2015 Septiembre

Cesar Pardo C.
Libardo Pantoja Y.
Ricardo Zambrano
Viviana Molano

Departamento de Sistemas

Universidad del Cauca

LIBRO

ESTRUCTURAS DE DATOS DINAMICAS

Una manera fácil de aprender

Editorial Unicauca

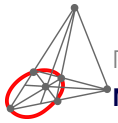
Departamento de Sistemas, Universidad del Cauca

J. Viviana Mora., W Libardo Pantoja Y.

Estructuras de Datos

Estructuras de Datos Dinámicas

Una forma fácil de aprender



Revista digital
Matemática, Educación e Internet. (http://www.tec-digital.itcr.ac.cr/revistamatematica/).

Correo Electrónico: editorial@unicauca.edu.co
Departamento de Sistemas
Facultad de Ingeniería Electrónica y Telecomunicaciones
Apdo. 159-7050, Popayán
Teléfono (057)8209800
Fax (058)8209900

Mora, Viviana.
Estructuras de Datos Dinámicas
W. Libardo Pantoja Yépez
– Departamento de Sistemas, FIET, Universidad del Cauca. 2015.
xxx p.
ISBN 978-9977-66-227-5
1. Estructuras. 2. de Datos 3. Dinámicas.

Licencia.

Revista digital
Matemática, Educación e Internet.
http://www.tec-digital.itcr.ac.cr/revistamatematica/.

Índice general

Prólogo

Este texto cubre aspectos básicos e intermedios sobre ipsúm dolor sit amet, consectetur adipiscing elit. Nam dignissim varius tempus. Cras eu malesuada ipsum. Pellentesque ut lorem velit. Mauris vehicula est orci, bibendum tincidunt enim mattis a. Interdum et malesuada fames ac ante ipsum primis in faucibus..

...

Popayan, 2015. V. MORA, W. PANTOJA.

1 Capítulo 1: Contenido y Estilos en Latex

Este capítulo sirve como modelo, es decir, para mostrar cómo utilizar latex. También muestra el posible contenido del libro. Un ejemplo de referencia bibliográfica es: [1].

Advertencia.

Las siguientes plantillas usan la versión 2014 del paquete `tcolorbox` (entre otros paquetes recientes), por lo tanto *debe actualizar los paquetes de sus distribución* TEX o instalar manualmente este paquete (ver el capítulo 9 del libro, `http://www.tec-digital.itcr.ac.cr/revistamatematica/Libros/LATEX/LaTeX_2014.pdf`). El paquete "psboxit" viene incluido en la carpeta.

1.1 Estructura General del Libro

1. Análisis de Algoritmos
   - *a*) Los algoritmos
   - *b*) El análisis de algoritmos
   - *c*) Función de complejidad
   - *d*) Cómo calcular la funcion de complejidad de un algoritmo
   - *e*) Orden de Magnitud (Notación O Grande)
   - *f*) Complejidad de un Algoritmo Recursivo
   - *g*) Ejercicios Propuestos
2. Introduccion a las estructuras de datos
   - *a*) Conceptos básicos sobre estructuras de datos.
   - *b*) Clasificación.
     1) Estructuras de Datos Estáticas.
     2) Estructuras de Datos Dinámicas.
3. Tipos abstractos de datos - TAD
   - *a*) Tipos de datos

b) Tipos abstractos de datos

c) Métodos para la Especificación de un TAD

d) Tipos de operaciones

e) Ejemplos de TADs

f) Ejercicios Propuestos

4. LISTAS DINÁMICAS

a) Definición

b) Usos de las listas

c) El TAD Lista

d) Implementación del TAD Lista orientado a objetos

e) Implementación del TAD lista mediante arreglos

f) Casos de estudio

g) Ejercicios propuestos

5. PILAS

a) Definición

b) Usos de las pilas

c) El TAD PIla

d) Implementación del TAD pila orientado a objetos

e) Implementación del TAD pila mediante arreglos

f) Casos de estudio

1) Correspondencia de delimitadores
2) Evaluación de expresiones aritméticas
3) Convertir una expresión dada en notación infija a una notación postfija
4) Evaluación de la Expresión en notación postfija

g) Ejercicios propuestos

6. COLAS

a) Definición

b) Usos de las colas

c) El TAD Cola

d) Implementación del TAD cola orientado a objetos

e) Implementación del TAD cola mediante arreglos

f) Casos de estudio

1) Correspondencia de delimitadores
2) Evaluación de expresiones aritméticas
3) Convertir una expresión dada en notación infija a una notación postfija
4) Evaluación de la Expresión en notación postfija

