Antes	Durante	Después
a++	22	22	23
++b	8	9	9

La segunda columna (antes) muestra el valor inicial de la variable, la tercer columna (durante) muestra el valor de la expresión, y la última columna (después) muestra el valor final de la variable, después de evaluarse la expresión. Como puede verse la posición del operador con respecto a la variable puede cambiar el resultado.

Para entender mejor esto considérese la **sentencia**:

```
j=i++;
```

asigna a j, el valor que tenía i. Por ejemplo, si i valía 3, después de ejecutar la sentencia, j toma el valor de 3 e i el valor de 4. Lo que es equivalente a las dos sentencias:

```
j=i;
i++;
```

Un resultado distinto se obtiene si el operador ++ está a la izquierda del operando:

```
j=++i;
```

Asigna a j el valor incrementado de i. Por ejemplo, si i valía 3, después de ejecutar la sentencia j e i toman el valor de 4. Lo que es equivalente a las dos sentencias:

```
++i;
j=i;
```

Se deberá de tener siempre el cuidado de inicializar la variable, antes de utilizar los operadores unarios con dicha variable.

1.4.3 Prueba del operador condicional

Este operador **ternario** tomado de C/C++ permite devolver valores en función de una expresión lógica.

La sentencia de asignación:

```
valor = (expresionLogica ? expresion_1 : expresion_2);
```

Es equivalente a:

```
if (expresionLogica)
     valor = expresion_1;
else
     valor = expresion_2
```

Analice el siguiente programa:

```
1.  public class MinTest
2.  {
3.  public static void main( String [ ] args )
4.  {
5.  int a = 3;
6.  int b = 7;

7.  System.out.println( min( a, b ) );
8.  }

9.  // Function definition
10. public static int min( int x, int y )
11. {
12. return x < y ? x : y;
13. }
14. }
```

La línea siete presenta en pantalla el resultado de la función **min,** la cual utiliza dos parámetros de entrada **a** y **b.** La función está definida fuera de la sección principal del programa, en la línea doce se toman los valores x, y y se ejecuta la instrucción:

```
return x < y ? x :y
```

Similar a:

```
if (x<y)
     x;
else
     y
```

la sentencia *return* devuelve el valor de la función.

1.4.4 Prueba del método *Scanner*

La clase *Scanner* está diseñada para leer los *bytes* y convertirlos en valores primitivos *(int, double, bool, etc.)* o en valores *String* (Weiss,2013).

```
1.  import java.util.Scanner;

2.  public class DivideEnDos
3.  {
4.  public static void main( String [ ] args )
5.  {
6.  Scanner in = new Scanner( System.in );
7.  int x;
8.  System.out.println( "Introduzca un número entero: " );
9.  try
10. {
11. String oneLine = in.nextLine( );
12. x = Integer.parseInt( oneLine );
13. System.out.println( "Half of x is " + ( x / 2 ) );
14. }
15. catch( NumberFormatException e )
16. { System.out.println( e ); }
17. }
18. }
```

Para importar clases de un paquete se usa el comando *import*. Se puede importar una clase individual

```
import java.awt.Font;
```

o bien, se pueden importar las clases declaradas públicas de un paquete completo, utilizando un asterisco (*) para reemplazar los nombres de clase individuales.

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

En este caso la línea uno importa al programa la clase *Scanner*, que se utiliza para crear un objeto *in* que toma los parámetros *System.in*. En la línea ocho se muestran en pantalla el mensaje para el usuario.

En las líneas nueve y quince se colocan bloques para el manejo de errores. La técnica básica consiste en colocar las instrucciones que podrían provocar problemas dentro de un **bloque try,** y colocar a continuación uno o más **bloques *catch*,** de tal forma que, si se provoca un error de un determinado tipo, lo que sucederá es que se tendrá que **saltar al bloque catch** capaz de **gestionar** ese tipo de error específico. Suponiendo que no se hubiesen provocado errores en el bloque *try*, nunca se ejecutarían los bloques catch.

En el bloque *try* se está tomando un valor que introduce el usuario usando el objeto *in* (de la clase *Scanner*) y ejecutando el método *nextLine* que permite leer todos los caracteres hasta encontrar un ENTER. Los caracteres son convertidos en números enteros con el método *parseInt*. La ocurrencia de

un error es tratado en el bloque *catch*. *Java* lanza una excepción en respuesta a una situación poco usual. Las excepciones en *Java* son objetos de clases derivadas de la clase base *Exception*.

En este caso, si se introducen caracteres no numéricos, o no se quitan los espacios en blanco al principio y al final del *String*, mediante la función *trim*, se lanza una excepción *NumberFormatException*.

En el siguiente programa se analizarán otras instrucciones (Weiss,2013).

```
1.  import java.util.Scanner;

2.  public class LeerCadenas
3.  {
4.  public static void main( String [ ] args )

5.  {
6.  String [ ] array = getStrings( );//metodo
7.  for( int i = 0; i < array.length; i++ )
8.  System.out.println( array[ i ] );
9.  }

10. // Read an unlimited number of String; return a String [ ]
11. public static String [ ] getStrings( )
12. {
13. Scanner in = new Scanner( System.in );
14. String [ ] array = new String[ 5 ];
15. int itemsRead = 0;

16. System.out.println( "Enter any number of strings, one
    per line; " );
17. System.out.println( "Terminate with empty line: " );

18. while( in.hasNextLine( ) )
19. {
20. String oneLine = in.nextLine( );
21. if( oneLine.equals( "" ) )
22. break;
23. if( itemsRead == array.length )
24. array = resize( array, array.length * 2 );
25. array[ itemsRead++ ] = oneLine;
26. }
```

```
27. System.out.println( "Done reading" );
28. return resize( array, itemsRead );
29. }

30. // Resize a String[ ] array; return new array
31. public static String [ ] resize( String [ ] array,
    int newSize )
32. {
33. String [ ] original = array;
34. int numToCopy = Math.min( original.length, newSize );

35. array = new String[ newSize ];
36. for( int i = 0; i < numToCopy; i++ )
      a.  array[ i ] = original[ i ];
37. return array;
38. }
39. }
```

En la línea cuatro se tiene la clase principal, como se puede ver siempre colocamos *String[ ] args* , aunque se le suele dar el nombre *args,* no es obligatorio que este parámetro se llame así, se puede utilizar cualquier otro nombre. La función de este arreglo es aceptar entradas desde el teclado para ser pasadas como parámetros a la clase, por ejemplo:

C:\> *java* Busqueda 3 5 7 8

en esta operación los parámetros 3, 5, 7, 8 son pasados a la clase en el objeto *args* que es un arreglo de tipo String, en este caso el arreglo señala un espacio que ocupa cada uno de los parámetros de entrada como se muestra:

args[0]	"3"
args[1]	"5"
args[2]	"7"
args[3]	"8"

El cuerpo del programa principal inicia con la declaración de un arreglo de tipo *String* cuyos elementos serán los obtenidos del método *getStrings().*

La función *getStrings()* crea un objeto de la clase *Scanner* que servirá para leer caracteres desde el teclado, en este caso se emplea el método *hasNextline*, este método regresa un booleano true, si existe otra línea de entrada para la clase *Scanner*, esta es la condición del ciclo *while* , dentro del ciclo se estará llenando la variable *oneLine* con el método *nextLine* que permite leer todos los caracteres hasta encontrar un ENTER hasta que se encuentre una línea vacía lo que terminará el ciclo (Weiss,2013).

Esta función también llama a otra función, que permite redimensionar el tamaño del arreglo, esto con el fin de poder admitir más caracteres.

1.4.5 Prueba del manejo de bloques de errores

Como se dijo anteriormente *Java* permite manejar estructuras de errores con la finalidad de permitir un mejor flujo del programa. El código siguiente inicia con la importación de cinco clases que pertenecen al mismo número de paquetes.

El paquete *java*.util es muy útil porque contiene 34 clases y 13 **interfaces** que implementan algunas de las estructuras de datos más comunes, algunas operaciones sobre fechas y sobre el calendario, y otras cosas (Weiss,2013).

Además el paquete *java.util* incluye otros subpaquetes que son: *java.util.mime, java.util.zip* y *java.util.* jar que sirven respectivamente para tratar archivos de tipo MIME, de tipo ZIP y de tipo *Java* Archive (JAR). La clase *Scanner* ya fue analizada, la clase *NoSuchElementException* es una excepción que indica que no hay más elementos en una secuencia enumerada.

El paquete *java*.io es el encargado de gestionar las operaciones de entrada/salida. Entrada estándar sería *System.in* (es un objeto *InputStream*), salida estándar sería *System.out* y salida estándar de errores sería *System.err* (las salidas son objetos *PrintStreams*). La excepción que se obtiene (*IOException*) con este paquete ssignifica que se ha producido un error en la entrada/salida. Un *InputStream* es cualquier dispositivo desde el cual se leen *bytes*. Puede ser el teclado, un archivo, un *socket*, o cualquier otro dispositivo de entrada. Esto, por un lado es una ventaja. Si todas esas cosas son *InputStream*, podemos hacer código que lea de ellas sin saber qué estamos leyendo. Una clase *Reader* es una clase que lee caracteres. Hay una clase en *java*, la *InputStreamReader*, que nos hace la conversión entre el *stream* y el *reader*. El problema aquí es que con estas clases se requiere conocer el número de caracteres que se leen, cosa que no se sabe pues es el usuario quien está introduciéndolos. Para ello se emplea la clase *BufferedReader*, con esta clase se obtiene de una sola vez todos los caracteres que se introducen por algún medio (Weiss,2013).

De la línea seis a la dieciocho se tiene una primera clase con su *main()*, esta clase toma dos números enteros examinando en un *if* si se cumple con dicha condición, y da como resultado el mayor de ellos. Como la entrada puede ser cualquier carácter, tenemos un *System.err,* el método err es usado por convención para desplegar un mensaje de error que indica que se debe poner una atención inmediata.

En la segunda clase ***MaxTestB,*** los números se leen uno a uno, y se deja el manejo del error a una excepción.

La clase ***MaxTestC,*** permite introducir ambos números en una línea con el *System.in* y después con otro método extraer los dos enteros. De igual manera los errores son tratados a partir de una excepción.

Finalmente, la clase ***MaxTestD,*** se trata de asegurar que únicamente sean dos enteros los que se introducen empleando otra llamada a ***Scanner*** con el ***hasNext(),*** para identificar si algo va mal (Weiss,2013).

```
1.  import java.util.Scanner;
2.  import java.util.NoSuchElementException;

3.  import java.io.IOException;
4.  import java.io.InputStreamReader;
5.  import java.io.BufferedReader;


6.  class MaxTestA
7.  {
8.  public static void main( String [ ] args )
9.  {
10. Scanner in = new Scanner( System.in );
11. int x, y;

12. System.out.println( "Introduzca dos enteros: " );
13. if( in.hasNextInt( ) )
14. {
      a.  x = in.nextInt( );
      b.  if( in.hasNextInt( ) )
      c.  {
      d.  y = in.nextInt( );
      e.  System.out.println( "Max: " + Math.max( x, y ) );
      f.  return;
      g.  }
15. }

16. System.err.println( "Error: need two ints" );
17. }
18. }

19. class MaxTestB
20. {
21. public static void main( String [ ] args )
22. {
23. Scanner in = new Scanner( System.in );

24. System.out.println("Introduzca dos enteros:" );
25. try
```

```
26. {
      a.  int x = in.nextInt( );
      b.  int y = in.nextInt( );
      c.  System.out.println( "Max: " + Math.max( x, y ) );
27. }
28. catch( NoSuchElementException e )
29. { System.err.println( "Error: need two ints" ); }
30. }
31. }

32. class MaxTestC
33. {
34. public static void main( String [ ] args )
35. {
36. Scanner in = new Scanner( System.in );

37. System.out.println( "Introduzca dos enteros en una
    linea: " );
38. try
39. {
      a.  String oneLine = in.nextLine( );
      b.  Scanner str = new Scanner( oneLine );
      c.  int x = str.nextInt( );
      d.  int y = str.nextInt( );
      e.  System.out.println( "Max: " + Math.max( x, y ) );
40. }
41. catch( NoSuchElementException e )
42. { System.err.println( "Error: need two ints" ); }
43. }
44. }

45. class MaxTestD
46. {
47. public static void main( String [ ] args )
48. {
49. Scanner in = new Scanner( System.in );

50. System.out.println( "Enter 2 ints on one line: " );
51. try
```

```
52. {
      a.  String oneLine = in.nextLine( );
      b.  Scanner str = new Scanner( oneLine );
      c.  int x = str.nextInt( );
      d.  int y = str.nextInt( );
      e.  if( !str.hasNext( ) )
      f.  System.out.println( "Max: " + Math.max( x, y ) );
      g.  else
      h.  System.err.println( "Error: extraneous data on
          the line." );
53. }
54. catch( NoSuchElementException e )
55. { System.err.println( "Error: need two ints" ); }
56. }
57. }

58. class MaxTestE
59. {
60. public static void main( String [ ] args )
61. {
62. BufferedReader in = new BufferedReader( new
          i.  InputStreamReader( System.in ) );

63. System.out.println( "Enter 2 ints on one line: " );
64. try
65. {
      a.  String oneLine = in.readLine( );
      b.  if( oneLine == null )
      c.  return;
      d.  Scanner str = new Scanner( oneLine );

      e.  int x = str.nextInt( );
      f.  int y = str.nextInt( );

      g.  System.out.println( "Max: " + Math.max( x, y ) );
66. }
67. catch( IOException e )
68. { System.err.println( "Unexpected I/O error" ); }
69. catch( NoSuchElementException e )
```

```
70. { System.err.println( "Error: need two ints" ); }
71. }
72. }
```

1.4.6 Manejo básico de archivos

El siguiente programa implementa una clase que permite leer un archivo de la línea de comandos y mostrar su contenido. Para ello se emplea la clase *FileReader,* que funciona de manera similar al *BufferedReader* (Weiss,2013).

```
1.  import java.util.Scanner;
2.  import java.io.FileReader;
3.  import java.io.IOException;

4.  public class ListFiles
5.  {
6.  public static void main( String [ ] args )
7.  {
8.  if( args.length == 0 )
      a.  System.out.println( "No files specified" );
9.  for( String fileName : args )
      a.  listFile( fileName );

10. }
11. public static void listFile( String fileName )
12. {
13. Scanner fileIn = null;

14. System.out.println( "FILE: " + fileName );
15. try
16. {
      a.  fileIn  = new Scanner( new FileReader( fileName ) );
      b.  while( fileIn.hasNextLine( ) )
      c.  {
      d.  String oneLine = fileIn.nextLine( );
      e.  System.out.println( oneLine );
      f.  }
17. }
18. catch( IOException e )
19. {  System.out.println( e ); }
20. finally
```

```
21. {
      a.  // Close the stream
      b.  if( fileIn != null )
      c.  fileIn.close( );
22. }
23. }
24. }
```

Recuérdese que la clase *main* implementa un arreglo de caracteres en una cadena *String*, que es almacenada en *args*, esto se utiliza para leer el nombre del archivo, a continuación, se implementa un método *fileIn*, para leer el contenido del archivo, al final cerramos el archivo con la finalidad de no quedarnos sin flujo de datos.

El siguiente ejemplo muestra el acceso al arreglo de la clase main (Weiss,2013).

```
public class Echo
{
      public static void main( String [ ] args )
      {
            for( int i = 0; i < args.length - 1; i++ )
                  System.out.print( args[ i ] + " " );
            if( args.length != 0 )
                  System.out.println( args[ args.length - 1 ] );
            else
                  System.out.println( "No arguments to echo" );
      }
}
```

Obsérvese que cada espacio del arreglo es ocupado por un carácter que es escrito en la línea de comandos al ejecutar la clase. Otro ejemplo es la siguiente clase.

```
import java.util.Scanner;

class Echo
{
      public static void main (String[] args)
      {

      String inData;
      Scanner scan = new Scanner( System.in );

      System.out.println("Enter the data:");
      inData = scan.nextLine();
```

```
        System.out.println("You entered:" + inData );
        }
}
```

1.4.7 Programa para generar números aleatorios

La clase ***Random*** proporciona un generador de números aleatorios que es más flexible que la función estática *random* de la clase *Math* (Weiss,2013).

```
1.  import java.util.Random;

2.  public class NumerosAleatorios
3.  {
4.  // Genera números aleatorios (1-100)
5.  // Muestra el número de ocurrencias de cada numero

6.  public static final int DIFF_NUMBERS     =     100;
7.  public static final int TOTAL_NUMBERS    = 1000000;

8.  public static void main( String [ ] args )
9.  {
10. // Create the array; initialize to zero
11. int [ ] numbers = new int [ DIFF_NUMBERS + 1 ];
12. for( int i = 0; i < numbers.length; i++ )
13. numbers[ i ] = 0;

14. Random r = new Random( );

15. // Generate the numbers
16. for( int i = 0; i < TOTAL_NUMBERS; i++ )
      a.  numbers[ r.nextInt( DIFF_NUMBERS ) + 1 ]++;

17. // Output the summary
18. for( int i = 1; i <= DIFF_NUMBERS; i++ )
      a.  System.out.println( i + ": " + numbers[ i ] );

19. }
20. }
```

Se importa la clase *Random* del paquete de clases para crear objetos de ese tipo, en seguida en las líneas ocho y nueve se declaran dos constantes usando la declaración **final**. La palabra reservada 1 en este contexto indica que una variable es de tipo constante, esto es, que no admitirá cambios después de su declaración y asignación de valor, final determina que un atributo no puede ser sobrescrito o redefinido. En la línea catorce se crea el arreglo llamado *numbers* y se inicializa en cero por medio del ciclo *for*. En la línea dieciocho se crea el objeto r que generará los números aleatorios, el rango de valores será hasta el límite definido por la constante *DIFF_NUMBERS*.