g) Ejercicios propuestos

7. ESTRUCTURAS DE DATOS NO LINEALES. ARBOLES BINARIOS
    - *a*) Introducción
    - *b*) Definición de árbol
    - *c*) Definición de árbol binario
    - *d*) Árbol de expresiones
    - *e*) Balance o equilibrio de un árbol binario
    - *f*) Árbol binario completo
    - *g*) TAD Árbol binario
    - *h*) Implementación del TAD de un árbol binario
    - *i*) Recorridos de un árbol
        1) Recorrido inorden
        2) Recorrido en preorden
        3) Recorrido en postorden
        4) Recorrido en anchura
    - *j*) Árbol binario de búsqueda
        1) Operación de inserción
        2) Operación de búsqueda
        3) Operación de eliminación
    - *k*) Árbol binario de búsqueda equilibrados AVL
        1) Eficiencia en la búsqueda de un árbol equilibrado
        2) Inserción en árboles AVL
        3) Borrado de un nodo en un árbol AVL
    - *l*) Ejercicios propuestos
8. ESTRUCTURAS DE DATOS NO LINEALES. ARBOLES N-ARIOS
    - *a*) Introducción
    - *b*) Definiciones y conceptos básicos
    - *c*) El TAD ArbolN
    - *d*) Implementación del TAD ArbolN
    - *e*) Ejercicios propuestos
9. ARBOL1-2-3: UN ÁRBOL TRIARIO ORDENADO
    - *a*) Introducción
    - *b*) Definiciones
    - *c*) El TAD ARBOL1-2-3
    - *d*) Implementación del TAD ARBOL1-2-3
    - *e*) Ejercicios propuestos
10. ARBOL2-3: UN ÁRBOL TRIARIO ORDENADO
    - *a*) Introducción

b) Definiciones
c) Un árbol B
d) El TAD ARBOL2-3
e) Implementación del TAD ARBOL2-3
f) Ejercicios propuestos

11. TRIE: CONJUNTO DE PALABRAS

a) Introducción
b) Definiciones
c) El TAD TRIE
d) Implementación del TAD TRIE
e) Ejercicios propuestos

12. CUADTREE: REPRESENTACIÓN DE IMÁGENES

a) Introducción
b) Definiciones
c) El TAD CUADTREE
d) Implementación del TAD CUADTREE
e) Ejercicios propuestos

13. ESTRUCTURA DINÁMICAS NO LINEALES: GRAFOS

a) Introducción
b) Definiciones
c) El TAD Grafo
d) Representación de los Grafos
    1) Matriz de adyacencia
    2) Implementación de la Matriz de Adyacencia
    3) Listas de adyacencia
    4) Implementación de la lista de Adyacencia
e) Recorridos de un Grafo
    1) Recorrido en anchura
    2) Recorrido en profundidad
f) Conexiones en un grafo
    1) Componentes conexas de un grafo
    2) Matriz de caminos, cierre transitivo
    3) Matriz de caminos y cierre transitivo
g) Matriz de caminos:Algoritmo de Warshall
h) Algoritmo de costos mínimos:Dijkstra
i) Algoritmo de Floyd
j) Ejercicios propuestos

1.2 Prueba de entornos

Definición 1.1 (Igualdad)

$$a = b$$

Según la definición 1.1, la igualdad...

Teorema 1.1

$$a = b$$

Ejemplo 1.1

$$a = b$$

Lema 1.1

$$a = b$$

Corolario 1.1

$$a = b$$

Una caja de comentario

$$a = b$$

1.2.1 Tablas

Iteración		
	x_i	$y_i = f(x_i)$
A	$x_0 = 0$	0
B	$x_1 = 0{,}75$	$-0{,}0409838$
C	$x_2 = 1{,}5$	1,31799

1.2.2 Ejemplo codigo fuente Java

```
1 package com.unicauca.ejemplo;
2 public class Hello {
3 //Comentario
4 /*Comentario*/
5 /**Comentario*/
6 public static void main(String[] args) {
7 System.out.println("Hola mundo");
8 }
9 }
```

2 Pilas

2.1 Definición de Pila

Una pila (stack en inglés) es una lista ordinal o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. Se aplica en multitud de ocasiones en informática debido a su simplicidad y ordenación implícita en la propia estructura.
La Figura 2.1 muestra la representación gráfica de una pila con sus operaciones fundamentales de apilar (push en inglés) y desapilar o retirar (pos en inglés.).
La pila es muy útil en situaciones cuando los datos deben almacenarse y luego recuperarse en orden inverso.