1.4.8 Uso de *javadoc*

En una clase es mejor documentar: a) Nombre de la clase, descripción general, número de versión, nombre de autores; b) documentación de cada constructor o método (especialmente los públicos) incluyendo: nombre del constructor o método, tipo de retorno, nombres y tipos de parámetros si los hay, descripción general, descripción de parámetros (si los hay), descripción del valor que devuelve (Weiss,2013).

```
1.  /**
2.  A class for simulating an integer memory cell
3.  @author Mark A. Weiss
4.  */
5.  public class IntCell
6.  {
7.  /**
8.  Get the stored value.
9.  @return the stored value.
10. */
11. public int read( )
12. {
13. return storedValue;
14. }

15. /**
16. Store a value
17. @param x the number to store.
18. */
19. public void write( int x )
20. {
21. storedValue = x;
22. }

23. private int storedValue;
}
```

La documentación es obtenida a partir de la **ejecución** **de *javadoc* Nombre _de _clase.*java***

1.4.9 Análisis de una clase

A continuación, se presenta la implementación de una clase y de la cual se explicarán algunos de los elementos que la componen (Weiss,2013).

```
1.  package weiss.math;

2.  import weiss.math.BigInteger;

3.  public class BigRational
4.  {
5.  public static final BigRational ZERO = new
    BigRational( );
6.  public static final BigRational ONE = new BigRational
    ( "1" );
7.  public BigRational( String str )
8.  {
9.  if( str.length( ) == 0 )
      a. throw new IllegalArgumentException( "Zero-length
         string" );

10. // Check for "/"
11. int slashIndex = str.indexOf( '/' );
12. if( slashIndex == -1 )
13. {
      a. // no denominator... use 1
      b. den = BigInteger.ONE;
      c. num = new BigInteger( str.trim( ) );
14. }
15. else
16. {
      a. den = new BigInteger( str.substring( slashIndex +
         1 ).trim( ) );
      b. num = new BigInteger( str.substring( 0, slashIndex
         ).trim( ) );
      c. check00( ); fixSigns( ); reduce( );
17. }
18. }
```

```
19. private void check00( )
20. {
21. if( num.equals( BigInteger.ZERO ) && den.equals(
    BigInteger.ZERO ) )
      a.  throw new IllegalArgumentException( "0/0" );
22. }

23. // Ensure that the denominator is positive
24. private void fixSigns( )
25. {
26. if( den.compareTo( BigInteger.ZERO ) < 0 )
27. {
      a.  num = num.negate( );
      b.  den = den.negate( );
28. }
29. }

30. // Divide num and den by their gcd
31. private void reduce( )
32. {
33. BigInteger gcd = num.gcd( den );
34. num = num.divide( gcd );
35. den = den.divide( gcd );
36. }

37. public BigRational( BigInteger n, BigInteger d )
38. {
39. num = n;
40. den = d;
41. check00( ); fixSigns( ); reduce( );
42. }

43. public BigRational( BigInteger n )
44. {
45. this( n, BigInteger.ONE );
46. }

47. public BigRational( )
48. {
```

```
49. this( BigInteger.ZERO );
50. }

51. public BigRational abs( )
52. {
53. return new BigRational( num.abs( ), den );
54. }

55. public BigRational negate( )
56. {
57. return new BigRational( num.negate( ), den );
58. }

59. public BigRational add( BigRational other )
60. {
61. BigInteger newNumerator =
      i.  num.multiply( other.den ).add(
      ii. other.num.multiply( den ) );
62. BigInteger newDenominator =
      i.  den.multiply( other.den );

63. return new BigRational( newNumerator, newDenominator );
64. }

65. public BigRational subtract( BigRational other )
66. {
67. return add( other.negate( ) );
68. }

69. public BigRational multiply( BigRational other )
70. {
71. BigInteger newNumer = num.multiply( other.num );
72. BigInteger newDenom = den.multiply( other.den );

73. return new BigRational( newNumer, newDenom );
74. }

75. public BigRational divide( BigRational other )
76. }
```

```
77. if( other.num.equals( BigInteger.ZERO ) && num.
    equals( BigInteger.ZERO))
      a. throw new ArithmeticException( "ZERO DIVIDE BY
         ZERO" );
78. BigInteger newNumer = num.multiply( other.den );
79. BigInteger newDenom = den.multiply( other.num );

80. return new BigRational( newNumer, newDenom );
81. }

82. public boolean equals( Object other )
83. {
84. if( ! ( other instanceof BigRational ) )
      a. return false;

85. BigRational rhs = (BigRational) other;

86. return num.equals( rhs.num ) && den.equals( rhs.den );
87. }

88. public String toString( )
89. {
90. if( den.equals( BigInteger.ZERO ) )
      a. if( num.compareTo( BigInteger.ZERO ) < 0 )
      b. return "-infinity";
      c. else
      d. return "infinity";

91. if( den.equals( BigInteger.ONE ) )
      a. return num.toString( );
92. else
      a. return num + "/" + den;
93. }

94. private BigInteger num;  // only this can be neg
95. private BigInteger den;
96. }
```

Como se dijo anteriormente la palabra *package* permite acceder a un paquete que contiene clases. Los paquetes resuelven el problema del conflicto entre los nombres de las clases. Al crecer el número de

clases crece la probabilidad de designar con el mismo nombre a dos clases diferentes. Es importante considerar lo siguiente (Weiss,2013):

- Los archivos *.java* que no contienen la palabra *package*, deben colocarse en la carpeta principal del proyecto.
- Si por ejemplo un archivo *.java* contiene un *package* **paquete1;** se debe crear en la carpeta del proyecto un **subdirectorio con el nombre paquete1** y meter ahí el archivo *.java* correspondiente.
- Si un fichero *.java* lleva dentro un *package* paquete1.subpaquete1; se debe crear en la carpeta del proyecto un subdirectorio paquete1 y dentro de este otro subdirectorio subpaquete1 y dentro de este el archivo fuente *.java* que corresponda.

Para compilar, debe situarse en la carpeta del proyecto y escribir:

*java***c codigo1.***java* **paquete1\codigo2.***java* **paquete1\subpaquete1\codigo3.***java*

Una vez que se han generado los *.class,* se debe ejecutar situándose en la carpeta principal del proyecto y de acuerdo a la clase que contiene el *main.*

Al usar la palabra reservada *static* la declaración de la constante ha de realizarse obligatoriamente en cabecera de la clase. Si se trata de incorporar en un método se obtendrá un error.

1.5 Análisis de algoritmos

Un algoritmo es un conjunto claramente especificado de instrucciones que la computadora seguirá para resolver un problema. Una vez que el algoritmo se verifica, se debe determinar la cantidad de recursos, como por ejemplo, tiempo y espacio de memoria. Este paso se denomina análisis de algoritmos.

El tiempo de ejecución de un algoritmo es una función del tamaño de la entrada. Su valor exacto dependerá de muchos factores como, por ejemplo, la velocidad de la máquina utilizada, de la calidad del compilador y en algunos casos, de la calidad del programa. De las funciones que comúnmente podemos encontrar en el análisis de algoritmos, las lineales representan los algoritmos más eficientes. Por otra parte, algunos algoritmos no lineales requieren tiempos de ejecución muy grandes. Por ejemplo, una función cúbica tiene un término dominante que es igual a una constante multiplicada por N^3. De manera general, la expresión *O(NlogN)* representa una función cuyo término dominante es *N* veces el logaritmo de *N*. El logaritmo es una función que crece lentamente. Una forma de comparar dos funciones es a través de la tasa de crecimiento, la cual adquiere máxima importancia cuando *N* es suficientemente grande (Weiss,2013).

Se utiliza la notación *O*, o notación de **Landau**, para capturar el término más dominante de una función y para representar la tasa de crecimiento. Los algoritmos cuadráticos no resultan prácticos

para tamaños de entrada superiores a unos pocos miles, mientras que los algoritmos cúbicos no resultan prácticos para tamaños de entradas de solo unos pocos centenares.

1.5.1 Análisis asintótico

La pregunta obligada en el análisis de algoritmos es: si se tienen dos algoritmos para una tarea, ¿cómo saber cuál es mejor?

Una forma empírica de hacer esto es implementar los algoritmos y ejecutar los dos programas en una computadora para diferentes entradas y ver cuál lleva menos tiempo. Hay muchos problemas con este enfoque para el análisis de algoritmos:

1. Es posible que para algunas entradas, el primer algoritmo tenga un mejor rendimiento que el segundo. Y para algunas entradas, el segundo funciona mejor.

2. También podría ser posible que, para algunas entradas, el primer algoritmo funcione mejor en una máquina y el segundo funcione mejor en otra máquina para algunas otras entradas.

El **análisis asintótico** es muy utilizado, pues solventa los problemas anteriores al analizar algoritmos. En el análisis asintótico, se evalúa el rendimiento de un algoritmo en términos de tamaño de entrada (no se mide el tiempo real de ejecución). Se calcula, cómo el tiempo (o espacio) tomado por un algoritmo aumenta con el tamaño de entrada (Weiss,2013).

Por ejemplo, considere el problema de búsqueda (buscar un elemento determinado) en una matriz ordenada. Una forma de buscar es Búsqueda lineal (el orden de crecimiento es lineal) y de otra manera es Búsqueda binaria (el orden de crecimiento es logarítmico). Para comprender cómo el análisis asintótico resuelve los problemas mencionados anteriormente al analizar algoritmos, suponga que se ejecuta la búsqueda lineal en una computadora rápida y la búsqueda binaria en una computadora lenta. Para valores pequeños de tamaño de matriz de entrada n, la computadora rápida puede tomar menos tiempo. Pero, después de cierto valor de tamaño de matriz de entrada, la búsqueda binaria definitivamente comenzará a tomar menos tiempo en comparación con la búsqueda lineal, aunque la búsqueda binaria se está ejecutando en una máquina lenta. El motivo es el orden de crecimiento de la búsqueda binaria con respecto al tamaño de entrada logarítmico, mientras que el orden de crecimiento de la búsqueda lineal es lineal. Por lo tanto, las constantes dependientes de la máquina siempre se pueden ignorar después de ciertos valores de tamaño de entrada.

Aunque el análisis asintótico no es perfecto, es la mejor manera disponible para analizar algoritmos. Por ejemplo, suponga que hay dos algoritmos de clasificación que toman *1000nLogn* y *2nLogn* respectivamente en una máquina. Ambos algoritmos son asintóticamente iguales (el orden de crecimiento es *nLogn)*. Entonces, con el Análisis asintótico, no podemos juzgar cuál es mejor ya que no tenemos un parámetro constante.

Además, en el análisis asintótico, siempre hablamos de tamaños de entrada más grandes que un valor constante. Es posible que esas grandes entradas nunca se entreguen al software y un algoritmo que es asintóticamente más lento, siempre se comporta mejor para su situación particular. Por lo tanto,

se puede terminar eligiendo un algoritmo que es asintóticamente más lento pero más rápido para el software.

Por ejemplo, tómese el siguiente método que realiza una búsqueda lineal (Weiss,2013).

```
1)  import java.util.Scanner;

2)  class LinearSearch
3)  {
4)  public static void main(String args[])
5)  {
6)  int c, n, search, array[];

7)  Scanner in = new Scanner(System.in);
8)  System.out.println("Enter number of elements");
9)  n = in.nextInt();
10) array = new int[n];

11) System.out.println("Enter " + n + " integers");

12) for (c = 0; c < n; c++)
13) array[c] = in.nextInt();

14) System.out.println("Enter value to find");
15) search = in.nextInt();

16) for (c = 0; c < n; c++)
17) {
18) if (array[c] == search)     /* Searching element is
    present */
19) {
20) System.out.println(search + " is present at
    location " + (c + 1) + ".");
21) break;
22) }
23) }
24) if (c == n)  /* Element to search isn't present */
25) System.out.println(search + " isn't present in
    array.");
26) }
27) }
```

En el análisis del peor de los casos (mayormente utilizado), se calcula el límite superior en tiempo de ejecución de un algoritmo. Se debe conocer el caso que causa que se ejecute el número máximo de operaciones. Para la búsqueda lineal, el peor caso ocurre cuando el elemento que se va a buscar (*search* en el código anterior) no está presente en la matriz. Cuando *search* no está presente, las funciones de búsqueda ( ) lo comparan con todos los elementos (n) del array[ ] uno por uno. Por lo tanto, la peor complejidad de tiempo de la búsqueda lineal sería Θ (n).

En el análisis del caso promedio (algunas veces empleado), se toman todas las entradas posibles y se calcula el tiempo de ejecución para todas las entradas. Después se suman todos los valores calculados y se dividen por el número total de entradas. Debe saber (o predecir) la distribución de los casos. Para el problema de búsqueda lineal, suponga que todos los casos están distribuidos uniformemente (incluido el caso de que *search* no esté presente en el conjunto). Entonces se suman todos los casos y se divide la suma por (n + 1). A continuación, se muestra el valor de la complejidad promedio de tiempo de casos (Weiss,2013).

$$\text{Tiempo para el análisis del caso promedio} = \frac{\sum_{(n=1)}^{n+1} \theta(i)}{n+1}$$

$$\text{Tiempo para el análisis del caso promedio} = \frac{\theta((n+1)*(n+2)/2)}{n+1} = \Theta(n)$$

En el análisis del mejor de los casos, se calcula un límite inferior en el tiempo de ejecución de un algoritmo. Se debe conocer el caso que causa que se ejecute un número mínimo de operaciones. En el problema de búsqueda lineal, el mejor caso ocurre cuando *search* está presente en la primera ubicación. El número de operaciones en el mejor de los casos es constante (no depende de n). Entonces, la complejidad del tiempo en el mejor de los casos sería Θ (1)

La mayoría de las veces, se realiza el análisis del peor de los casos para analizar algoritmos. En este análisis, garantizamos un límite superior en el tiempo de ejecución de un algoritmo que es una información buena.

El análisis del caso promedio no es fácil de hacer en la mayoría de las situaciones prácticas y rara vez se hace. En el análisis del caso promedio, se debe conocer (o predecir) la distribución matemática de todas las entradas posibles.

El análisis del mejor caso no es confiable. Garantizar un límite inferior en un algoritmo no proporciona ninguna información, ya que, en el peor de los casos, un algoritmo puede tardar años en ejecutarse.

Recuerde que la idea principal del análisis asintótico es tener una medida de la eficiencia de los algoritmos que no dependa de las constantes específicas de la máquina, y no requiere de la implementación de algoritmos ni el tiempo tomado por programas para ser comparados. Las notaciones asintóticas son herramientas matemáticas para representar la complejidad del tiempo de los algoritmos para el análisis asintótico. Las siguientes tres notaciones asintóticas se utilizan principalmente para representar la complejidad del tiempo de los algoritmos (Weiss,2013).

1. Notación Θ : la notación theta limita las funciones superiores e inferiores, por lo que define el comportamiento asintótico exacto. Una forma simple de obtener la notación Theta de una expresión es ignorar los términos de bajo orden e ignorar las constantes iniciales. Por ejemplo, considere la siguiente expresión:

   $$3n^3 + 6n^2 + 6000 = \Theta (n3)$$

   Se ignoran los términos de orden inferior ya que siempre habrá un n0 después del cual Θ (n3) tiene valores más altos que Θ(n2) independientemente de las constantes involucradas.

   Para una función dada g(n), denotamos que Θ(g(n)) es el siguiente conjunto de funciones: si f(n) es theta de g(n), entonces el valor f(n) siempre está entre c1 * g(n) y c2 * g(n) para valores grandes de n (n> = n0). La definición de theta también requiere que f (n) no sea negativa para valores de n mayores que n0.

2. Notación Big O: la notación Big O define un límite superior de un algoritmo, limita una función solo desde arriba. Por ejemplo, considere el caso de una selección para inserción. Se toma el tiempo lineal en el mejor de los casos y el tiempo cuadrático en el peor de los casos. Se puede decir con seguridad que la complejidad de tiempo del tipo selección para inserción es O (n2). Debe tenerse en cuenta que O (n2) también cubre el tiempo lineal. Si utilizamos la notación Θ para representar la complejidad temporal de la ordenación por inserción, tenemos que usar dos afirmaciones para el mejor y el peor de los casos:

   1. La peor complejidad de tiempo del caso de selección para inserción es Θ (n ^ 2).
   2. La mejor complejidad temporal del caso de selección para inserción es Θ (n).

   La notación de Big O es útil cuando solo tenemos límite superior en la complejidad de tiempo de un algoritmo. Muchas veces encontramos fácilmente un límite superior simplemente mirando el algoritmo.

3. Notación Ω: al igual que la notación Big O proporciona un límite superior asintótico en una función, la notación Ω proporciona un límite inferior asintótico. La notación Ω puede ser útil cuando tenemos un límite inferior en la complejidad del tiempo de un algoritmo. Como se discutió en la publicación anterior, el mejor rendimiento de un algoritmo en general no es útil, la notación Omega es la notación menos utilizada entre las tres. Para una función dada g (n), denotamos por Ω (g (n)) el conjunto de funciones.

   Ω (g (n)) = {f (n): existen constantes positivas c y n0 tal que 0 <= cg(n) <= f(n) para todo n> = n0}.

Considérese el mismo ejemplo de ordenamiento de inserción. La complejidad de tiempo de **selección para inserción** puede escribirse como Ω (n), pero no es una información muy útil sobre la ordenación por inserción, ya que generalmente estamos interesados en el peor de los casos y, en ocasiones, en el caso promedio.

Tema 2.
Soluciones Divide y Vencerás para la Ordenación y la Selección

A un método que esté parcialmente definido en términos de sí mismo, se le denomina **recursivo**. La recursión es una potente herramienta de programación que en muchos casos puede proporcionar algoritmos que son, a la vez, cortos y eficientes.

2.1 Recursividad

Un método recursivo es un método que hace directa o indirectamente una llamada a sí mismo. Esta acción puede parecer similar a los razonamientos circulares. La clave está en que el método F se llama así mismo dentro de un contexto diferente, que generalmente es más simple que el anterior. Por ejemplo, para examinar directorios y subdirectorios en una computadora puede hacerse recursivamente examinando cada archivo de cada subdirectorio, y luego examinando todos los archivos del directorio. Otro ejemplo puede ser buscar palabras en un diccionario, la estrategia recursiva puede ser la siguiente: si sabemos el significado de una palabra, habremos terminado; en caso contrario, la buscaremos en el diccionario. Si comprendemos todas las palabras de la definición,

habremos terminado. En caso contrario, intentamos averiguar qué significa la definición buscando recursivamente aquellas palabras que no conozcamos (Weiss,2013).

La **inducción** suele emplearse para demostrar teoremas que se cumplen para números positivos. Una demostración por inducción se lleva a cabo en dos etapas. En primer lugar, se demuestra que el teorema es cierto para los valores más pequeños, luego demostramos que si el teorema es cierto para los primeros casos, puede ampliarse para incluir el caso siguiente. Una vez que se haya demostrado cómo ampliar el rango de casos para los que el teorema se cumple, habremos demostrado que se cumple para todos los casos. Es decir, una demostración por inducción emplea el caso más sencillo como base y puede demostrarse a mano. Una vez que se establece la base utilizamos la hipótesis inductiva para asumir que el teorema es cierto para un valor k arbitrario, y luego con esa suposición se demuestra que, si el teorema es cierto para k, entonces es cierto para k+1.

2.1.1 Recursión básica.

Un método recursivo se define en términos de una instancia más pequeña de sí mismo. Debe existir algún caso base que pueda calcularse sin necesidad de usar la recursión. A partir de esto podemos plantear dos reglas de recursión fundamentales:

1. **Caso base.** Siempre tiene que haber un caso que se pueda resolver sin utilizar recursión.
2. **Progresión.** Toda llamada recursiva debe progresar hacia un caso base.

Un ejemplo de uso de la recursión es la impresión de números en cualquier base (Weiss,2013).

```
// Imprmir n en base 10, de forma recursiva.
// Precondicion: n>=0
public static void printDecimal ( long n)
     {
          if (n>=10)
          printDecimal (n/10);
          System.out.print ((char) ('0' +(n%10)));
     }|
```

El programa anterior es una rutina recursiva para imprimir N (varios números).

El siguiente programa es una forma recursiva para imprimir N números de cualquier base entre 2 y 16 (Weiss,2013).

```
//imprimir n en cualquier base de forma recursiva
//precondición: n>=0, base es válida.
public static void printInt (long n, int base)
{
     if (n>=base)
     printInt (n/base, base);
```

```
        System.out.print (DIGIT_TABLE.charAt ( int
        (n%base)));
    }
```

Cuando ocurre un fallo a la hora de progresar hacia el caso base, implica que el programa no funciona. Podemos hacer la rutina más robusta añadiendo una comprobación explícita para la base. El problema con esa estrategia es que la comprobación se realizaría en cada una de las llamadas recursivas, si la base fuese válida en la primera llamada sería absurdo volver a comprobarlo. Una forma de evitar esta ineficiencia consiste en programar una rutina de preparación. La rutina de preparación comprueba la validez de la base y luego invoca a la rutina recursiva. Utilizando esta estrategia, siempre podemos asumir que las llamadas recursivas funcionan porque, cuando se desarrolla una demostración, esta suposición se emplea como hipótesis inductiva. Esto nos lleva a la tercera regla fundamental de la recursión: *3) Es necesario creer.* Hay que asumir siempre que la llamada recursiva funciona.

No se debe olvidar que la implementación de la recursión requiere de una serie de tareas adicionales de gestión por parte de la computadora. La gestión de invocaciones de métodos en un lenguaje procedimental y orientado a objetos se lleva a cabo utilizando una pila de registros de activación. La recursión es un subproducto natural de este sistema de trabajo. La estrecha relación existente entre la recursión y las pilas sugiere que los programas recursivos siempre pueden implementarse de forma iterativa con una pila explícita. El resultado de utilizar una pila explícita es un código ligeramente más largo pero más rápido.

La cuarta regla fundamental de la recursión es: *4) Nunca duplique el trabajo que hay que realizar resolviendo la misma instancia de un problema mediante llamadas recursivas separadas.*

2.1.2 Algunos ejemplos

La resolución de un factorial es el producto de los N primeros enteros. Por tanto, podemos expresar N! como N veces (N-1)! (Weiss,2013).

```
// Evaluar n!
public static long factorial( int n)
{
      if( n<=1)
//Caso base
      return 1;
      else
           return n*factorial(n-1);
}
```

En una búsqueda binaria realizamos una búsqueda dentro de una matriz ordenada A examinando el elemento medio. Si encontramos una correspondencia habremos terminado. En caso contrario, si el elemento que estamos buscando es más pequeño que el elemento medio, buscamos en la submatriz situada a la izquierda del elemento medio; en caso contrario, buscamos en la submatriz situada a la

derecha del elemento medio. Este procedimiento asume que la submatriz no está vacía; si lo está, entonces el elemento no habrá podido ser encontrado.

```
public class BinarySearchRecursive
{
     public static final int NOT_FOUND = -1;

     /**
     * Performs the standard binary search
     * using two comparisons per level.
     * This is a driver that calls the recursive method.
     * @return index where item is found or NOT_FOUND if
     not found.
     */
     public static <AnyType extends Comparable<? super
     AnyType>> int
     binarySearch(
AnyType [ ] a, AnyType x )
     {
          return binarySearch( a, x, 0, a.length -1 );
     }

     /**
      * Hidden recursive routine.
      */
     private static <AnyType extends Comparable<? super
     AnyType>>
          int binarySearch( AnyType [ ] a, AnyType x, int
          low, int high )
     {
          if( low > high )
               return NOT_FOUND;

          int mid = ( low + high ) / 2;

          if( a[ mid ].compareTo( x ) < 0 )
               return binarySearch( a, x, mid + 1, high );
          else if( a[ mid ].compareTo( x ) > 0 )
               return binarySearch( a, x, low, mid - 1 );
          else
```

```
            return mid;
        }
        // Test program
        public static void main( String [ ] args )
        {
            int SIZE = 8;
            Integer [ ] a = new Integer [ SIZE ];
            for( int i = 0; i < SIZE; i++ )
                a[ i ] = i * 2;
            for( int i = 0; i < SIZE * 2; i++ )
                System.out.println( "Found " + i + " at " +
                binarySearch( a, i )  );
        }
}
```

2.2 Aplicaciones numéricas

La aritmética modular también conocida informalmente como aritmética de reloj, define el comportamiento cíclico en un intervalo denominado módulo.

Definición. Si m ≥1 es un entero, podemos decir que los enteros a y b son **congruentes** al módulo m si la diferencia a – b es divisible por m. Se puede escribir de la siguiente forma:

$$a \equiv b \pmod{m}$$

Se presentan dos problemas de la teoría de números para aplicar los principios de recursividad:

1. **Exponenciación modular**
2. **Máximo común divisor**

2.2.1 Exponenciación modular

La exponenciación modular es un tipo de exponenciación realizada sobre un módulo. Es particularmente útil en ciencias de la computación, especialmente en el campo de la **criptografía**. La siguiente rutina contiene la exponenciación modular (Weiss,2013).

```
/**
*Devuelve x^n (mod p)
*Asume que x, n>=0, p >0, x<p,0^0=1
*Puede producirse desbordamiento si p>31 bits
*/
public static long power ( long x, long n, long p)
```

```
{
     if(n==0)
          return 1;
     long tmp=power((x*x)%p,n/2,p);
     if( n%2!=0)
          tmp=(tmp*x)%p;
          return tmp;
}
```

La rutina se basa en el hecho de que si:

```
N es par:  X^N=(X*X)^N/2
```

N es impar X^N=X*X^N-1=X*(X*X)^N/2

2.2.2 Máximo común divisor

El máximo común divisor se define como el mayor entero D que es tanto divisor de A como de B. Se puede verificar que si D es divisor de A y B entonces también lo es de A-B. Esta observación no lleva a un algoritmo sencillo en el que vamos restando sucesivamente B de A, transformando el problema en otro más pequeño. Este algoritmo se conoce como algoritmo de Euclides, pero es inútil para números muy grandes. Restar repetidamente B de A es equivalente a convertir A en A mod B. Por lo tanto, mcd(A,B)=mcd(B, A mod B), como se muestra en la siguiente rutina (Weiss,2013).

```
/**
*Devolver el máximo común divisor
*/
public static long mcd(long a, long b)
{
     if(b==0)
          return a;
     else
          return mcd(b,a%b);
}
```

2.2.3 Inversa Multiplicativa

El algoritmo de máximo común divisor se utiliza implícitamente para resolver la ecuación:

```
AX≡1(modN)
```

Para ≤X<*N*

```
//Variables internas para fullGcd
```

```
private static long x;
private static long y;

/**
*Aplica a la inversa el algoritmo de Euclides para
determiner
*x e y tales que si gcd(a,b)=1,
*ax+by=1,
*/
private static void fullGcd(long a, long b)
{
      long x1,y1;
      if(b==0)
      {
            x=1;
            y=0;
      }
      else
      {
            fullGcd(b, a%b);
            x1=x; y1=y;
            x=y1;
            y=x1-(a/b)*y1;
      }
}

/**
*Resolver ax==1(mod n), asumiendo que gcd(a,n)=1,
*@return x,
*/
Public static long inverse( long a, long n)
{
      fullGcd(a,n);
      return x>0 ? x: x+n;
}
```

2.3 Algoritmos del tipo divide y vencerás

El algoritmo del tipo divide y vencerás es un algoritmo recursivo que por regla general es muy eficiente. Está compuesto de dos partes (Weiss,2013):

- **Divide,** durante la cual se resuelven recursivamente problemas sucesivamente más pequeños (excepto los casos base, que se resuelven directamente).
- **Vencerás,** en la que se forma la solución al problema original componiendo las soluciones a los subproblemas.

Tradicionalmente, las rutinas en las que el algoritmo contiene al menos dos llamadas recursivas, se denominan algoritmos del tipo divide y vencerás, mientras que las rutinas cuyo texto contiene solo una llamada recursiva no se clasifican dentro de esa categoría.

2.3.1 Problema de la suma máxima de subsecuencia contigua

El problema es: dada una serie de enteros (posiblemente negativos), encontrar el valor máximo de la suma e identificar la secuencia correspondiente. La suma máxima de una secuencia contigua es cero si todos los enteros son negativos. Supóngase que la secuencia de entrada de ejemplo es {4, -3, 5, -2, -1, 2, 6, -2}. Si se divide esta entrada en dos mitades, la suma máxima puede producirse de tres formas distintas (Weiss,2013):

Caso 1. Reside enteramente en la primera mitad.

Caso 2. Reside enteramente en la segunda mitad.

Caso 3. Comienza en la primera mitad y termina en la segunda mitad.

A continuación, se muestra el algoritmo para la suma máxima de subsecuencia contigua (Weiss,2013).

```
/**
     * Recursive maximum contiguous subsequence sum
     algorithm.
     * Finds maximum sum in subarray spanning a[left..
     right].
     * Does not attempt to maintain actual best sequence.
     */
     private static int maxSumRec( int [ ] a, int left,
     int right )
     {
          int maxLeftBorderSum = 0, maxRightBorderSum = 0;
          int leftBorderSum = 0, rightBorderSum = 0;
          int center = ( left + right ) / 2;

          if( left == right )  // Base case
               return a[ left ] > 0 ? a[ left ] : 0;
```

```
        int maxLeftSum  = maxSumRec( a, left, center );
        int maxRightSum = maxSumRec( a, center + 1,
        right );

        for( int i = center; i >= left; i-- )
        {
            leftBorderSum += a[ i ];
            if( leftBorderSum > maxLeftBorderSum )
                maxLeftBorderSum = leftBorderSum;
         }

        for( int i = center + 1; i <= right; i++ )
        {
            rightBorderSum += a[ i ];
            if( rightBorderSum > maxRightBorderSum )
                maxRightBorderSum = rightBorderSum;
        }

        return max3( maxLeftSum, maxRightSum,
            maxLeftBorderSum + maxRightBorderSum );
    }

    /**
    * Return maximum of three integers.
    */
    private static int max3( int a, int b, int c )
    {
        return a > b ? a > c ? a : c : b > c ? b : c;
    }

    /**
    * Driver for divide-and-conquer maximum contiguous
    * subsequence sum algorithm.
    */
    public static int maxSubSum4( int [ ] a )
    {
        return a.length > 0 ? maxSumRec( a, 0, a.length
        - 1 ) : 0;
    }
```

```
public static void getTimingInfo( int n, int alg )
{
    int [] test = new int[ n ];

    long startTime = System.currentTimeMillis( );;
    long totalTime = 0;

    int i;
    for( i = 0; totalTime < 4000; i++ )
    {
            for( int j = 0; j < test.length; j++ )
                test[ j ] = rand.nextInt( 100 ) -
                50;

            switch( alg )
            {
                case 1:
                    maxSubSum1( test );
                    break;
                case 2:
                    maxSubSum2( test );
                    break;
                case 3:
                    maxSubSum3( test );
                    break;
                case 4:
                    maxSubSum4( test );
                    break;
            }
            totalTime = System.currentTimeMillis(
            ) - startTime;
        }

        System.out.println( "Algorithm #" + alg + "\t"
            + "N = " + test.length
            + "\ttime = " + ( totalTime * 1000 / i )
                        + " microsec" );
    }
```

```
        private static Random rand = new Random( );

        /**
        * Simple test program.
        */
        public static void main( String [ ] args )
        {
            int a[ ] = { 4, -3, 5, -2, -1, 2, 6, -2 };
            int maxSum;

            maxSum = maxSubSum1( a );
            System.out.println( "Max sum is " + maxSum
            + "; it goes"
                 + ";" from " + seqStart + " to " + seqEnd
            maxSum = maxSubSum2( a );
            System.out.println( "Max sum is " + maxSum
            + "; it goes"
                 + ";" from " + seqStart + " to " + seqEnd
            maxSum = maxSubSum3( a );
            System.out.println( "Max sum is " + maxSum
                                 + "; it goes"
                 + ";" from " + seqStart + " to " + seqEnd
            maxSum = maxSubSum4( a );
            System.out.println( "Max sum is " + maxSum );

            // Get some timing info
            for( int n = 10; n <= 1000000; n *= 10 )
            for( int alg = 4; alg >= 1; alg-- )
            {
                if( alg == 1 && n > 50000 )
                    continue;
                getTimingInfo( n, alg );
            }
    }
}
```

Tema 3.
Tabla *Hash*

3.1 Conceptos básicos

Hashing es una técnica que se utiliza para identificar de forma **única** un objeto específico de un grupo de objetos similares. Algunos ejemplos de cómo se usa *hash* son:

- En las universidades, a cada alumno se le asigna un número de lista único que se puede usar para recuperar información sobre ellos.
- En las bibliotecas, a cada libro se le asigna un número único que se puede utilizar para determinar la información sobre el libro, como su posición exacta en la biblioteca o los usuarios a los que se ha emitido, etc.
- En ambos ejemplos, los estudiantes y los libros se clasificaron fragmentando la información y tomando un número único.

Por ejemplo, supóngase que se tiene un objeto y se desea asignar una clave para facilitar la búsqueda. Para almacenar el par clave-valor, se puede usar una matriz simple como una estructura de datos donde las claves (enteros) pueden usarse directamente como un índice para almacenar valores. Sin

embargo, en los casos en que las claves son grandes y no se pueden usar directamente como índice, debe usarse la función *hash*.

Cuando se emplea un *hash*ing, las claves grandes se convierten en claves pequeñas mediante el uso de funciones *hash*. Los valores se almacenan en una estructura de datos llamada tabla *hash*. La idea de *hash* es distribuir entradas (pares clave-valor) uniformemente en una matriz. A cada elemento se le asigna una clave (clave convertida). Al usar esa clave se puede acceder al elemento en un tiempo O(1). Usando la clave, el algoritmo (función *hash*) calcula un índice que sugiere dónde se puede encontrar o insertar una entrada.

El *Hashing* se implementa en dos pasos:

- Un elemento se convierte en un entero utilizando una función *hash*. Este elemento se puede usar como un índice para almacenar el elemento original, que se encuentra dentro de la tabla *hash*.
- El elemento se almacena en la tabla *hash*, donde se puede recuperar rápidamente con la clave *hash*.

```
hash = hashfunc (clave)

index = hash% array_size
```

En este método, el *hash* es independiente del tamaño de la matriz y luego se reduce a un índice (un número entre 0 y *array_size* - 1) utilizando el operador de módulo (%).

3.2 Función *hash*

Una **función *hash*** es cualquier función que se puede usar para "mapear" un conjunto de datos de un tamaño arbitrario a un conjunto de datos de un tamaño fijo, que se encuentra dentro de la tabla *hash*. Los valores devueltos por una función *hash* se denominan **valores *hash***, **códigos *hash***, **sumas *hash*** o simplemente ***hash*** (Weiss,2013).

Para lograr un buen mecanismo *hash*, es importante tener una buena función *hash* con los siguientes requisitos básicos:

1. Fácil de calcular: debe ser fácil de calcular y no debe convertirse en un algoritmo en sí mismo.
2. Distribución uniforme: debe proporcionar una distribución uniforme a través de la tabla *hash* y no debe dar lugar a la agrupación.
3. Menos colisiones: las colisiones ocurren cuando los pares de elementos se asignan al mismo valor *hash*. Estos deben ser evitados.

3.2.1 Necesidad de una buena función *hash*

Suponga que tiene que almacenar cadenas en una tabla *hash* utilizando la técnica de *hash*ing {"abcdef", "bcdefa", "cdefab", "defabc"}.

Para calcular el índice que almacenará las cadenas, se usará una función *hash* que establezca lo siguiente:

- El índice para una cadena específica será igual a la suma de los valores ASCII de los caracteres módulo 599.
- Como 599 es un número primo, reducirá la posibilidad de indexar diferentes cadenas (colisiones). Se recomienda que use números primos al usar módulo. Los valores ASCII de a, b, c, d, e, y f son 97, 98, 99, 100, 101 y 102, respectivamente. Como todas las cadenas contienen los mismos caracteres con diferentes permutaciones, la suma será 599.
- La función *hash* calculará el mismo índice para todas las cadenas y las cadenas se almacenarán en la tabla *hash*. Como el índice de todas las cadenas es el mismo, se puede crear una lista en ese índice e insertar todas las cadenas.

Para acceder a una cadena específica tomará el tiempo O (n) (donde n es el número de cadenas). Esto muestra que esta elección de la función *hash* no es buena.

Se probará una función *hash* diferente. El índice de una cadena específica será igual a la suma de los valores ASCII de los caracteres multiplicados por su orden respectivo en la cadena, después de lo cual se toma el módulo con 2069 (número primo).

String	*Hash* function	Index
abcdef	(971 + 982 + 993 + 1004 + 1015 + 1026)%2069	38
bcdefa	(981 + 992 + 1003 + 1014 + 1025 + 976)%2069	23
cdefab	(991 + 1002 + 1013 + 1024 + 975 + 986)%2069	14
defabc	(1001 + 1012 + 1023 + 974 + 985 + 996)%2069	11

3.2.2 Tabla *Hash*

Una *tabla hash* es una estructura de datos que se usa para almacenar pares claves- valores. Utiliza una función *hash* para calcular un índice en una matriz en la que se insertará o buscará un elemento. Al usar una buena función de *hash*, el *hash* puede funcionar bien. Bajo supuestos razonables, el tiempo promedio requerido para buscar un elemento en una tabla *hash* es O(1) (Weiss,2013).