Definición 2.1 Pila

Una pila (stack en inglés) es una lista ordinal o estructura de datos en la que el modo de acceso a sus elementos es de tipo LIFO (del inglés Last In First Out, último en entrar, primero en salir) que permite almacenar y recuperar datos. FALTA REFERENCIA

2.2 El TAD Pila

A continuación se especifica el TAD de la Pila con sus operaciones fundamentales. Las operaciones apilar y desapilar son las más importantes. En seguida la especificación de cada operación del TAD al estilo C.

```
TAD Pila [ T ]
```

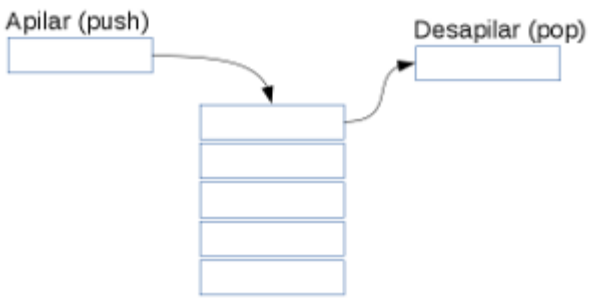


Figura 2.1. Representación de una Pila

```
{ invariante: TRUE }
Constructoras:
  crearPila:
Modificadoras:
   apilar: Pila T
   desapilar: Pila
Analizadoras:
   cima: Pila
   esVacia: Pila
Destructora:
   destruirPila: Pila

Pila crearPila( void )
/* Crea una pila vacia */
{ post: crearPila = \phi }

void apilar(Pila pil, T elem)
/* Coloca sobre el tope de la pila el elemento elem */\
{ post: pil = e1, e2, .. elem}

void desapilar(Pila pil)\\
/* Elimina el elemento que se encuentra en el tope de la pila */\\
{ pre: pil =e1, e2, ..en, n > 0 }
{ post: pil =e1, e2, .., en-1 }

T cima(Pila pil )
/* Retorna el elemento que se encuentra en el tope de la pila */
{ pre: n > 0 }
{ post: cima = en }

int esVacia( Pila pil )
/* Informa si la pila esta vacia */
{ post: esVacia = ( pil = \phi) }

void destruirPila( Pila pil )
/* Destruye la pila retornando toda la memoria ocupada */
{post: pil ha sido destruida }
```

2.3 Implementación del TAD Pila en Java

A continuación se muestra una implementación en Java del TAD Pila. Se ha implementado una pila dinámica utilizando nodos enlazados. Cada nodo almacena un valor y contiene una referencia al siguiente nodo. Supongamos una pila que almacene datos enteros, a la cual se le han aplicado las siguientes operaciones: apilar(10), apilar(35), apilar(12), apilar(7). La Figura 2.2 representa cómo quedaría la pila después de apilar en orden los datos 10, 35, 12 y 7. Se puede apreciar que cada nodo almacena un dato y a la vez existe una referencia que almacena la dirección del siguiente nodo. Además, existe una referencia llamada cabeza que apunta al