Considérese la cadena S. Se requiere contar la frecuencia de todos los caracteres de esta cadena.

```
string S = “ababcd”
```

La forma más simple de hacer esto es iterar sobre todos los caracteres posibles y contar su frecuencia uno por uno. La complejidad temporal de este enfoque es O (26 * N) donde N es el tamaño de la cadena y hay 26 caracteres posibles. La siguiente función realiza esa actividad (Weiss,2013).

```
1.  void countFre(string S)
2.  {
3.  for(char c = 'a';c <= 'z';++c)
4.  {
      a.  int frequency = 0;
      b.  for(int i = 0;i < S.length();++i)
      c.  if(S[i] == c)
           i.  frequency++;
      d.  cout << c << ' ' << frequency << endl;
5.  }
6.  }
```

Se aplicará el *hashing* a este problema. Se implementa un arreglo de tamaño 26 y se crean claves *hash* para cada uno de los 26 caracteres generando los índices para el arreglo, esto a través de la *función hash*. Luego, se itera sobre la cadena aumentando el valor de la frecuencia en el índice correspondiente para cada carácter. La complejidad de este enfoque es O (N) donde N es el tamaño de la cadena (Weiss,2013).

```
1.  int Frequency[26];

2.  int hashFunc(char c)
3.  {
4.  return (c - 'a');
5.  }ç

6.  void countFre(string S)
7.  {
8.  for(int i = 0;i < S.length();++i)
9.  {
      a.  int index = hashFunc(S[i]);
      b.  Frequency[index]++;
10. }
11. for(int i = 0;i < 26;++i)
      a.  cout << (char)(i+'a') << ' ' << Frequency[i] << endl;
12. }
```

3.3 Técnicas de resolución de colisión

3.3.1 Cadena separada (*hashing* abierto)

El encadenamiento independiente es una de las técnicas de resolución de colisiones más comúnmente utilizadas. Por lo general, se implementa utilizando listas vinculadas. En el encadenamiento separado, cada elemento de la tabla *hash* es una lista vinculada. Para almacenar un elemento en la tabla *hash*, se debe insertar en una lista vinculada específica. Si hay una colisión (es decir, dos elementos diferentes tienen el mismo valor *hash*) se deben almacenar ambos elementos en la misma lista vinculada (Weiss,2013).

El costo de la búsqueda es la exploración de la lista vinculada seleccionada por la clave requerida. Si la distribución de las claves es suficientemente uniforme, entonces el costo promedio de una búsqueda depende únicamente del número promedio de claves por lista vinculada. Por esta razón, las tablas *hash* encadenadas se mantienen vigentes incluso cuando el número de entradas de la tabla (N) es mucho mayor que el número de **ranuras**.

Para el encadenamiento separado, el peor de los casos es cuando todas las entradas se insertan en la misma lista vinculada. El procedimiento de búsqueda puede tener que escanear todas sus entradas, por lo que el costo del peor de los casos es proporcional al número (N) de entradas en la tabla.

Se presenta a continuación un código de ejemplo para la implementación de esta técnica (Weiss,2013).

Suposición: La función *hash* devolverá un número entero de 0 a 19.

```
vector <string> hashTable[20];
     int hashTableSize=20;
```

Insertar:

```
void insert(string s)
{
          // Compute the index using Hash Function
     int index = hashFunc(s);
     // Insert the element in the linked list at the
     particular index
     hashTable[index].push_back(s);
}
```

Búsqueda:

```
void search(string s)
     {
          //Compute the index by using the hash function
          int index = hashFunc(s);
```

```
        //Search the linked list at that specific index
        for(int i = 0;i < hashTable[index].size();i++)
        {
            if(hashTable[index][i] == s)
            {
                cout << s << " is found!" << endl;
                return;
            }
        }
        cout << s << " is not found!" << endl;
    }
```

3.3.2 Sonda lineal (direccionamiento abierto o *hashing* cerrado)

En el direccionamiento abierto, en lugar de en listas vinculadas, todos los registros de entrada se **almacenan en la matriz misma.** Cuando se debe insertar una nueva entrada, el índice *hash* del valor *hash* se calcula y luego se examina la matriz (comenzando con el índice *hash*). Si la ranura en el índice *hash* está desocupada, entonces el registro de entrada se inserta en la ranura en el índice *hash*, de lo contrario avanza en alguna secuencia de prueba hasta que encuentra una ranura desocupada (Weiss,2013).

La secuencia de prueba continúa atravesando todas las entradas. En otras secuencias de pruebas, se pueden tener diferentes intervalos entre ranuras o pruebas de entrada sucesivas.

Al buscar una entrada, la matriz se escanea en la misma secuencia hasta que se encuentra el elemento objetivo o se encuentra una ranura no utilizada. Esto indica que no hay tal clave en la tabla. El nombre "direccionamiento abierto" se refiere al hecho de que la ubicación o dirección del elemento no está determinada por su valor *hash*.

La **prueba lineal** se establece cuando el intervalo entre las pruebas sucesivas es fijo (generalmente a 1). Supongamos que el índice *hash* para una entrada en particular es índex. La secuencia de sondeo para el sondeo lineal será:

```
index = index % hashTableSize
index = (index + 1) % hashTableSize
index = (index + 2) % hashTableSize
index = (index + 3) % hashTableSize
```

La implementación de esta técnica es mostrada como ejemplo el siguiente código (Weiss,2013).

Suposición:

- No hay más de 20 elementos en el conjunto de datos.

- La función *hash* devolverá un número entero de 0 a 19.
- El conjunto de datos debe tener elementos únicos.

```
string hashTable[21];
     int hashTableSize = 21;
```

Insertar:

```
void insert(string s)
     {
          //Compute the index using the hash function
          int index = hashFunc(s);
          //Search for an unused slot and if the index
              will exceed the hashTableSize then roll back
          while(hashTable[index] != "")
               index = (index + 1) % hashTableSize;
          hashTable[index] = s;
     }
```

Búsqueda:

```
void search(string s)
     {
          //Compute the index using the hash function
          int index = hashFunc(s);
          //Search for an unused slot and if the index
          will exceed the hashTableSize then roll back
          while(hashTable[index] != s and hashTable[index] != "")
               index = (index + 1) % hashTableSize;
          //Check if the element is present in the hash table
          if(hashTable[index] == s)
               cout << s << " is found!" << endl;
          else
               cout << s << " is not found!" << endl;
     }
```

3.3.3 Prueba cuadrática

La prueba cuadrática es similar a la prueba lineal y la única diferencia es el intervalo entre pruebas sucesivas o ranuras de entrada. Aquí, cuando la ranura en un índice *hash* para un registro de entrada ya está ocupada, debe comenzar a atravesar hasta que encuentre una ranura desocupada. El intervalo entre ranuras se calcula sumando el valor sucesivo de un polinomio arbitrario en el índice *hash* original (Weiss,2013).

Supongamos que el índice *hash* para una entrada es índex y hay una ranura ocupada. La secuencia de la prueba será la siguiente:

```
index = index % hashTableSize
index = (index + 12) % hashTableSize
index = (index + 22) % hashTableSize
index = (index + 32) % hashTableSize
```

A continuación, se muestra la implementación de ejemplo (Weiss,2013):

Suposición:

- No hay más de 20 elementos en el conjunto de datos.
- La función *hash* devolverá un número entero de 0 a 19.
- El conjunto de datos debe tener elementos únicos.

```
string hashTable[21];
  int hashTableSize = 21;
```

Insertar:

```
void insert(string s)
     {
          //Compute the index using the hash function
          int index = hashFunc(s);
          //Search for an unused slot and if the index will
          exceed the hashTableSize roll back
          int h = 1;
          while(hashTable[index] != "")
          {
               index = (index + h*h) % hashTableSize;
                    h++;
               }
               hashTable[index] = s;
          }
```

Búsqueda:

```
void search(string s)
     {
          //Compute the index using the Hash Function
          int index = hashFunc(s);
          //Search for an unused slot and if the index will
          exceed the
          hashTableSize roll back
```

```
        int h = 1;
        while(hashTable[index] != s and hashTable[index]
        {
            index = (index + h*h) % hashTableSize;
                h++;
        }
        //Is the element present in the hash table
        if(hashTable[index] == s)
            cout << s << " is found!" << endl;
        else
            cout << s << " is not found!" << endl;
    }
```

3.3.4 Doble *hash*

El doble *hash* es similar a la prueba lineal y la única diferencia es el intervalo entre pruebas sucesivas. Aquí, el intervalo entre las sondas se calcula mediante el uso de dos funciones *hash* (Weiss,2013):

Suponga que el índice *hash* para un registro de entrada es un índice que se calcula mediante una función *hash* y el espacio en ese índice ya está ocupado. Se debe comenzar una secuencia de prueba específica para buscar una ranura desocupada. La secuencia de prueba será:

```
index = (index + 1 * indexH) % hashTableSize;
index = (index + 2 * indexH) % hashTableSize;
```

En este caso, *indexH* es el valor *hash* que se calcula mediante otra función *hash*.

Para implementar esta técnica se presenta el siguiente ejemplo (Weiss,2013):

Suposición:

- No hay más de 20 elementos en el conjunto de datos.
- Las funciones *hash* devolverán un número entero de 0 a 19.
- El conjunto de datos debe tener elementos únicos.

```
string hashTable[21];
  int hashTableSize = 21;
```

Insertar:

```
    void insert(string s)
        {
            //Compute the index using the hash function1
            int index = hashFunc1(s);
            int indexH = hashFunc2(s);
```

```
            //Search for an unused slot and if the index
            exceeds the
            hashTableSize roll back
            while(hashTable[index] != "")
                index = (index + indexH) % hashTableSize;
            hashTable[index] = s;
        }
```

Búsqueda:

```
    void search(string s)
    {
        //Compute the index using the hash function
        int index = hashFunc1(s);
        int indexH = hashFunc2(s);
        //Search for an unused slot and if the index exceeds
        the hashTableSize
        roll back
        while(hashTable[index] != s and hashTable[index] != "")
            index = (index + indexH) % hashTableSize;
        //Is the element present in the hash table
        if(hashTable[index] == s)
            cout << s << " is found!" << endl;
        else
            cout << s << " is not found!" << endl;
    }
```

Algunas aplicaciones de las tablas *hash* son:

- Conjuntos asociativos: las tablas *hash* se usan comúnmente para implementar muchos tipos de tablas en memoria. Se utilizan para implementar matrices asociativas (matrices cuyos índices son cadenas arbitrarias u otros objetos complicados).
- Indización de la base de datos: las tablas *hash* también se pueden usar como estructuras de datos basadas en disco e índices de bases de datos (como en *dbm*).
- Cachés: las tablas *hash* pueden utilizarse para implementar cachés, es decir, tablas de datos auxiliares que se utilizan para acelerar el acceso a los datos, que se almacenan principalmente en medios más lentos.
- Representación de objetos: varios lenguajes dinámicos, como *Perl, Python, JavaScript y Ruby,* utilizan tablas *hash* para implementar objetos.
- Las funciones *hash* se usan en varios algoritmos para hacer que su computación sea más rápida

Tema 4.
Árboles

Un árbol es una estructura de datos donde los objetos están organizados en términos de **relaciones jerárquicas.** Se puede decir que un árbol es un conjunto de elementos no vacíos, donde uno de estos elementos se denomina raíz y los elementos restantes (que pueden o no estar presentes) se dividen más en subárboles.

Cada elemento de datos de un árbol se almacena en una estructura llamada **nodo.** El *nodo superior* o de inicio del árbol se denomina *nodo raíz*. Todos los nodos están vinculados y forman *subárboles jerárquicos* que comienzan con el nodo raíz. La estructura de datos de árbol es útil en casos donde la representación lineal de datos no es suficiente, como por ejemplo, en la creación de un **árbol genealógico.**

A cada círculo de la jerarquía se le conoce como un nodo y a cada línea como *arista*. Los nodos "19", "21", "14" están debajo del nodo "7" y están directamente conectados a él. A estos nodos se les conoce como descendientes directos (nodos secundarios) del nodo "7", y el nodo "7" es el nodo padre de esos nodos. De la misma manera "1", "12" y "31" son hijos del nodo "19" y "19" es el padre de esos nodos. Intuitivamente podemos decir que "21" es hermano de "19", porque ambos son hijos de "7" (lo contrario también es verdad - "19" es hermano de "21"). Para "1", "12" "31", "23" y "6", el nodo "7"

les precede en la jerarquía, por lo que es su antecesor o padre indirecto, y a estos nodos se les llama sus *descendientes*. Se le llama raíz al nodo sin padre. En este ejemplo, es el nodo "7". Se le llama hoja al nodo sin nodos secundarios. En este ejemplo: "1", "12", "31", "21", "23" y "6" (Weiss,2013).

Los nodos internos son los nodos que no son hoja o raíz (es decir, todos los nodos, que tienen padre y al menos un hijo). Dichos nodos son "19" y "14". Se le llama ruta a la secuencia de nodos conectados con las aristas, en las cuales no hay repetición de nodos. Ejemplo de ruta es la secuencia "1", "19", "7" y "21". La secuencia "1", "19" y "23" no es una ruta, porque "19" y "23" no están conectados (Weiss,2013).

La longitud de la ruta es la cantidad de aristas que conectan la secuencia de nodos en la ruta. En realidad, es igual a la cantidad de nodos en la ruta menos 1. La longitud para la ruta ("1", "19", "7" y "21") es tres. Se le llama profundidad de un nodo a la longitud de la ruta desde la raíz a cierto nodo. En este ejemplo el nodo "7", como raíz, tiene profundidad cero, "19" tiene profundidad uno y "23" profundidad dos. La altura del árbol es la profundidad máxima de todos sus nodos. En este ejemplo, la altura del árbol es 2 (Weiss,2013).

El grado de nodo es el número de hijos directos del nodo dado. El grado de "19" y "7" es tres, pero el grado de "14" es dos. Las hojas tienen grado cero.

Considerando que un árbol es una estructura de datos recursiva, que consiste en nodos, conectados con aristas, se puede afirmar lo siguiente:

1. Cada nodo puede tener 0 o más descendientes directos (hijos).
2. Cada nodo tiene como máximo un padre. Solo hay un nodo especial sin padre: la raíz (si el árbol no está vacío).
3. Todos los nodos son accesibles desde la raíz, es decir, existe una ruta desde la raíz hasta cada nodo en el árbol.

4.1 Implementación de árboles en *java*

El siguiente código implementa un árbol dinámicamente, el cual contendrá números dentro de sus nodos, y cada nodo tendrá una lista de cero o más hijos. Cada nodo se define de forma recursiva consigo mismo. Cada nodo del árbol (*TreeNode* <*T*>) contiene una lista de elementos secundarios que son nodos (*TreeNode* <*T*>). El árbol en sí es otra clase *Tree* <*T*> que puede estar vacía o puede tener un nodo raíz. *Tree* <*T*> implementa operaciones básicas sobre árboles como construcción y recorrido (Weiss,2013).

```
1. using System;
2. using System.Collections.Generic;

3. /// <summary>Represents a tree node</summary>
4. /// <typeparam name="T">the type of the values in nodes
5. /// </typeparam>
```

```
6.  public class TreeNode<T>
7.  {
8.  // Contains the value of the node
9.  private T value;

10. // Shows whether the current node has a parent or not
11. private bool hasParent;

12. // Contains the children of the node (zero or more)
13. private List<TreeNode<T>> children;

14. /// <summary>Constructs a tree node</summary>
15. /// <param name="value">the value of the node</param>
16. public TreeNode(T value)
17. {
18. if (value == null)
19. {

      a.  throw new ArgumentNullException(
      b.  "Cannot insert null value!");
20. }
21. this.value = value;
22. this.children = new List<TreeNode<T>>();
23. }

24. /// <summary>The value of the node</summary>
25. public T Value
26. {
27. get
28. {
      a.  return this.value;
29. }
30. set
31. {
      a.  this.value = value;
32. }
33. }

34. /// <summary>The number of node's children</summary>
```

```
35. public int ChildrenCount
36. {
37. get
38. {
     a.  return this.children.Count;
39. }
40. }

41. /// <summary>Adds child to the node</summary>
42. /// <param name="child">the child to be added</param>
43. public void AddChild(TreeNode<T> child)
44. {
45. if (child == null)
46. {
a.  throw new ArgumentNullException(
b.  "Cannot insert null value!");
47. }

48. if (child.hasParent)
49. {
     a.  throw new ArgumentException(
     b.  "The node already has a parent!");
50. }

51. child.hasParent = true;
52. this.children.Add(child);
53. }

54. /// <summary>
55. /// Gets the child of the node at given index
56. /// </summary>
57. /// <param name="index">the index of the desired
    child</param>
58. /// <returns>the child on the given position</returns>
59. public TreeNode<T> GetChild(int index)
60. {
61. return this.children[index];
62. }
63. }
```

```
64. /// <summary>Represents a tree data structure
    </summary>
65. /// <typeparam name="T">the type of the values in the
66. /// tree</typeparam>
67. public class Tree<T>
68. {
69. // The root of the tree
70. private TreeNode<T> root;

71. /// <summary>Constructs the tree</summary>
72. /// <param name="value">the value of the node</param>
73. public Tree(T value)
74. {
75. if (value == null)
76. {
      a. throw new ArgumentNullException(
      b. "Cannot insert null value!");
77. }

78. this.root = new TreeNode<T>(value);
79. }

80. /// <summary>Constructs the tree</summary>
81. /// <param name="value">the value of the root node</param>
82. /// <param name="children">the children of the root
83. /// node</param>
84. public Tree(T value, params Tree<T>[] children)
85. : this(value)
86. {
87. foreach (Tree<T> child in children)
88. {
      a. this.root.AddChild(child.root);
89. }
90. }

91. /// <summary>
92. /// The root node or null if the tree is empty
93. /// </summary>
94. public TreeNode<T> Root
```

```
95. {
96. get
97. {
     a.  return this.root;
98. }
99. }

100.    /// <summary>Traverses and prints tree in
101.    /// Depth-First Search (DFS) manner</summary>
102.    /// <param name="root">the root of the tree to be
103.    /// traversed</param>
104.    /// <param name="spaces">the spaces used for
105.    /// representation of the parent-child relation</param>
106.    private void PrintDFS(TreeNode<T> root, string
spaces)
107.    {
108.    if (this.root == null)
109.    {
     a.  return;
110.    }

111.    Console.WriteLine(spaces + root.Value);

112.    TreeNode<T> child = null;
113.    for (int i = 0; i < root.ChildrenCount; i++)
114.    {
     a.  child = root.GetChild(i);
     b.  PrintDFS(child, spaces + "   ");
115.    }
116.    }

117.    /// <summary>Traverses and prints the tree in
118.    /// Depth-First Search (DFS) manner</summary>
119.    public void TraverseDFS()
120.    {
121.    this.PrintDFS(this.root, string.Empty);
122.    }
123.    }

124.    /// <summary>
```

```
125.    /// Shows a sample usage of the Tree<T> class
126.    /// </summary>
127.    public static class TreeExample
128.    {
129.    static void Main()
130.    {
131.    // Create the tree from the sample
132.    Tree<int> tree =
     a.  new Tree<int>(7,
     b.  new Tree<int>(19,
          i.  new Tree<int>(1),
          ii. new Tree<int>(12),
          iii.      new Tree<int>(31)),
     c.  new Tree<int>(21),
     d.  new Tree<int>(14,
          i.  new Tree<int>(23),
          ii. new Tree<int>(6))
     e.  );

133.    // Traverse and print the tree using
        Depth-First-Search
134.    tree.TraverseDFS();

135.    // Console output:
136.    // 7
137.    //    19
138.    //    1
139.    //    12
140.    //    31
141.    //    21
142.    //    14
143.    //    23
144.    //    6
145.    }
146.    }
```

En este ejemplo, se tiene una clase *Tree* <*T*>, que implementa el árbol real. También se tiene una clase *TreeNode* <*T*>, que representa un único nodo del árbol. Las funciones asociadas con el nodo, como crear un nodo, agregar un nodo secundario a este nodo y obtener el número de hijos, se implementan en el nivel de *TreeNode* <*T*>.

El resto de la funcionalidad (realizar un recorrido atravesando el árbol, por ejemplo) se implementa en el nivel de *Tree <T>*. La división lógica de la funcionalidad entre las dos clases hace que la implementación sea más flexible. La razón por la que se divide la implementación en dos clases es que algunas operaciones son típicas para cada nodo por separado (agregando un elemento secundario, por ejemplo), mientras que otras son para todo el árbol (buscando un nodo por su número). En esta variante de la implementación, el árbol es una clase que conoce su raíz y cada nodo conoce sus elementos secundarios. En esta implementación se puede tener un árbol vacío (cuando *root = null).*

Aquí hay algunos detalles sobre la implementación *TreeNode <T>*. Cada nodo del árbol se compone de un valor de campo privado y una lista de hijos - *childrens*. La lista de hijos consta de elementos del mismo tipo. De esta forma, cada nodo contiene una lista de referencia a sus hijos directos. También hay propiedades públicas para acceder a los valores de los campos del nodo. Los métodos que pueden invocarse desde un código fuera de la clase son:

- *AddChild (TreeNode <T> child)* - agrega un hijo
- *TreeNode <T> GetChild (int index)* - devuelve un hijo por índice dado
- *ChildrenCount* - devuelve el número de hijos de cierto nodo

Para cumplir la condición de que cada nodo tenga solo un padre, hemos definido el campo privado *hasParent*, que determina si este nodo tiene *parent* o no. Esta información se usa solo dentro de la clase y la necesitamos en el método *AddChild (Tree <T> child)*. Dentro de este método comprobamos si el nodo que se va a agregar ya tiene el elemento primario y, si es así, lo lanzamos y hacemos una excepción, diciendo que esto es imposible. En la clase *Tree <T>* solo se tiene la propiedad *TreeNode <T> Root*, que devuelve la raíz del árbol.

4.2 Recorrido de un árbol

Existen múltiples algoritmos para recorrer un árbol. Los dos principales son: **DFS (búsqueda en profundidad) y BFS (búsqueda en anchura).**

El algoritmo **Depth-First-Search** tiene como objetivo visitar exactamente cada uno de los nodos del árbol. Dicha visita de todos los nodos se llama recorrido transversal del árbol.

El **algoritmo DFS** se inicia desde un nodo dado y va tan profundo en la jerarquía de árbol como puede. Cuando llega a un nodo que no tiene hijos que visitar o todos han sido visitados, regresa al nodo anterior. Se puede describir el algoritmo de búsqueda en profundidad por los siguientes pasos simples:

1. Atravesar el nodo actual.
2. Recorrer secuencial y recursivamente cada uno de los nodos secundarios de los nodos actuales (atraviesa los subárboles del nodo actual). Esto se puede hacer mediante una llamada recursiva al mismo método para cada nodo secundario.

En la clase *Tree <T>* se implementa el método *TraverseDFS ()*, que llama al método privado *PrintDFS (TreeNode <T> root, string spaces),* que recorre el árbol en profundidad e imprime en la salida estándar sus elementos en el diseño de árbol usando desplazamiento correcto (agregando espacios).

4.3 Creación de árboles

Para facilitar la creación de un árbol, se define un constructor especial, que toma para los parámetros de entrada un valor de nodo y una lista de sus subárboles. Eso permite dar cualquier número de **argumentos** de tipo *Tree <T>* (subárboles). Se usa exactamente el mismo constructor para crear el árbol de ejemplo.

4.3.1 Recorrido de los directorios de la unidad de disco duro

Los directorios en su disco duro son en realidad una estructura jerárquica, que puede representarse como un árbol, ya que se tienen carpetas (nodos de árbol) que pueden tener otras carpetas y archivos secundarios (que también son nodos de árbol).

Como se sabe, de manera cotidiana se crean carpetas en el disco duro que pueden contener subcarpetas y archivos. Las subcarpetas también pueden contener subcarpetas, y así sucesivamente hasta que alcances cierto límite máximo de profundidad.

Se puede acceder al árbol de directorios del sistema de archivos a través de la funcionalidad de compilación *.NET*: la clase *System.IO.DirectoryInfo.* No está presente como una estructura de datos, pero se pueden obtener las subcarpetas y los archivos de cada directorio, de modo que se puede atravesar el árbol del sistema de archivos utilizando un algoritmo de recorrido de árbol estándar, como la búsqueda en profundidad (DFS).

El siguiente ejemplo ilustra cómo se puede recorrer de forma recursiva la estructura del árbol de una carpeta determinada (utilizando *Depth-First-Search)* e imprimir en la salida estándar su contenido (Weiss,2013).

```
1.  using System;
2.  using System.IO;

3.  /// <summary>
4.  /// Sample class, which traverses recursively given
    directory
5.  /// based on the Depth-First-Search (DFS) algorithm
6.  /// </summary>
7.  public static class DirectoryTraverserDFS
8.  {
9.  /// <summary>
10. /// Traverses and prints given directory recursively
```

```
11. /// </summary>
12. /// <param name="dir">the directory to be traversed
    </param>
13. /// <param name="spaces">the spaces used for
    representation
14. /// of the parent-child relation</param>
15. private static void TraverseDir(DirectoryInfo dir,
16. string spaces)
17. {
18. // Visit the current directory
19. Console.WriteLine(spaces + dir.FullName);

20. DirectoryInfo[] children = dir.GetDirectories();

21. // For each child go and visit its sub-tree
22. foreach (DirectoryInfo child in children)
23. {
      a.  TraverseDir(child, spaces + "  ");
24. }
25. }
26. /// <summary>
27. /// Traverses and prints given directory recursively
28. /// </summary>
29. /// <param name="directoryPath">the path to the
    directory
30. /// which should be traversed</param>
31. static void TraverseDir(string directoryPath)
32. {
33. TraverseDir(new DirectoryInfo(directoryPath),
      a.  string.Empty);
34. }

35. static void Main()
36. {
37. TraverseDir("C:\\");
38. }
39. }
```

Como se puede observar, el algoritmo de recorrido recursivo del contenido del directorio es el mismo que se utiliza para el árbol. Aquí se puede ver parte del resultado del recorrido:

```
C:\
    C:\Config.Msi
    C:\Documents and Settings
        C:\Documents and Settings\Administrator
            C:\Documents and Settings\Administrator\.ARIS70
            C:\Documents and Settings\Administrator\.jindent
            C:\Documents and Settings\Administrator\.nbi
                C:\Documents and Settings\Administrator\.nbi\
                  downloads
                C:\Documents and Settings\Administrator\.nbi\log
                C:\Documents and Settings\Administrator\.nbi\cache
                C:\Documents and Settings\Administrator\.nbi\tmp
                C:\Documents and Settings\Administrator\.nbi\wd
            C:\Documents and Settings\Administrator\.netbeans
                C:\Documents and Settings\Administrator\
                  .netbeans\6.0
```

Hay que considerar que el programa anterior puede bloquearse con *UnauthorizedAccessException* en caso de que no se tengan permisos de acceso para algunas carpetas en el disco duro. Esto es típico para algunas instalaciones de Windows, por lo que puede comenzar el recorrido desde otro directorio.

4.3.2 Breath-First-Search (BFS)

Es un algoritmo para atravesar estructuras de datos ramificados (como árboles y grafos). El algoritmo *BFS* primero atraviesa el nodo de inicio, luego todos sus hijos directos, luego sus hijos directos y así sucesivamente (Weiss,2013).

El algoritmo *Breath-First-Search* (BFS) consta de los siguientes pasos:

1. Coloca el nodo de inicio en la cola Q.
2. Mientras Q no esté vacío, repita los dos pasos siguientes:
    - Quitar de la cola el siguiente nodo v de Q e imprimirlo.
    - Agregue todos los hijos de v en la cola.

El algoritmo *BFS* es muy simple y siempre recorre primero los nodos que están más cerca del nodo de inicio, y luego los más distantes y así sucesivamente hasta que llega más lejos. El algoritmo *BFS* es muy utilizado en la resolución de problemas, por ejemplo, para encontrar el camino más corto en un laberinto.

A continuación, se muestra una implementación de ejemplo de algoritmos BFS que imprime todas las carpetas en el sistema de archivos (Weiss,2013):

```
1.  using System;
2.  using System.Collections.Generic;
3.  using System.IO;

4.  /// <summary>
5.  /// Sample class, which traverses given directory
6.  /// based on the Breath-First-Search (BFS) algorithm
7.  /// </summary>
8.  public static class DirectoryTraverserBFS
9.  {
10. /// <summary>
11. /// Traverses and prints given directory with BFS
12. /// </summary>
13. /// <param name="directoryPath">the path to the directory ///
    which should be traversed</param>
14. static void TraverseDir(string directoryPath)
15. {
16. Queue<DirectoryInfo> visitedDirsQueue =
      a.  new Queue<DirectoryInfo>();
17. visitedDirsQueue.Enqueue(new DirectoryInfo(directoryPath));
18. while (visitedDirsQueue.Count > 0)
19. {
      a.  DirectoryInfo currentDir = visitedDirsQueue Dequeue();
      b.  Console.WriteLine(currentDir.FullName);

      c.  DirectoryInfo[] children = currentDir.GetDirectories();
      d.  foreach (DirectoryInfo child in children)
      e.  {
      f.  visitedDirsQueue.Enqueue(child);
g.  }
20. }
21. }

22. static void Main()
23. {
24. TraverseDir(@"C:\");
25. }
26. }
```

Si se inicia el programa para recorrer el disco duro local, se podría observar que BFS visita primero los directorios más cercanos a la raíz (profundidad 1), luego las carpetas a profundidad 2, luego profundidad 3 y así sucesivamente. Aquí hay una salida de muestra del programa (Weiss,2013):

C:\
C:\Config.Msi
C:\Documents and Settings
C:\Inetpub
C:\Program Files
C:\RECYCLER
C:\System Volume Information
C:\WINDOWS
C:\wmpub
C:\Documents and Settings\Administrator
C:\Documents and Settings\All Users
C:\Documents and Settings\Default User

4.4 Árboles Binarios

En esta sección se mostrará un tipo específico de árbol, conocido como árbol binario. Este tipo de árbol resulta ser muy útil en la programación. La terminología para árboles también es válida para árboles binarios, pero existen características específicas sobre la estructura.

Un Árbol Binario es un árbol cuyos nodos tienen un grado igual o menor que 2. Como los hijos de cada nodo son como mucho 2, se les conoce como hijo izquierdo y como hijo derecho. Son las raíces del subárbol izquierdo y del subárbol derecho del nodo padre. Algunos nodos pueden tener solo un hijo izquierdo o solo uno derecho, no ambos. Algunos nodos pueden no tener hijos y se llaman hojas.

El árbol binario se puede definir recursivamente de la siguiente manera: un solo nodo es un árbol binario y puede tener hijos izquierdo y derecho que también son árboles binarios.

4.4.1 Recorridos en un árbol binario

Un recorrido en un árbol binario consiste en visitar todos sus vértices o nodos, de tal manera que cada vértice se visite una sola vez.

Se distinguen tres tipos de recorrido: INORDEN, POSORDEN Y PREORDEN.

En cada recorrido se tiene en cuenta la posición de la raíz y que siempre se debe ejecutar primero el hijo izquierdo y luego el derecho.

Recorrido IN-ORDEN. Este recorrido primero se recorre el subárbol izquierdo, segundo visita la raíz y por último, va al subárbol derecho. En resumen: hijo izquierdo-raíz-hijo derecho.

Recorrido PRE-ORDEN. Este recorrido primero visita la raíz; segundo recorre el subárbol izquierdo y por último va a subárbol derecho. En resumen: raíz-hijo izquierdo-hijo derecho.

Recorrido POS-ORDEN. Primero recorre el subárbol izquierdo; segundo, recorre el subárbol derecho y por último, visita la raíz. En resumen: hijo izquierdo– hijo derecho-raíz.

El siguiente ejemplo muestra una implementación del árbol binario, que se recorre utilizando el esquema recursivo IN-ORDEN (Weiss,2013).

```
1.  using System;
2.  using System.Collections.Generic;

3.  /// <summary>Represents a binary tree</summary>
4.  /// <typeparam name="T">Type of values in the tree
    </typeparam>
5.  public class BinaryTree<T>
6.  {
7.  /// <summary>The value stored in the curent node
    </summary>
8.  public T Value { get; set; }

9.  /// <summary>The left child of the current node
    </summary>
10. public BinaryTree<T> LeftChild { get; private set; }

11. /// <summary>The right child of the current node
    </summary>
12. public BinaryTree<T> RightChild { get; private set; }

13. /// <summary>Constructs a binary tree</summary>
14. /// <param name="value">the value of the tree node
    </param>
15. /// <param name="leftChild">the left child of the
    tree</param>
16. /// <param name="rightChild">the right child of the
    tree
17. /// </param>
18. public BinaryTree(T value,
19. BinaryTree<T> leftChild, BinaryTree<T> rightChild)
20. {
21. this.Value = value;
22. this.LeftChild = leftChild;
23. this.RightChild = rightChild;
```

```
24. }

25. /// <summary>Constructs a binary tree with no children
26. /// </summary>
27. /// <param name="value">the value of the tree node</param>
28. public BinaryTree(T value) : this(value, null, null)
29. {
30. }

31. /// <summary>Traverses the binary tree in pre-order</summary>
32. public void PrintInOrder()
33. {
34. // 1. Visit the left child
35. if (this.LeftChild != null)
36. {
     a. this.LeftChild.PrintInOrder();
37. }

38. // 2. Visit the root of this sub-tree
39. Console.Write(this.Value + " ");
40. // 3. Visit the right child
41. if (this.RightChild != null)
42. {
a. this.RightChild.PrintInOrder();
43. }
44. }
45. }

46. /// <summary>
47. /// Demonstrates how the BinaryTree<T> class can be used
48. /// </summary>
49. public class BinaryTreeExample
50. {
51. static void Main()
52. {
53. // Create the binary tree from the sample
54. BinaryTree<int> binaryTree =
     a. new BinaryTree<int>(14,
          i. new BinaryTree<int>(19,
```

```
            ii. new BinaryTree<int>(23),
            iii.      new BinaryTree<int>(6,
1.  new BinaryTree<int>(10),
2.  new BinaryTree<int>(21))),
      iv. new BinaryTree<int>(15,
      v.  new BinaryTree<int>(3),
      vi. null));

55. // Traverse and print the tree in in-order manner
56. binaryTree.PrintInOrder();
57. Console.WriteLine();

58. // Console output:
59. // 23 19 10 6 21 14 3 15
60. }
61. }
```

Esta implementación del árbol binario es ligeramente diferente de la del árbol ordinario y se simplifica significativamente.

Se tiene una definición de clase recursiva *BinaryTree <T>*, que contiene un valor y nodos secundarios izquierdo y derecho que son del mismo tipo *BinaryTree <T>*. Se tienen exactamente dos nodos secundarios (izquierdo y derecho) en lugar de la lista de hijos.

El método *PrintInOrder ()* funciona recursivamente con el algoritmo DFS. Atraviesa cada nodo en "orden" (primero el hijo izquierdo, luego el nodo en sí, luego el hijo derecho). El algoritmo de recorrido DFS realiza los siguientes pasos:

1. Llamada recursiva para atravesar el subárbol izquierdo del nodo dado.
2. Recorre el nodo en sí mismo (imprime su valor).
3. Llamada recursiva para atravesar el subárbol correcto.

Tema 5.
Cola de prioridad y montículo binario

Muchas aplicaciones requieren que se procesen elementos que tienen valores en orden, pero no necesariamente en orden completo y no necesariamente todos a la vez. A menudo, se recolecta un conjunto de ítems, luego se procesa el que tiene el valor más grande, luego tal vez se recolecten más ítems, luego se procesa el que tiene el valor más grande actual, y así sucesivamente. Un tipo de datos apropiado en dicho entorno admite dos operaciones: eliminar el máximo e insertar, o bien, eliminar el mínimo e insertar. Tal tipo de datos se denomina **cola de prioridad** (Weiss,2013).

Una **cola de prioridad** es una variación importante de una cola. Una cola de prioridad actúa como una cola en la que un elemento se quita de la parte frontal. En una cola de prioridad, el orden lógico de los elementos dentro de una cola siempre está determinado por su prioridad. Los elementos de mayor prioridad se encuentran en la parte delantera de la cola y los elementos de menor prioridad están en la parte posterior. Por lo tanto, cuando se coloca un elemento en una cola de prioridad, el nuevo elemento puede moverse hasta el frente.