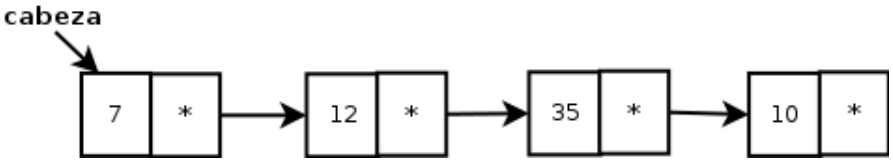


Figura 2.2. Representación de la Pila con Nodos Enlazados

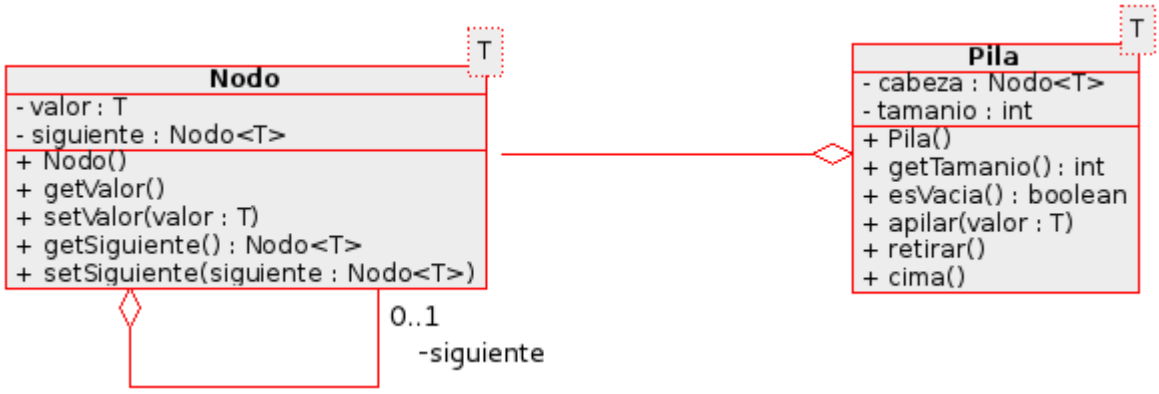


Figura 2.3. Diagrama de clase de la implementación del TAD Pila

último elemento que entró (en este caso al dato 7).

La Figura 2.3 muestra el diagrama de clases. Se puede apreciar básicamente dos clases: *Nodo* y *Pila*. La clase *Nodo* representa cada uno de los nodos enlazados que almacenan los objetos que se apilan. Tiene dos atributos, *valor* que representa el valor que guarda el nodo, en este caso es una referencia a un objetivo de tipo T (siendo T un tipo genérico). El atributo *siguiente*, representa la referencia al siguiente nodo. Los demás son únicamente, constructor y getters y setters de cada atributo. A continuación el código fuente de la clase Nodo.

```
1  package co.unicauca.pilas;
2  public class Nodo<T> {
3     //Atributo valor de tipo T. Almacena la referencia al objeto que se guarda
          en el nodo
4     private T valor;
5     //Referencia al siguiente nodo enlazado
6     Nodo<T> siguiente;
7     //Constructor por defecto
8     public Nodo() {
9        valor = null;
10       siguiente = null;
11    }
12    //Devuelve el valor
13    public T getValor() {
14       return valor;
15    }
16    //Modifica el valor
17    public void setValor(T valor) {
18       this.valor = valor;
19    }
```

```
20    //Devuelve el atributo siguiente
21    public Nodo<T> getSiguiente() {
22       return siguiente;
23    }
24     //Modifica el atributo siguiente
25    public void setSiguiente(Nodo<T> siguiente) {
26       this.siguiente = siguiente;
27    }
28 }
```

La clase *Pila* representa la pila como tal con sus operaciones principales de *apilar* y *retirar*. A continuación el código fuente de la clase Pila.

```
1  package co.unicauca.pilas;
2  public class Pila<T> {
3     //Atributo cabeza, que apunta al tope la pila
4     private Nodo<T> cabeza;
5     //Almacena el total de elemento de la pila
6     private int tamanio;
7     //Constructor por defecto
8     public Pila() {
9        cabeza = null;
10       tamanio = 0;
11    }
12    //Devuelve el total de elementos de la pila
13    public int getTamanio() {
14       return tamanio;
15    }
16    //Verifica si la pila esta vacia
17    public boolean esVacia() {
18       return (cabeza == null);
19    }
20    //Apila un elemento nuevo
21    public void apilar(T valor) {
22       //Crear un nuevo Nodo
23       Nodo<T> nuevo = new Nodo<T>();
24       //Fijar el valor dentro del nodo
25       nuevo.setValor(valor);
26       if (esVacia()) {
27          //Cabeza apunta al nodo nuevo
28          cabeza = nuevo;
29       } else {
30          //Se enlaza el campo siguiente de nuevo con la cabeza
31          nuevo.setSiguiente(cabeza);
32          //La nueva cabeza de la pila pasa a ser nuevo
33          cabeza = nuevo;
34       }
35       //Incrementa el tamanio porque hay un nuevo elemento en la pila
36       tamanio++;
```

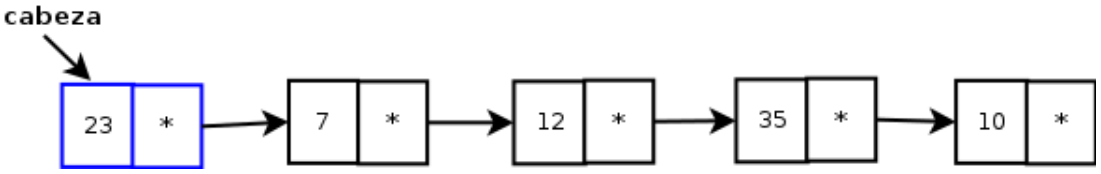


Figura 2.4. Representación de la Pila después de apilar el elemento 23


```
37    }
38    //Elimina un elemento de la pila
39    public void retirar() {
40       if (!esVacia()) {
41          cabeza = cabeza.getSiguiente();
42          tamanio--;
43 .     }
44    }
45    //Devuelve el elemento almacenado en el tope de la pila
46    public T cima() {
47       if (!esVacia())
48          return cabeza.getValor();
49       else
50          return null;
51    }
52 }
```

En la línea 21 se puede apreciar el método **apilar**. Este método recibe como parámetro el **valor** que quiere apilar. Lo primero que se hace es crear un *nuevo* nodo y fijar su valor (lineas 23 y 24). Una vez creado el nodo, se pregunta si la pila es vacía (linea 26). En caso verdadero la referencia *cabeza* apunta al nodo nuevo (linea 28). De lo contrario se enlaza el campo *siguiente* de nuevo con la *cabeza* (linea 31), y finalmente la *cabeza* apunta a *nuevo* (linea 33) para indicar cual fue el último nodo que entró a la pila. Finalmente, la variable *tamanio* se incrementa (indicando que la pila tiene una elemento mas). Por ejemplo si a la pila de la Figura 2.2 le aplicáramos la operación apilar(23), nos daría como resultado la pila de la Figura 2.4.

En la linea 39 se aprecia el método **retirar** (o desapilar). Lo primero que hace es preguntar si la pila no esta vacía (linea 40), en caso afirmativo, la *cabeza* se desplaza al siguiente nodo (linea 41). Finalmente se decrementa la variable *tamanio* (indicando que hay un elemento menos en la pila).

A continuación el código de un Cliente que instancia la Pila que hemos creado. En este caso se almacenan objetos de tipo entero, se apilan algunos números, se imprimen los valores del tope de pila y se desapilan sus elementos.

```
1  package co.unicauca.pilas;
2  public class ClienteMain {
3     public static void main(String[] args) {
4        //Crear una nueva pila de enteros
5        Pila<Integer> pila2 = new Pila<Integer>();
6        //Se apilan algunos datos enteros
7        pila2.apilar(2);
8        pila2.apilar(5);
```

```
9       pila2.apilar(7);
10      System.out.println("El tope de la pila es: " + pila2.cima());
11      //Se desapila
12      pila2.retirar();
13      System.out.println("El tope de la pila es: " + pila2.cima());
14      //Se desapila
15      pila2.retirar();
16      System.out.println("El tope de la pila es: " + pila2.cima());
17      //Se desapila, como la pila esta vacia devuelve null
18      pila2.retirar();
19      System.out.println("El tope de la pila es: " + pila2.cima());
20      //Probar con otra pila, donde se almacenen objetos
21   }
22 }
```

La salida por consola de este programa sería la siguiente.

```
El tope de la pila es: 7
El tope de la pila es: 5
El tope de la pila es: 2
El tope de la pila es: null
```

2.4 La clase Stack de java

La API de java ya trae implementada la pila mediante la clase *java.util.StackStack*. Esta clase es muy sencilla y al crear un objeto de tipo Stack con el constructor básico crea una pila con ningún elemento.

Las operaciones básicas de la clase Stack son *push* que introduce un elemento en la pila, *pop* que saca un elemento de la pila, *peek* que consulta el primer elemento de la cima de la pila *empty* que comprueba si la pila está vacía y *search* que busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella.

Entonces la pregunta que nos podemos hacer es, ¿si java ya trae una pila por qué implementar una propia mediante nodos enlazados? La respuesta es simple, porque implementando nuestra propia pila sabremos cómo funciona internamente una pila y habrá más aprendizaje. A continuación se muestra el mismo ejemplo de la sección Implementación del TAD en Java pero utilizando la clase Stack de java.

```
1  /* Ejemplo del uso de la clase Stack*/
2  import java.util.Stack;
3  public class EjemploStack {
4     public static void main(String arg[]) {
5        //Crea una nueva pila de enteros
6        Stack<Integer> pila = new Stack<Integer>();
7        //Se apilan algunos datos enteros
8        pila.push(2);
9        pila.push(5);
10       pila.push(7);
11       System.out.println("El tope de la pila es: " + pila.peek());
```

```
12        //Se desapila un elemento
13        pila.pop();
14        System.out.println("El tope de la pila es: " + pila.peek());
15        //Se desapila un elemento
16        pila.pop();
17        System.out.println("El tope de la pila es: " + pila.peek());
18        //Se desapila
19        pila.pop();
20        //Como la pila esta vacia dispara la excepcion java.util.
              EmptyStackException
21        System.out.println("El tope de la pila es: " + pila.peek());
22    }
23 }
```