El siguiente es un código que muestra la utilización de las colas de prioridad (Weiss,2013).

```
/*************************************************
 *  Compilation:  javac TopM.java
```

```
 *  Execution:    java TopM m < input.txt
 *  Dependencies: MinPQ.java Transaction.java StdIn.java
    StdOut.java
 *  Data files:   https://algs4.cs.princeton.edu/24pq/
    tinyBatch.txt
 *
 *  Given an integer m from the command line and an
    input stream where
 *  each line contains a String and a long value, this
    MinPQ client
 *  prints the m lines whose numbers are the highest.
 *
 *  % java TopM 5 < tinyBatch.txt
 *  Thompson    2/27/2000  4747.08
 *  vonNeumann  2/12/1994  4732.35
 *  vonNeumann  1/11/1999  4409.74
 *  Hoare       8/18/1992  4381.21
 *  vonNeumann  3/26/2002  4121.85
 *
 ************************************************************/

/**
 * The {@code TopM} class provides a client that reads a sequence
of
 *  transactions from standard input and prints the <em>m</em> largest
ones
 *  to standard output. This implementation uses a {@link MinPQ}
    of size
 *  at most <em>m</em> + 1 to identify the <em>M</em> largest
transactions
 *  and a {@link Stack} to output them in the proper order.
 *  <p>
 *  For additional documentation, see <a href="https://
    lgs4.cs.princeton.
edu/24pq">Section 2.4</a>
 *  of <i>Algorithms, 4th Edition</i> by Robert Sedgewick
    and Kevin Wayne.
 *
 *  @author Robert Sedgewick
 *  @author Kevin Wayne
```

```
 */
1.  public class TopM {

2.  // This class should not be instantiated.
3.  private TopM() { }

4.  /**
5.  Reads a sequence of transactions from standard input; takes a
6.  command-line integer m; prints to standard output the m largest
7.  transactions in descending order.
8.  *
9.  @param args the command-line arguments
10. */
11. public static void main(String[] args) {
12. int m = Integer.parseInt(args[0]);
13. MinPQ<Transaction> pq = new MinPQ<Transaction>(m+1);

14. while (StdIn.hasNextLine()) {
      a.  // Create an entry from the next line and put on the PQ.
      b.  String line = StdIn.readLine();
      c.  Transaction transaction = new Transaction(line);
      d.  pq.insert(transaction);

      e.  // remove minimum if m+1 entries on the PQ
      f.  if (pq.size() > m)
      g.  pq.delMin();
15. }   // top m entries are on the PQ

16. // print entries on PQ in reverse order
17. Stack<Transaction> stack = new Stack<Transaction>();
18. for (Transaction transaction : pq)
a.  stack.push(transaction);
19. for (Transaction transaction : stack)
a.  StdOut.println(transaction);
20. }
21. }
```

Este programa es un cliente de colas de prioridad que toma un argumento de línea de comandos M, lee las transacciones de la entrada estándar e imprime las M mayores transacciones.

Para implementar una cola de prioridad se pueden utilizar las funciones y listas de clasificación. Sin embargo, insertar en una lista es de orden O(n) y ordenar una lista es O(*nlogn*). Esto se puede mejorar. La forma clásica de implementar una cola de prioridad es usar una estructura de datos llamada montículo binario.

5.1 Montículo binario

Un montículo binario permitirá tanto poner en cola como quitar elementos en el orden O*(logn)*. El montículo binario es interesante de estudiar porque cuando se realiza el diagrama, el montículo es muy parecido a un árbol, pero cuando se implementa, se utiliza una sola lista como representación interna.

El montículo binario tiene dos variaciones comunes: el montículo mínimo, en el que el valor clave más pequeño siempre está en la parte delantera, y el montículo máximo, en el que el valor clave más grande siempre está en la parte delantera.

Por lo tanto, podemos decir que un montículo binario es un árbol binario tal que,

1. El elemento máximo se encuentra en la raíz.
2. Un montículo de n elementos tiene altura `h = □log2 (n+1)□.`

Se representan árboles binarios completos secuencialmente dentro de una matriz al poner los nodos con orden de nivel, con la raíz en la posición 1, sus hijos en las posiciones 2 y 3, sus hijos en las posiciones 4, 5, 6 y 7, y así sucesivamente.

En un montículo binario, el padre del nodo en la posición k está en la posición k / 2; y, por el contrario, los dos hijos del nodo en la posición k se encuentran en las posiciones 2k y 2k+1. Podemos desplazarnos hacia arriba y hacia abajo haciendo aritmética simple en los índices de matriz: para mover el árbol desde [k] se establece k a k / 2; para bajar al árbol se estable k a 2*k o 2*(k + 1).

5.1.1 Algoritmos en montículos binarios

Se representa un montículo binario de tamaño n en el conjunto privado *pq* [ ] de longitud n+1, con *pq* [0] sin usar y el montículo en *pq* [1] a través de *pq* [n]. Se accede a las claves solo a través de las funciones de ayuda privadas *less()* y *exch()*. Las operaciones de montículo realizan primero una modificación simple, que podría violar la condición del montículo, y luego viajarán a través del montículo, modificando el montículo según sea necesario para garantizar que la condición se satisfaga en todas partes. Este proceso se conoce como *reheapifying*, o restauración el orden del montículo.

En este sentido se muestran dos situaciones: *Bottom-up reheapify*, conocido como nadar o flotar, y *Top-down heapify*, conocido como bajar o hundir (Weiss,2013).

1. ***Bottom-up reheapify***

Si se viola la orden de pila porque la clave de un nodo se vuelve más grande que la clave de padres de ese nodo, entonces se puede avanzar hacia la reparación de la violación intercambiando el nodo con su padre. Después del intercambio, el nodo es más grande que sus dos hijos (uno es el padre antiguo y el otro es más pequeño que el padre anterior porque era hijo de ese nodo) pero el nodo puede ser aún mayor que su padre. Se puede arreglar esa violación de la misma manera, y así sucesivamente, moviéndose hacia arriba hasta llegar a un nodo con una clave más grande, o a la raíz.

2. ***Top-down heapify***

Si se viola el orden del montículo porque la clave de un nodo se vuelve más pequeña que otra, es posible avanzar hacia la solución de la infracción intercambiando el nodo con el mayor de sus dos hijos. Este cambio puede causar una violación en el hijo; se puede arreglar esa violación de la misma manera, y así sucesivamente, bajando el montículo hasta que se llegue a un nodo con ambos hijos más pequeños, o a la parte inferior.

5.1.2 Montículos binarios basados en cola de prioridad

En relación al código mostrado, las operaciones son:

Insertar. Se agregamos el nuevo elemento al final de la matriz, se incrementa el tamaño del montículo y luego se "nada" a través del montículo con ese elemento para restaurar la condición de montículo.

Eliminar el máximo. Se elimina el elemento más grande de la parte superior, se coloca el elemento del extremo del montículo en la parte superior, se reducimos el tamaño del montículo y luego se "hunde" en el montículo para restaurar la condición del mismo.

Las operaciones ***sink*()** y ***swim*()** proporcionan la base para la implementación eficiente de la cola de prioridad, tal como se implementa en los códigos MaxPQ.*java* y MinPQ.*java* (Weiss,2013).

```
/******************************************************
 *  Compilation:  javac MaxPQ.java
 *  Execution:    java MaxPQ < input.txt
 *  Dependencies: StdIn.java StdOut.java
 *  Data files:   https://algs4.cs.princeton.edu/24pq/
   tinyPQ.txt
 *
 *  Generic max priority queue implementation with a
   binary heap.
 *  Can be used with a comparator instead of the natural
   order,
 *  but the generic Key type must still be Comparable.
 *
```

```
 *  % java MaxPQ < tinyPQ.txt
 *  Q X P (6 left on pq)
 *
 *  We use a one-based array to simplify parent and
    child calculations.
 *
 *  Can be optimized by replacing full exchanges with
    half exchanges
 *  (ala insertion sort).
 *
 ******************************************************/

1.  import java.util.Comparator;
2.  import java.util.Iterator;
3.  import java.util.NoSuchElementException;

4.  /**
5.  The {@code MaxPQ} class represents a priority queue
    of generic keys.
6.  It supports the usual <em>insert</em> and <em>delete-the-
maximum</em>
7.  operations, along with methods for peeking at the maximum key,
8.  testing if the priority queue is empty, and iterating through
9.  the keys.
10. <p>
11. This implementation uses a binary heap.
12. The <em>insert</em> and <em>delete-the-maximum</em>
operations take
13. logarithmic amortized time.
14. The <em>max</em>, <em>size</em>, and <em>is-empty</em>
operations take constant time.
15. Construction takes time proportional to the specified
    capacity or the number of
16. items used to initialize the data structure.
17. <p>
18. For additional documentation, see <a href="https://
    algs4.cs.princeton.edu/24pq">Section 2.4</a> of
19. <i>Algorithms, 4th Edition</i> by Robert Sedgewick
    and Kevin Wayne.
```

```
20. *
21. @author Robert Sedgewick
22. @author Kevin Wayne
23. *
24. @param <Key> the generic type of key on this priority
    queue
25. */

26. public class MaxPQ<Key> implements Iterable<Key> {
27. private Key[] pq;                    // store items
    at indices 1 to n
28. private int n;                       // number of
    items on priority queue
29. private Comparator<Key> comparator;  // optional
    comparator

30. /**
31. Initializes an empty priority queue with the given
    initial capacity.
32. *
33. @param  initCapacity the initial capacity of this
    priority queue
34. */
35. public MaxPQ(int initCapacity) {
36. pq = (Key[]) new Object[initCapacity + 1];
37. n = 0;
38. }

39. /**
40. Initializes an empty priority queue.
41. */
42. public MaxPQ() {
43. this(1);
44. }

45. /**
46. Initializes an empty priority queue with the given
    initial capacity,
47. using the given comparator.
```

```
48. *
49. @param  initCapacity the initial capacity of this
    priority queue
50. @param  comparator the order in which to compare the keys
51. */
52. public MaxPQ(int initCapacity, Comparator<Key>
    comparator) {
53. this.comparator = comparator;
54. pq = (Key[]) new Object[initCapacity + 1];
55. n = 0;
56. }

57. /**
58. Initializes an empty priority queue using the given
    comparator.
59. *
60. @param  comparator the order in which to compare the keys
61. */
62. public MaxPQ(Comparator<Key> comparator) {
63. this(1, comparator);
64. }

65. /**
66. Initializes a priority queue from the array of keys.
67. Takes time proportional to the number of keys, using
    sink-based heap construction.
68. *
69. @param  keys the array of keys
70. */
71. public MaxPQ(Key[] keys) {
72. n = keys.length;
73. pq = (Key[]) new Object[keys.length + 1];
74. for (int i = 0; i < n; i++)
      a.  pq[i+1] = keys[i];
75. for (int k = n/2; k >= 1; k--)
      a.  sink(k);
76. assert isMaxHeap();
77. }
```

```
78. /**
79. Returns true if this priority queue is empty.
80. *
81. @return {@code true} if this priority queue is empty;
82. {@code false} otherwise
83. */
84. public boolean isEmpty() {
85. return n == 0;
86. }

87. /**
88. Returns the number of keys on this priority queue.
89. *
90. @return the number of keys on this priority queue
91. */
92. public int size() {
93. return n;
94. }

95. /**
96. Returns a largest key on this priority queue.
97. *
98. @return a largest key on this priority queue
99. @throws NoSuchElementException if this priority queue
    is empty
100.    */
101.         public Key max() {
102.         if (isEmpty()) throw new NoSuchElementExcep
             ion("Priority queue underflow");
103.         return pq[1];
104.         }

105.         // helper function to double the size of
             the heap array
106.         private void resize(int capacity) {
107.         assert capacity > n;
108.         Key[] temp = (Key[]) new Object[capacity];
109.         for (int i = 1; i <= n; i++) {
     a.  temp[i] = pq[i];
```

```
110.        }
111.        pq = temp;
112.        }

113.        /**
114.        Adds a new key to this priority queue.
115.        *
116.        @param  x the new key to add to this
            priority queue
117.        */
118.        public void insert(Key x) {

119.        // double size of array if necessary
120.        if (n == pq.length - 1) resize(2 * pq.length);

121.        // add x, and percolate it up to maintain
            heap invariant
122.        pq[++n] = x;
123.        swim(n);
124.        assert isMaxHeap();
125.        }

126.        /**
127.        Removes and returns a largest key on this
            priority queue.
128.        *
129.        @return a largest key on this priority queue
130.        @throws NoSuchElementException if this
            priority queue is empty
131.        */
132.        public Key delMax() {
133.        if (isEmpty()) throw new
            NoSuchElementException("Priority queue
            underflow");
134.        Key max = pq[1];
135.        exch(1, n--);
136.        sink(1);
137.        pq[n+1] = null;     // to avoid loiterig
            and help with garbage collection
```

```
138.          if ((n > 0) && (n == (pq.length - 1) / 4))
              resize(pq.length / 2);
139.          assert isMaxHeap();
140.          return max;
141.          }


142.          /***************************************
143.          Helper functions to restore the heap invariant.
144.          ***************************************/

145.          private void swim(int k) {
146.          while (k > 1 && less(k/2, k)) {
      a.  exch(k, k/2);
      b.  k = k/2;
147.          }
148.          }

149.          private void sink(int k) {
150.          while (2*k <= n) {
      a.  int j = 2*k;
      b.  if (j < n && less(j, j+1)) j++;
      c.  if (!less(k, j)) break;
      d.  exch(k, j);
      e.  k = j;
151.          }
152.          }

153.          /***************************************
154.          Helper functions for compares and swaps.
155.          ***************************************/
156.          private boolean less(int i, int j) {
157.          if (comparator == null) {
      a.  return ((Comparable<Key>) pq[i]).compareTo(pq[j]) < 0;
158.          }
159.          else {
      a.  return comparator.compare(pq[i], pq[j]) < 0;
160.          }
161.          }
```

```
162.        private void exch(int i, int j) {
163.        Key swap = pq[i];
164.        pq[i] = pq[j];
165.        pq[j] = swap;
166.        }

167.        // is pq[1..N] a max heap?
168.        private boolean isMaxHeap() {
169.        return isMaxHeap(1);
170.        }

171.        // is subtree of pq[1..n] rooted at k a max heap?
172.        private boolean isMaxHeap(int k) {
173.        if (k > n) return true;
174.        int left = 2*k;
175.        int right = 2*k + 1;
176.        if (left  <= n && less(k, left))  return
false;
177.        if (right <= n && less(k, right)) return
false;
178.        return isMaxHeap(left) && isMaxHeap(right);
179.        }


180.        /**************************************
181.        Iterator.
182.        **************************************/

183.        /**
184.        Returns an iterator that iterates over the
            keys on this priority queue
185.        in descending order.
186.        The iterator doesn't implement {@code
            remove()} since it's optional.
187.        *
188.        @return an iterator that iterates over the
            keys in descending order
189.        */
190.        public Iterator<Key> iterator() {
191.        return new HeapIterator();
```

```
192.          }

193.          private class HeapIterator implements
              Iterator<Key> {

194.          // create a new pq
195.          private MaxPQ<Key> copy;

196.          // add all items to copy of heap
197.          // takes linear time since already in heap
              order so no keys move
198.          public HeapIterator() {
      a.  if (comparator == null) copy = new
              MaxPQ<Key>(size());
      b.  else          copy = new MaxPQ<Key>(size(),
          comparator);
      c.  for (int i = 1; i <= n; i++)
      d.  copy.insert(pq[i]);
199.          }

200.          public boolean hasNext()  { return !copy.
              isEmpty();                      }
201.          public void remove()      { throw new
         UnsupportedOperationException(); }

202.          public Key next() {
      a.  if(!hasNext())thrownewNoSuchElementException();
      b.  return copy.delMax();
203.          }
204.          }

205.          /**
206.          Unit tests the {@code MaxPQ} data type.
207.          *
208.          @param args the command-line arguments
209.          */
210.          public static void main(String[] args) {
211.          MaxPQ<String> pq = new MaxPQ<String>();
212.          while (!StdIn.isEmpty()) {
```

```
a.  String item = StdIn.readString();
b.  if (!item.equals("-")) pq.insert(item);
c.  else if (!pq.isEmpty()) StdOut.print(pq.delMax() + " ");
213.        }
214.        StdOut.println("(" + pq.size() + " left on pq)");
215.        }

216.        }

217.        /*************************************
218.        Compilation:  javac MinPQ.java
219.        *  Execution:    java MinPQ < input.txt
220.        Dependencies: StdIn.java StdOut.java
221.        *  Data files:   https://algs4.cs.princeton.
            edu/24pq/tinyPQ.txt
222.        *
223.        Generic min priority queue implementation
            with a binary heap.
224.        Can be used with a comparator instead of
            the natural order.
225.        *
226.        % java MinPQ < tinyPQ.txt
227.        E A E (6 left on pq)
228.        *
229.        We use a one-based array to simplify parent
            and child calculations.
230.        *
231.        Can be optimized by replacing full
            exchanges with half exchanges
232.        (ala insertion sort).
233.        *
234.        *************************************/

235.        import java.util.Comparator;
236.        import java.util.Iterator;
237.        import java.util.NoSuchElementException;

238.        /**
239.        The {@code MinPQ} class represents a
```

```
        priority queue of generic keys.
240.          It supports the usual <em>insert</em> and
              <em>delete-the-
        minimum</em>
241.          operations, along with methods for peeking
              at the minimum key,
242.          testing if the priority queue is empty, and
              iterating through
243.          the keys.
244     <p>
245.          This implementation uses a binary heap.
246.          The <em>insert</em> and <em>delete-the-
              minimum</em>
        operations take
247.          logarithmic amortized time.
248.          The <em>min</em>, <em>size</em>, and
              <em>is-empty
        </em> operations take constant time.
249.          Construction takes time proportional to the
              specified capacity or the
        number of
250.          items used to initialize the data
              structure.
251.          <p>
252.          For additional documentation, see <a
              href="https://algs4.
        cs.princeton. edu/24pq">Section 2.4</a> of
253.          <i>Algorithms, 4th Edition</i> by Robert
              Sedgewick and Kevin
        Wayne.
254.          *
255.          @author Robert Sedgewick
256.          @author Kevin Wayne
257.          *
258.          @param <Key> the generic type of key on
              this priority queue
259.          */
260.          public class MinPQ<Key> implements
              Iterable<Key> {
```

```
261.        private Key[] pq;                    //
            store items at indices 1 to n
262.        private int n;                       //
            number of items on priority queue
263.        private Comparator<Key> comparator;  //
            optional comparator

264.        /**
265.        Initializes an empty priority queue with
            the given initial capacity.
266.        *
267.        @param  initCapacity the initial capacity
            of this priority queue
268.        */
269.        public MinPQ(int initCapacity) {
270.        pq = (Key[]) new Object[initCapacity + 1];
271.        n = 0;
272.        }

273.        /**
274.        Initializes an empty priority queue.
275.        */
276.        public MinPQ() {
277.        this(1);
278.        }

279.        /**
280.        Initializes an empty priority queue with
            the given initial capacity,
281.        using the given comparator.
282.        *
283.        @param  initCapacity the initial capacity
            of this priority queue
284.        @param  comparator the order in which to
            compare the keys
285.        */
286.        public MinPQ(int initCapacity,
            Comparator<Key> comparator) {
287.        this.comparator = comparator;
```

```
288.          pq = (Key[]) new Object[initCapacity + 1];
289.          n = 0;
290.          }

291.          /**
292.          Initializes an empty priority queue using
              the given comparator.
293.          *
294.          @param  comparator the order in which to
              compare the keys
295.          */
296.          public MinPQ(Comparator<Key> comparator) {
297.          this(1, comparator);
298.          }

299.          /**
300.          Initializes a priority queue from the array
              of keys.
301.          <p>
302.          Takes time proportional to the number of
              keys, using sink-based heap construction.
303.          *
304.          @param  keys the array of keys
305.          */
306.          public MinPQ(Key[] keys) {
307.          n = keys.length;
308.          pq = (Key[]) new Object[keys.length + 1];
309.          for (int i = 0; i < n; i++)
     a.  pq[i+1] = keys[i];
310.          for (int k = n/2; k >= 1; k--)
     a.  sink(k);
311.          assert isMinHeap();
312.          }
313.          /**
314.          Returns true if this priority queue is
              empty.
315.          *
316.          @return {@code true} if this priority queue
              is empty;
```

```
317.        {@code false} otherwise
318.        */
319.        public boolean isEmpty() {
320.        return n == 0;
321.        }

322.        /**
323.        Returns the number of keys on this priority
            queue.
324.        *
325.        @return the number of keys on this priority
            queue
326.        */
327.        public int size() {
328.        return n;
329.        }

330.        /**
331.        Returns a smallest key on this priority
            queue.
332.        *
333.        @return a smallest key on this priority
            queue
334.        @throws NoSuchElementException if this
            priority queue is empty
335.        */
336.        public Key min() {
337.        if (isEmpty()) throw new
            NoSuchElementException("Priority queue
            underflow");
338.        return pq[1];
339.        }

340.        // helper function to double the size of
            the heap array
341.        private void resize(int capacity) {
342.        assert capacity > n;
343.        Key[] temp = (Key[]) new Object[capacity];
344.        for (int i = 1; i <= n; i++) {
```

```
    a.  temp[i] = pq[i];
345.        }
346.        pq = temp;
347.        }

348.        /**
349.        Adds a new key to this priority queue.
350.        *
351.        @param  x the key to add to this priority queue
352.        */
353.        public void insert(Key x) {
354.        // double size of array if necessary
355.        if (n == pq.length - 1) resize(2 *
            pq.length);

356.        // add x, and percolate it up to maintain
            heap invariant
357.        pq[++n] = x;
358.        swim(n);
359.        assert isMinHeap();
360.        }

361.        /**
362.        Removes and returns a smallest key on this
            priority queue.
363.        *
364.        @return a smallest key on this priority queue
365.        @throws NoSuchElementException if this
            priority queue is empty
366.        */
367.        public Key delMin() {
368.        if (isEmpty()) throw new
            NoSuchElementException("Priority queue
            underflow");
369.        Key min = pq[1];
370.        exch(1, n--);
371.        sink(1);
372.        pq[n+1] = null;     // to avoid loiterig
            and help with garbage
```

```
        collection
373.          if ((n > 0) && (n == (pq.length - 1) / 4))
              resize(pq.length / 2);
374.          assert isMinHeap();
375.          return min;
376.          }


377.          /***************************************
378.          Helper functions to restore the heap
              invariant.
379.          ***************************************/

380.          private void swim(int k) {
381.          while (k > 1 && greater(k/2, k)) {
     a.  exch(k, k/2);
     b.  k = k/2;
382.          }
383.          }

384.          private void sink(int k) {
385.          while (2*k <= n) {
     a.  int j = 2*k;
     b.  if (j < n && greater(j, j+1)) j++;
     c.  if (!greater(k, j)) break;
     d.  exch(k, j);
     e.  k = j;
386.          }
387.          }

388.          /***************************************
389.          Helper functions for compares and swaps.
390.          ***************************************/
391.          private boolean greater(int i, int j) {
392.          if (comparator == null) {
     a.  return ((Comparable<Key>) pq[i]).compareTo(pq[j]) > 0;
393.          }
394.          else {
     a.  return comparator.compare(pq[i], pq[j]) > 0;
```

```
395.          }
396.          }

397.          private void exch(int i, int j) {
398.          Key swap = pq[i];
399.          pq[i] = pq[j];
400.          pq[j] = swap;
401.          }

402.          // is pq[1..N] a min heap?
403.          private boolean isMinHeap() {
404.          return isMinHeap(1);
405.          }

406.          // is subtree of pq[1..n] rooted at k a min heap?
407.          private boolean isMinHeap(int k) {
408.          if (k > n) return true;
409.          int left = 2*k;
410.          int right = 2*k + 1;
411.     if (left  <= n && greater(k, left))  return false;
412.     if (right <= n && greater(k, right)) return false;
413.          return isMinHeap(left) && isMinHeap(right);
414.          }

415.          /**
416.          Returns an iterator that iterates over the
              keys on this priority queue
417.          in ascending order.
418.          <p>
419.          The iterator doesn't implement {@code
              remove()} since it's optional.
420.          *
421.          @return an iterator that iterates over the
              keys in ascending order
422.          */
423.          public Iterator<Key> iterator() {
424.          return new HeapIterator();
425.          }
```

```
426.          private class HeapIterator implements
              Iterator<Key> {
427.          // create a new pq
428.          private MinPQ<Key> copy;

429.          // add all items to copy of heap
430.          // takes linear time since already in heap
              order so no keys move
431.          public HeapIterator() {
     a. if(comparator==null)copy=newMinPQ<Key>(size());
     b. else                copy = new MinPQ<Key>(size(),
        comparator);
     c. for (int i = 1; i <= n; i++)
     d. copy.insert(pq[i]);
432.          }

433.     public boolean hasNext()  { return !copy.isEmpty();}
434.          public void remove()      { throw new
              UnsupportedOperationException();  }

435.          public Key next() {
     a. if(!hasNext())thrownewNoSuchElementException();
     b. return copy.delMin();
436.          }
437.          }

438.          /**
439.          Unit tests the {@code MinPQ} data type.
440.          *
441.          @param args the command-line arguments
442.          */
443.          public static void main(String[] args) {
444.          MinPQ<String> pq = new MinPQ<String>();
445.          while (!StdIn.isEmpty()) {
     a. String item = StdIn.readString();
     b. if (!item.equals("-")) pq.insert(item);
     c. else if (!pq.isEmpty()) StdOut.print(pq.delMin()
        + " ");
446.          }
```

```
447.            StdOut.println("(" + pq.size() + " left on pq)");
448.            }

449.            }
```