2.5 Problemas que se resuelven con Pilas

Las pilas son muy útiles en situaciones cuando los datos deben almacenarse y luego recuperarse en orden inverso. A continuación se ilustran algunos ejemplos.

2.5.1 Evaluación de la correspondencia de delimitadores

. Un problema que se puede solucionar utilizando una pila, es la correspondencia de delimitadores en un programa. Esto es un ejemplo importante debido a que la correspondencia de delimitadores es parte de cualquier compilador: ningún programa se considera correcto si los delimitadores no tienen su pareja. Por ejemplo, la instrucción:

```
while (m<(n[8]+o)) { p = 7; /* comentarios */ 6=6;}
```

Se puede apreciar que está bien construida pues todos los paréntesis izquierdos corresponden con sus paréntesis derechos, así como las llaves y comentarios.
El algoritmo que resuelve la correspondencia adecuada de delimitadores evalúa la expresión de izquierda a derecha y sigue los siguientes pasos:

1. Obtener el carácter de la expresión y repetir pasos 2 al 3 para cada carácter.
2. Si es un operador de apertura: (, , [, /* se lo apila.
3. Si es un operador de cierre:), ,], */:
   a) Comparar que corresponda con el operador del tope de la pila.
   b) Si no corresponde, termina el programa y la expresión es incorrecta.
   c) Si hay correspondencia, se elimina elemento del tope de la pila y volver al paso 1.
4. Si al terminar de evaluar la expresión quedan elementos en la pila, la expresión es incorrecta.
5. Si al terminar de evaluar la expresión la pila queda vacía, la expresión es correcta.
6. Fin de algoritmo.

2.5.2 Evaluación de expresiones aritméticas

Las pilas se utilizan para evaluar expresiones aritméticas. Por ejemplo:

$$\frac{(100+23)*231}{(31-14)^2} - (34-12)$$

Para evaluar este tipo de expresiones se deben pasar a expresiones en *notación postfija* y luego aplicar un algoritmo para evaluar la expresión en notación postfija. Para entender este tipo de notaciones se sugiere ver el siguiente cuadro de definiciones.

Definición 2.2 Notaciones

La expresión **A+B** se dice que esta en *notación infija*, y su nombre se debe a que el operador + está entre los operandos A y B.
Dada la expresión **AB+** se dice que esta en *notación postfija* y su nombre se debe a que el operador + esta después de los operandos A y B.
Dada la expresión **AB** se dice que esta en *notación prefija*, y su nombre se debe a que el operador + está antes que los operandos A y B.

La ventaja de usar expresiones en notación postfija es que no son necesarios los paréntesis para indicar orden de operación, ya que éste queda establecido por la ubicación de los operadores con respecto a los operandos, Por ejemplo,

- Expresión infija: (X + Z) * W / T ^Y – V
- Expresión postfija: X Z + W * T Y ^/ V –

Para convertir una expresión dada en notación infija a una notación postfija, deberá establecerse previamente ciertas condiciones:

- Solamente se manejarán los siguientes operadores (Están dados ordenadamente de mayor a menor según su prioridad de ejecución):

Prioridad de Operadores		
Operador	**Prioridad dentro de la pila**	**Prioridad fuera de la pila**
^ : potencia	3	3
* / : Multiplicación y división	2	2
+ - : Suma y resta	1	1
(: Paréntesis izquierdo	0	4

- Los operadores de más alta prioridad se ejecutan primero. Si hubiera en una expresión dos o más operadores de igual prioridad, entonces se procesan de izquierda a derecha. Las subexpresiones parentizadas tendrán más prioridad que cualquier operador.
- Obsérvese que no se trata el paréntesis derecho ya que éste provoca sacar operadores de la pila hasta el paréntesis izquierdo.

Algoritmo para Convertir una expresión infija a postfija

El algoritmo para convertir una expresión aritmética de notación infija a notación postfija debe tener en cuenta las siguientes consideraciones:

- Se parte de una expresión en notación infija que tiene operandos, operadores y puede tener paréntesis. Los operandos vienen representados por letras y los operadores son: ^, *, /, +, -
- La transformación se realiza utilizando una pila en la cual se almacenan los operadores y los paréntesis izquierdos.
- La expresión aritmética se va leyendo desde el teclado de izquierda a derecha, caracter a caracter, los operandos pasan directamente a formar parte de la expresión en postfija la cual se guarda en un arreglo.
- Los operadores se meten en la pila siempre que ésta esté vacía, o bien siempre que tengan mayor prioridad que el operador de la cima de la pila (o bien si es la máxima prioridad).
- Si la prioridad es menor o igual se saca el elemento cima de la pila y se vuelve a hacer comparación con el nuevo elemento cima.
- Los paréntesis izquierdos siempre se meten en la pila; dentro de la pila se les considera de mínima prioridad para que todo operador que se encuenetra dentro del paréntesis entre en la pila.
- Cuando se lee un paréntesis derecho hay que sacar todos los operadores de la pila pasando a formar parte de la expresión postfija, hasta llegar a un paréntesis izquierdo, el cual se elimina ya que los paréntesis no forman parte de la expresión postfija.
- El proceso termina cuando no hay más elementos de la expresión y la pila esté vacía.

Con las anteriores consideraciones el algoritmo de conversión de expresiones infijas a postfijas es el siguiente:

1. Obtener caracteres de la expresión y repetir pasos 2 al 5 para cada caracter.
2. Si es un operando pasarlo a la expresión postfija
3. Si es un operador:
   a) Si la pila está vacía, meterlo en la pila. Repetir a partir del paso 1.
   b) Si la pila no está vacía:
      1) Si la prioridad del operador leído es mayor que la prioridad del operador cima de la pila, meterlo en la pila y repetir a partir del paso 1.
      2) Si la prioridad del operador es menor o igual que la prioridad del operador de la cima, sacar operador cima de la pila y pasarlo a la expresión postfija. Volver al paso 3.
4. Si es paréntesis derecho:
   a) Sacar operador cima de la pila y pasarlo a la expresión postfija.
   b) Si nueva cima es paréntesis izquierdo, suprimir elemento cima.

c) Si cima no es paréntesis izquierdo volver a paso 4.1

d) Volver a partir del paso 1.

5. Si es paréntesis izquierdo pasarlo a la pila.
6. Si quedan elementos en la pila, pasarlos a la expresión postfija
7. Fin de algoritmo

Como se puede apreciar el algoritmo anterior utiliza como mecanismo central el uso de pilas.

Evaluación de la Expresión en notación postfija

Una vez convertida la expresión infija a notación postifja se puede aplicar un algoritmo (el cual también maneja una pila) para evaluar dicha expresión y calcular su valor. Para ello, se almacena la expresión aritmética transformada a notación postfija en un vector, en la que los operandos están representados por variables de una sola letra. Antes de evaluar la expresión requiere dar valores numéricos a los operandos. Una vez que se tiene los valores de los operandos, la expresión es evaluada. El algoritmo de evaluación utiliza una pila de operandos de números reales. El algoritmo se describe a continuación.

1. Examinar el vector desde el elemento 1 hasta el N, repetir los pasos 2 y 3 para cada elemento del vector.
2. Si el elemento es un operando meterlo en la pila.
3. Si el elemento es un operador, lo designamos por ejemplo con +:

   a) Sacar los dos elementos superiores de la pila, los denominamos con los identificadores x,y respectivamente.

   b) Evaluar x+y; el resultado es z = x + y.

   c) El resultado z, meterlo en la pila.

   d) Repetir a partir del paso 1.
4. El resultado de la evaluación de la expresión está en el elemento cima de la pila.
5. Fin de algoritmo.

2.6 Ejercicios Propuestos

A continuación se plantean los siguientes ejercicios los cuales utilizan pilas como estructura central.

1. Utilizando el algoritmo de conversión de infijas a postfijas, pasar a notación postfija las siguientes expresiones:

   a) (X + Y * Q) / R^T *Respuesta:* X Y Q * + R T ^/

   b) X ^Y + Z - A * D - R *Respuesta:* X Y ^Z + A D * - R -

   c) A + (B - D) / Y – (A - C) / (Y - X) + Z *Respuesta:* A B D -Y / + A C - Y X - / - Z +
2. De las anteriores expresiones postfijas, dar valores numéricas a las variables y aplicar el algoritmo que permite evaluar una expresion postfijas.