Estos dos programas demuestran la implementación de los montículos binarios en colas de prioridad.

Tema 6.
Grafos

Los grafos son una estructura de datos ampliamente utilizada en informática y diferentes aplicaciones informáticas. Un grafo G (V, A) es un objeto que consiste en un conjunto de nodos o vértices y un conjunto de aristas. La siguiente figura muestra un ejemplo de un grafo (Weiss,2013).

Los puntos etiquetados son los nodos o vértices del gráfico, y las líneas entre ellos representan las aristas. Entonces, por ejemplo, los puntos 1 y 2 son vértices, mientras que la línea {1, 2} es una arista. Una arista representa una relación entre dos vértices. Si dos vértices tienen una arista entre ellos, se considera que estos vértices son **adyacentes** y se dice que la arista es incidente a los vértices que toca.

La teoría de grafos proporciona una forma simple de modelar una serie de problemas, como el almacenamiento de sustancias químicas peligrosas, la búsqueda de rutas, el transporte de mercancías y la extensión de árboles. La estructura de datos grafos funciona particularmente bien en la aplicación de algoritmos, como con los algoritmos de búsqueda de ruta y árbol de expansión.

Todos los grafos se dividen en dos grandes grupos: gráficos dirigidos y no dirigidos. La diferencia es que las aristas en los grafos dirigidos, llamados arcos, tienen una dirección. Una arista se puede dibujar como una línea. Si se dirige un gráfico, cada línea tiene una flecha.

La secuencia de vértices, de modo que haya una arista de cada vértice al siguiente en secuencia, se denomina ruta. Al primer vértice en el camino se llama el vértice de inicio; el último vértice en el camino se llama el vértice final. Si los vértices de inicio y fin son iguales, la ruta se llama ciclo. La ruta se llama simple si incluye a todos los vértices solo una vez. El ciclo se llama simple si incluye a todos los vértices solo una vez, excepto iniciar (finalizar). A continuación se observan ejemplos de ruta y ciclo.

Una característica de los grafos es la posibilidad de ponderación de las aristas. Un grafo se denomina ponderado, si cada arista está asociada con un número real, llamado peso de la arista. Por ejemplo, en una red vial, el peso de cada carretera puede ser su longitud o el tiempo mínimo necesario para conducir. La siguiente figura muestra un ejemplo de un grafo ponderado.

6.1 Representación de grafos no dirigidos

Existen varias formas de representar grafos no dirigidos. Las más frecuentes son la matriz de **adyacencia** y la lista de adyacencia.

6.1.1 Matriz de adyacencia

Cada celda *aij* de una matriz de adyacencia contiene 0, si hay una arista entre los vértices i-ésimo y j-ésimo, y 1 en caso contrario. Un grafo no dirigido tiene una **matriz de adyacencia** es simétrica. De hecho, en el gráfico no dirigido, si hay una arista (2, 5), también hay una arista (5, 2). Esta es también la razón por la que hay dos celdas por cada arista en la muestra. Los bucles, si están permitidos en un grafo, corresponden a los elementos diagonales de una matriz de adyacencia. Una ventaja de la matriz de adyacencia es que es muy simple de programar. (Weiss,2013)

6.1.2 Lista de adyacencia

Este tipo de representación gráfica es una de las alternativas a la matriz de adyacencia. Requiere menos cantidad de memoria y, en situaciones particulares, incluso puede superar la matriz de adyacencia. Para cada lista de adyacencia de vértices se almacena una lista de vértices, que son adyacentes a la actual. Esto se puede observar en la siguiente figura.

La lista de adyacencia nos permite almacenar el grafo en una forma más compacta, que la matriz de adyacencia, pero la diferencia disminuye a medida que el grafo se vuelve más denso. La ventaja es que la lista de adyacencia permite obtener la lista de vértices adyacentes en O(1) tiempo, lo que es una gran ventaja para algunos algoritmos.

El siguiente código muestra un ejemplo de implementación de un grafo. Iniciando con la clase vértice, en este caso llamada *Vertex*. Un vértice es un nodo en el gráfico, como se describió anteriormente. Los vecinos del vértice se describen con un arreglo *ArrayList <Edge>*. El propósito de usar la incidencia de

vecinos, es decir, las aristas (*Edges*) en lugar de la adyacencia de vecinos, es decir, vértices (*Vertex*) fue porque los algoritmos de búsqueda de ruta y árbol de expansión necesitan que las aristas funcionen. Se tiene que considerar que esta implementación no permite múltiples objetos de aristas (*Edge*) entre el mismo par de objetos vértice (*Vertex*) (Weiss,2013).

```
1. import java.util.ArrayList;
2. /**
3. This class models a vertex in a graph. For ease of
4. the reader, a label for this vertex is required.
5. Note that the Graph object only accepts one Vertex
   per label,
6. so uniqueness of labels is important. This vertex's
   neighborhood
7. is described by the Edges incident to it.
8. *
9. @author Michael Levet
10. @date June 09, 2015
11. */
12. public class Vertex {

13. private ArrayList<Edge> neighborhood;
14. private String label;

15. /**
16. *
17. @param label The unique label associated with this
   Vertex
18. */
19. public Vertex(String label){
20. this.label = label;
21. this.neighborhood = new ArrayList<Edge>();
22. }


23. /**
24. This method adds an Edge to the incidence
   neighborhood of this graph iff
25. the edge is not already present.
26. *
27. @param edge The edge to add
28. */
```

```
29. public void addNeighbor(Edge edge){
30. if(this.neighborhood.contains(edge)){
a.  return;
31. }
32. this.neighborhood.add(edge);
33. }

34. /**
35. *
36. @param other The edge for which to search
37. @return true iff other is contained in this.
    eighborhood
38. */
39. public boolean containsNeighbor(Edge other){
40. return this.neighborhood.contains(other);
41. }

42. /**
43. *
44. @param index The index of the Edge to retrieve
45. @return Edge The Edge at the specified index in this.
    eighborhood
46. */
47. public Edge getNeighbor(int index){
48. return this.neighborhood.get(index);
49. }

50. /**
51. *
52. @param index The index of the edge to remove from
    this.neighborhood
53. @return Edge The removed Edge
54. */
55. Edge removeNeighbor(int index){
56. return this.neighborhood.remove(index);
57. }

58. /**
59. *
```

```
60. @param e The Edge to remove from this.neighborhood
61. */
62. public void removeNeighbor(Edge e){
63. this.neighborhood.remove(e);
64. }

65. /**
66. *
67. @return int The number of neighbors of this Vertex
68. */
69. public int getNeighborCount(){
70. return this.neighborhood.size();
71. }

72. /**
73. *
74. @return String The label of this Vertex
75. */
76. public String getLabel(){
77. return this.label;
78. }

79. /**
80. *
81. @return String A String representation of this Vertex
82. */
83. public String toString(){
84. return "Vertex " + label;
85. }

86. /**
87. *
88. @return The hash code of this Vertex's label
89. */
90. public int hashCode(){
91. return this.label.hashCode();
92. }

93. /**
```

```
94. *
95. @param other The object to compare
96. @return true iff other instanceof Vertex and the two
    Vertex objects have
the same label
97. */
98. public boolean equals(Object other){
99. if(!(other instanceof Vertex)){
     a.  return false;
100.    }

101.    Vertex v = (Vertex)other;
102.    return this.label.equals(v.label);
103.    }

104.    /**
105.    *
106.    @return ArrayList<Edge> A copy of this.
        eighborhood. Modifying the returned
107.    ArrayList will not affect the neighborhood of
        this Vertex
108.    */
109.    public ArrayList<Edge> getNeighbors(){
110.    return new ArrayList<Edge>(this.neighborhood);
111.    }


112.    }
```

A continuación, se puede examinar la clase arista (Edge). Una arista modela la relación de adyacencia entre dos vértices. Entonces la clase arista (Edge) tiene dos vértices. En muchas situaciones, la noción de un peso de la arista también es importante. Intuitivamente, el peso de la arista representa la distancia entre dos vértices. Incluso en los grafos no ponderados, es una noción comúnmente aceptada que atravesar dos aristas tiene un costo mayor que permanecer en el vértice actual. Entonces la clase arista (Edge) permite un atributo de peso, asumiendo por convención un peso uniforme de 1 si no se especifica peso. Esto permite tratar el grafo como "no ponderado" de una manera que sigue siendo coherente con las condiciones previas de muchos algoritmos de grafos comunes (Weiss,2013).

Es importante observar si el método *compareTo* () devuelve un valor de 0, entonces el método *equals* () devolverá un valor de verdadero para el parámetro dado. El método *compareTo* () se utiliza para comparar estrictamente los pesos de la arista, mientras que el método *equals* () simplemente prueba

si el borde hace referencia al mismo par de vértices. La decisión de diseño tiene dos consideraciones. El primero es prohibir múltiples aristas para el mismo par de vértices, a lo que contribuye el método *equals* (). La segunda consideración es que al método *compareTo* () solo le interesan las ponderaciones de las aristas (Edge) en consideraciones algorítmicas.

Por último, se considera la implementación de la clase *Graph*. Se utiliza una implementación de lista de incidencia, que es favorable para muchos (pero no todos) algoritmos de grafos. Las operaciones básicas de inserción, búsqueda y eliminación son compatibles con los objetos *Vertex* y *Edge*. Los objetos vértice se identifican de forma única por sus etiquetas, y no hay dos objetos *Vertex* que puedan compartir la misma etiqueta. Sin embargo, existe soporte para sobrescribir objetos *Vertex* existentes. Hay que considerar que si se hace, se eliminarán del grafo todas las aristas asociadas con el vértice existente.

En este caso no se proporciona el método *equals* () para la clase *Graph*. La noción de igualdad de grafos se conoce como isomorfismo del grafo .import java.util.*; (Weiss,2013).

```
1.  /**
2.  This class models a simple, undirected graph using an
3.  incidence list representation. Vertices are identified
4.  uniquely by their labels, and only unique vertices
    are allowed.
5.  At most one Edge per vertex pair is allowed in this
    Graph.
6.  *
7.  Note that the Graph is designed to manage the Edges. You
8.  should not attempt to manually add Edges yourself.
9.  *
10. @author Michael Levet
11. @date June 09, 2015
12. */
13. public class Graph {

14. private HashMap<String, Vertex> vertices;
15. private HashMap<Integer, Edge> edges;

16. public Graph(){
17. this.vertices = new HashMap<String, Vertex>();
18. this.edges = new HashMap<Integer, Edge>();
19. }

20. /**
21. This constructor accepts an ArrayList<Vertex> and
```

```
   populates
22. this.vertices. If multiple Vertex objects have the
   same label,
23. then the last Vertex with the given label is used.
24. *
25. @param vertices The initial Vertices to populate this
   Graph
26. */
27. public Graph(ArrayList<Vertex> vertices){
28. this.vertices = new HashMap<String, Vertex>();
29. this.edges = new HashMap<Integer, Edge>();

30. for(Vertex v: vertices){
     a.  this.vertices.put(v.getLabel(), v);
31. }

32. }

33. /**
34. This method adds am edge between Vertices one and two
35. of weight 1, if no Edge between these Vertices
   already
36. exists in the Graph.
37. *
38. @param one The first vertex to add
39. @param two The second vertex to add
40. @return true iff no Edge relating one and two exists
   in the Graph
41. */
42. public boolean addEdge(Vertex one, Vertex two){
43. return addEdge(one, two, 1);
44. }

45. /**
46. Accepts two vertices and a weight, and adds the edge
47. ({one, two}, weight) iff no Edge relating one and two
48. exists in the Graph.
49. *
50. @param one The first Vertex of the Edge
```

```
51. @param two The second Vertex of the Edge
52. @param weight The weight of the Edge
53. @return true iff no Edge already exists in the Graph
54. */
55. public boolean addEdge(Vertex one, Vertex two, int
    weight){
56. if(one.equals(two)){
57. return false;
58. }

59. //ensures the Edge is not in the Graph
60. Edge e = new Edge(one, two, weight);
61. if(edges.containsKey(e.hashCode())){
62. return false;
63. }

64. //and that the Edge isn't already incident to one of
    the vertices
65. else if(one.containsNeighbor(e) ||
    two.containsNeighbor(e)){
66. return false;
67. }

68. edges.put(e.hashCode(), e);
69. one.addNeighbor(e);
70. two.addNeighbor(e);
71. return true;
72. }

73. /**
74. *
75. @param e The Edge to look up
76. @return true iff this Graph contains the Edge e
77. */
78. public boolean containsEdge(Edge e){
79. if(e.getOne() == null || e.getTwo() == null){
80. return false;
81. }
```

```
82. return this.edges.containsKey(e.hashCode());
83. }


84. /**
85. This method removes the specified Edge from the Graph,
86. including as each vertex's incidence neighborhood.
87. *
88. @param e The Edge to remove from the Graph
89. @return Edge The Edge removed from the Graph
90. */
91. public Edge removeEdge(Edge e){
92. e.getOne().removeNeighbor(e);
93. e.getTwo().removeNeighbor(e);
94. return this.edges.remove(e.hashCode());
95. }
96. /**
97. *
98. @param vertex The Vertex to look up
99. @return true iff this Graph contains vertex
100.          */
101.          public boolean containsVertex(Vertex
              vertex){
102.          return this.vertices.get(vertex.getLabel())
              != null;
103.          }

104.          /**
105.          *
106.          @param label The specified Vertex label
107.          @return Vertex The Vertex with the specified label
108.          */
109.          public Vertex getVertex(String label){
110.          return vertices.get(label);
111.          }

112.          /**
113.          This method adds a Vertex to the graph. If
              a Vertex with the same label
```

```
114.          as the parameter exists in the Graph, the
              existing Vertex is overwritten
115.          only if overwriteExisting is true. If the
              existing Vertex is overwritten,
116.          the Edges incident to it are all removed
              from the Graph.
117.          *
118.          @param vertex
119.          @param overwriteExisting
120.          @return true iff vertex was added to the Graph
121.          */
122       public boolean addVertex(Vertex vertex, boolean
          overwriteExisting){
123.            Vertex current = this.vertices.get
                (vertex.getLabel());
124.            if(current != null){
     a.  if(!overwriteExisting){
     b.  return false;
     c.  }

     d.  while(current.getNeighborCount() > 0){
     e.  this.removeEdge(current.getNeighbor(0));
     f.  }
125.            }


126.          vertices.put(vertex.getLabel(), vertex);
127.          return true;
128.          }

129.          /**
130.          *
131.          @param label The label of the Vertex to remove
132.          @return Vertex The removed Vertex object
133.          */
134.          public Vertex removeVertex(String label){
135.          Vertex v = vertices.remove(label);

136.          while(v.getNeighborCount() > 0){
```

```
    a.  this.removeEdge(v.getNeighbor((0)));
137.        }

138.        return v;
139.        }

140.        /**
141.        *
142.        @return Set<String> The unique labels of
            the Graph's Vertex objects
143.        */
144.        public Set<String> vertexKeys(){
145.        return this.vertices.keySet();
146.        }

147.        /**
148.        *
149.        @return Set<Edge> The Edges of this graph
150.        */
151.        public Set<Edge> getEdges(){
152.        return new HashSet<Edge>
            (this.edges.values());
153.        }


154.        }
```

En el siguiente código se puede observar la funcionalidad de la clase *Graph:*

```
1.  /**
2.  *
3.  @author Michael Levet
4.  @date June 09, 2015
5.  */
6.  public class DemoGraph {

7.  public static void main(String[] args){
8.  Graph graph = new Graph();


9.  //initialize some vertices and add them to the graph
```

```
10. Vertex[] vertices = new Vertex[5];
11. for(int i = 0; i < vertices.length; i++){
      a. vertices[i] = new Vertex("" + i);
      b. graph.addVertex(vertices[i], true);
12. }

13. //illustrate the fact that duplicate edges aren't
    added
14. for(int i = 0; i < vertices.length - 1; i++){
      a. for(int j = i + 1; j < vertices.length; j++){
      b. graph.addEdge(vertices[i], vertices[j]);
      c. graph.addEdge(vertices[i], vertices[j]);
      d. graph.addEdge(vertices[j], vertices[i]);
      e. }
15. }

16. //display the initial setup- all vertices adjacent
    to each other
17. for(int i = 0; i < vertices.length; i++){
      a. System.out.println(vertices[i]);

      b. for(int j = 0; j < vertices[i].getNeighborCount(); j++){
      c. System.out.println(vertices[i].getNeighbor(j));
      d. }

      e. System.out.println();
18. }

19. //overwritte Vertex 3
20. graph.addVertex(new Vertex("3"), true);
21. for(int i = 0; i < vertices.length; i++){
      a. System.out.println(vertices[i]);

      b. for(int j = 0; j < vertices[i].getNeighborCount(); j++){
      c. System.out.println(vertices[i].getNeighbor(j));
      d. }

      e. System.out.println();
22. }
```

```
23. System.out.println("Vertex 5: " +
    graph.getVertex("5")); //null
24. System.out.println("Vertex 3: " +
    graph.getVertex("3")); //Vertex 3

25. //true
26. System.out.println("Graph Contains {1, 2}: " +
      a.  graph.containsEdge(new Edge(graph.getVertex("1"), graph.
          getVertex("2"))));

27. //true
28. System.out.println(graph.removeEdge(new Edge
    (graph.getVertex("1"),
graph.getVertex("2"))));

29. //false
30. System.out.println("Graph Contains {1, 2}: " + graph.
    containsEdge(newEdge(graph.getVertex("1"),
    graph.getVertex("2"))));

31. //false
32. System.out.println("Graph Contains {2, 3} " + graph.
    containsEdge(newEdge(graph.getVertex("2"),
    graph.getVertex("3"))));

33. System.out.println(graph.containsVertex
    (new Vertex("1"))); //true
34. System.out.println(graph.containsVertex
    (new Vertex("6"))); //false
35. System.out.println(graph.removeVertex("2")); //Vertex 2
36. System.out.println(graph.vertexKeys()); //[3, 1, 0, 4]

37. }
38. }
```

Tema 7.
Conjuntos disjuntos

Esta estructura de datos tiene varias aplicaciones, como en el algoritmo de Kruskal o cuando se trata de organizar/programar tareas. Los Conjuntos disjuntos permiten determinar rápidamente qué elementos están conectados y a fusionar dos componentes en una sola entidad. Por lo tanto, se trata de una estructura de datos para realizar un seguimiento de un conjunto de elementos particionados en una serie de subconjuntos no superpuestos (disjuntos).

Existen 2 operaciones principales (Weiss,2013):

- Union (unir): ayuda a verificar si un gráfico es cíclico o no. También ayuda a conectar (fusionar) dos subconjuntos.
- Find (encontrar): ayuda a determinar a qué subconjunto pertenece un elemento en particular

La importancia de decidir si dos nodos están conectados en un gráfico, se basa en la posibilidad de crear, por ejemplo, un árbol de expansión con el algoritmo de Kruskal. Cada vértice en el gráfico original pertenece a un conjunto disjunto distinto al principio. Debido a que se representa el conjunto disjunto con una estructura de árbol, al principio se tienen tantos nodos distintos como el número de vértices en el gráfico. Se continúan conectando los vértices (así es como funciona el algoritmo

de Kruskal), y se asegura que no hayan ciclos en el árbol de expansión final. De ahí la necesidad de emplear la estructura de datos Conjuntos disjuntos, ya que es posible verificar los ciclos en el orden de O(1) tiempo de ejecución.

7.1 Creación de los conjuntos disjuntos

Primero se tiene que crear un conjunto distinto para todos los ítems. Por ejemplo, para todos los nodos en un gráfico (usualmente se usa esta estructura de datos cuando se trabaja con el algoritmo de Kruskal).

Por lo tanto, al inicio se tienen nodos independientes. Estos nodos representan los conjuntos disjuntos (tantos conjuntos disjuntos distintos como el número de vértices que se tengan en el gráfico). La implementación de esta operación de simple: se establece que el padre del nodo sea él mismo. Esto se puede observar en la siguiente declaración:

```
public void makeSet(Node node) {
     node.parent = node;
}
```

7.2 La operación find (encontrar)

A continuación, se usa una estructura de árbol para representar el conjunto disjunto. Varios elementos pueden pertenecer al mismo conjunto (teniendo en cuenta que al comienzo hay tantos conjuntos distintos como número de vértices en el gráfico). Por lo general, se representa el conjunto dado con uno de sus elementos. A este se llama el representante del conjunto. Cuando se busca un ítem, la operación devolverá a este representante.

La compresión de ruta es uno de los métodos de optimización. El nodo que se está buscando será un hijo directo de la raíz (representante). Esto altera el árbol de manera que las llamadas futuras llegarán a la raíz más rápidamente (en el orden O(1) de complejidad). Entonces, esta es la razón por la cual se necesita la compresión de ruta. Se debe de asegurar que todos los nodos sean hijos del representante (nodo raíz). Entonces se podrá acceder al nodo raíz en el orden O(1) de complejidad de tiempo.

Esto se puede observar en el siguiente código (Weiss,2013):

```
1.  public int find(Node n) {
2.  Node current = n;
3.  //find the root node (representative)
4.  while (current.getParent() != null)
5.  current = current.getParent();

6.  Node root = current;

7.  //path compression: set the node to be the child of
    the root
```

```
8.  current = n;
9.  while (current != root) {
10. Node temp = current.getParent();
11. current.setParent( root );
12. current = temp;
13. }

14. //representative's id
15. return root.getId();
16. }
```

7.3 La operación union (unión)

Con la operación de unión, se pueden fusionar dos conjuntos disjuntos conectándolos de acuerdo con los representantes. Debido a esto, tenemos que definir el parámetro de rango. El rango es muy similar al parámetro de altura de los árboles de búsqueda binarios. Por lo tanto, los nodos de hoja tienen rango 0. El nodo en la capa superior tiene rango 1 y así sucesivamente (Weiss,2013).

Para asegurarse de que el árbol no esté desequilibrado, es útil realizar la compresión de ruta. En cuanto a la fusión, se utiliza la unión por rango. Así que siempre se adjunta el árbol más pequeño a la raíz del más grande. Por lo tanto, el árbol estará más equilibrado y, debido a esto, será más rápido.

Esto se puede observar en el siguiente código (Weiss,2013):

```
1.  public void union(Node node1, Node node2) {

2.  int index1 = find(node1);
3.  int index2 = find(node2);

4.  //if they already part of the same set we return
5.  if (index1 == index2)
i.  return;

6.  //get the root nodes of each set (this will run in
    constant time)
7.  Node root1 = this.rootNodes.get(index1);
8.  Node root2 = this.rootNodes.get(index2);

9.  //attach the smaller tree to the root of the larger
    tree "union by height"
10. if (root1.getHeight() < root2.getHeight()) {
a.  root1.setParent(root2);
```

```
11. } else if (root1.getHeight() > root2.getHeight()) {
i.  root2.setParent(root1);
12. } else {
i.  root2.setParent(root1);
ii. root1.setHeight(root1.getHeight()+1);
13. }

14. this.setCount--;
15. }
```

7.4 Bosque de conjuntos disjuntos

La finalidad del bosque es mantener una colección de árboles, donde cada elemento apunta a su padre. La raíz de cada árbol es el representante del conjunto

Se usan dos estrategias para mejorar el tiempo de ejecución:

- Unión por rango
- Compresión de camino

Una de las muchas aplicaciones de estructuras de datos disjuntos surge al determinar los componentes de un gráfico no dirigido. Otra aplicación de la estructura de datos de conjuntos disjuntos es que se usa en el algoritmo de Kruskal para verifica si hay aristas seguras y eliminar los caminos que no son seguros, para que de este modo, solo las aristas seguras se agreguen al grafo para generar un árbol de expansión mínimo.

Glosario

Argumentos

También llamados parámetros, son una forma de intercambiar información con el método. Pueden servir para introducir datos para ejecutar el método (entrada) o para obtener o modificar datos tras su ejecución (salida).

Asintótico

Comportamiento de un algoritmo en función de los argumentos que se le pasen y la escala de los mismos.

Block de notas

Editor de texto incluido en los sistemas operativos de Microsoft desde 1985.

Bloques anidados

Inclusión de bloques de control dentro de otros, usualmente indicado mediante la inclusión de distintos niveles de sangría (llamada *indentation* en inglés) dentro del código fuente.

Ciclos

Es una sentencia que ejecuta repetidas veces un trozo de código, hasta que la condición asignada a dicho bucle deja de cumplirse.

Compilador

Es un programa informático que traduce un programa que ha sido escrito en un lenguaje de programación a un lenguaje común.

Concatenar

Se refiere a unir bajo una sola cadena de texto información proveniente de otras cadenas de texto.

Congruentes

Término usado en la teoría de números, para designar que dos números enteros a,b tienen el mismo resto al dividirlos por un número natural m distinto de cero, llamado módulo.

Consola

Es un método que permite a los usuarios dar instrucciones a algún programa informático por medio de una línea de texto.

Constantes

Es un valor que no puede ser alterado/modificado durante la ejecución de un programa, únicamente puede ser leído.

Criptografía

Arte y técnica de escribir con procedimientos o claves secretas o de un modo enigmático, de tal forma que lo escrito solamente sea inteligible para quien sepa descifrarlo.

Declaración

Acción de indicar el tipo de dato que podrá almacenar una variable o constante y su nombre (identificador)

Directiva

Instrucción del motor de búsqueda que utiliza un formato determinado para solicitar una función y pasa parámetros a la función en un bloque de parámetros.

Ejecución

Es el proceso mediante el cual una computadora lleva a cabo las instrucciones de un programa informático.

Entidades

Es la representación de un objeto.

Función

Es un conjunto de líneas de código que realizan una tarea específica y puede retornar un valor. Las funciones pueden tomar parámetros que modifiquen su funcionamiento.

Gestionar

Se refiere a manejar, controlar y administrar.

Indexación

Inclusión de un número a una posición particular de un arreglo.

Índice

El índice es una estructura que mejora la velocidad de las operaciones, por medio de identificador único, permitiendo un rápido acceso a los registros.

Inducción

Forma de razonamiento que consiste en establecer una ley o conclusión general a partir de la observación de hechos o casos particulares.

Inicializar

Es el darle un valor después que se ha declarado pero antes de que se ejecuten las sentencias en las que se emplea.

Instancias

Todo objeto que derive de algún otro. De esta forma, todos los objetos son instancias de algún otro, menos la clase *Object* que es la madre de todas.

Interfaces

Es un medio común para que los objetos no relacionados se comuniquen entre sí.

Máquina virtual

Software que simula un sistema de computación y puede ejecutar programas como si fuese una computadora rea.

Miembros

Son conjuntos de funciones o datos contenidos en una clase.

Null

Es un valor especial aplicado a un puntero usado para indicar que no se apunta a un objeto o dato válidos.

Operadores

Son símbolos que indican cómo se deben manipular los operandos. Los operadores junto con los operandos forman una expresión, que es una fórmula que define el cálculo de un valor.

Paquete

Es un contenedor de clases que permite agrupar las distintas partes de un programa y que por lo general tiene una funcionalidad y elementos comunes, definiendo la ubicación de dichas clases en un directorio de estructura jerárquica.

Paradigma

Define un conjunto de reglas, patrones y estilos de programación que son usados por un grupo de lenguajes de programación.

Ranura

Cada posición de la tabla *hash*.

Referencia

Una referencia en cualquier lenguaje de programación hace alusión a la posición en memoria RAM que tiene dicha variable u objeto

Sentencia

Son las unidades ejecutables más pequeñas de un programa, en otras palabras una línea de código escrita es una sentencia.

Ternario

Es un operador que toma tres argumentos.

Variable de entorno

Describe el entorno operativo de un proceso, como el directorio de inicio o el terminal en uso.

Variables

Son espacios reservados en la memoria que, como su nombre indica, pueden cambiar de contenido a lo largo de la ejecución de un programa.

Enlaces

Data Structure and Algorithms Tutorial

Este tutorial le proporcionará una gran comprensión de las estructuras de datos necesarias para comprender la complejidad de las aplicaciones de nivel empresarial y la necesidad de algoritmos y estructuras de datos.

https://www.tutorialspoint.com/data_structures_algorithms/

Introduction to Algorithms

Este curso proporciona una introducción al modelado matemático de problemas computacionales. Cubre los algoritmos comunes, paradigmas algorítmicos y estructuras de datos utilizados para resolver estos problemas. El curso enfatiza la relación entre algoritmos y programación, e introduce medidas básicas de rendimiento y técnicas de análisis para estos problemas.

https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-006-introduction-to-algorithms-fall-2011/index.htm

Java Tutorial Network: Java Tutorials for Beginners and Professionals

Java Tutorial Network (JavaTutorial.net) se dedica a proporcionarle tutoriales gratuitos de alta calidad de *Java* con una gran cantidad de ejemplos de *Java* que puede utilizar para aprender o completar sus tareas como desarrollador de *Java*.

https://*java*tutorial.net/

The Java™ Tutorials

Los Tutoriales de *Java* son guías prácticas para programadores que desean usar el lenguaje de programación *Java* para crear aplicaciones. Incluyen cientos de ejemplos completos de trabajo y docenas de lecciones.

https://docs.oracle.com/*java*se/tutorial/

Bibliografía

Referencias bibliográficas

Weiss, M. A., (2013). *Estructuras de datos en Java.* Madrid, España. PEARSON Educación, S.A.

Bibliografía recomendada

Guardati, S. (2015). *Estructura de datos básicos - programación orientada a objetos con java.* D.F, México. Alfaomega Grupo Editor.

Sznajdleder, P. (2017). *Programación orientada a objetos y estructura de datos a fondo implementación de algoritmos en Java.* Buenos Aires, Argentina. Alfaomega Grupo Editor.

Agradecimientos

Autor
Dr. Oscar Antonio Palma Gamboa
Colaboración
Dra. Martha Rocío Ceballos Hernández