3. Buscar en Internet un algoritmo que permita pasar expresiones de notación infija a notación prefija. Entender el algoritmo mediante ejemplos.

4. Proponga una implementación en java (o cualquier lenguaje) del algoritmo que evalúa la correspondencia de limitadores.

5. Proponga una implementación en java (o cualquier lenguaje) del algoritmo que pasa expresiones de notación infija a postija.

6. Proponga una implementación en java (o cualquier lenguaje) del algoritmo que evalúa expresiones en notación postfija.

3 Colas

3.1 Definición de Cola

Una cola es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo y la operación de extracción por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir. Este tipo de estructura de datos abstracta se implementa en lenguajes orientados a objetos mediante clases, en forma de nodos enlazados.
La Figura 3.1 muestra la representación gráfica de una cola con sus operaciones fundamentales de encolar y desencolar.

Definición 3.1 Cola

Una cola es una estructura de datos, caracterizada por ser una secuencia de elementos en la que la operación de inserción se realiza por un extremo y la operación de extracción por el otro. También se le llama estructura FIFO (del inglés First In First Out), debido a que el primer elemento en entrar será también el primero en salir.

3.2 El TAD Cola

A continuación se especifica el TAD de la Cola (al estilo C) con sus operaciones fundamentales. Las operaciones encolar y desencolar son las más importantes.

```
TAD Cola [ T ]
```

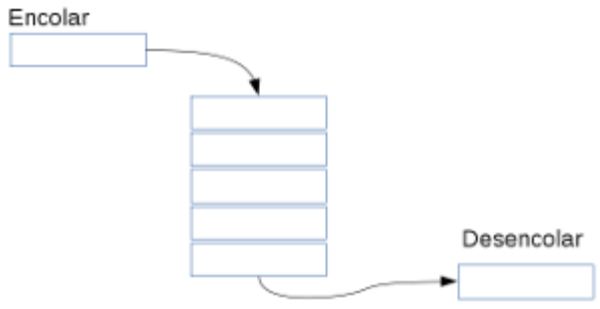


Figura 3.1. Representación de una Cola

```
{ invariante: TRUE }
Constructoras:
  crearCola:
Modificadoras:
   encolar: Cola T
   desencolar: Cola
Analizadoras:
   frente: Cola
   esVacia: Cola
Destructora:
   destruirCola: Cola

Cola crearCola( void )
/* Crea una cola vacia */
{ post: crearCola = \phi }

void encolar(Cola col, T elem)
/* Agrega elem al final de la cola*/
{ post: col = e1, e2, .. elem}

void desencolar(Cola col)\\
/* Elimina el primer elemento de la cola */
{ pre: n > 0 }
{ post: col = e2, .., en }

T frente(Cola col )
/* Retorna el primer elemento de la cola */
{ pre: n > 0 }
{ post: frente = e1 }

int esVacia( Cola col )
/* Informa si la cola esta vacia */

{ post: esVacia = ( col = \phi) }

void destruirCola( Cola col )
/* Destruye la cola retornando toda la memoria ocupada */
{post: col ha sido destruida }
```

3.3 Implementación del TAD Cola en Java

A continuación se muestra una implementación en Java del TAD Cola. Se ha implementado una cola dinámica utilizando nodos enlazados. Cada nodo almacena un valor y contiene una referencia al siguiente nodo. Supongamos una cola que almacene datos enteros, a la cual se le han aplicado las siguientes operaciones: encolar(10), encolar(35), encolar(12), encolar(7). La Figura 3.2 representa cómo quedaría la cola después de encolar en orden los datos 10, 35, 12 y 7. Se puede apreciar que cada nodo almacena un dato y a la vez existe una referencia que

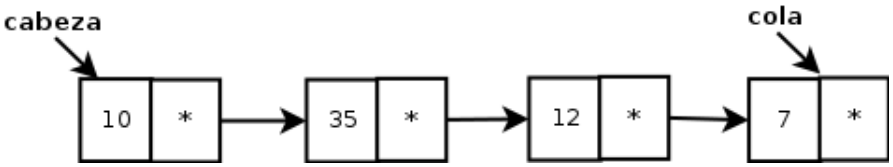


Figura 3.2. Representación de la Cola con Nodos Enlazados

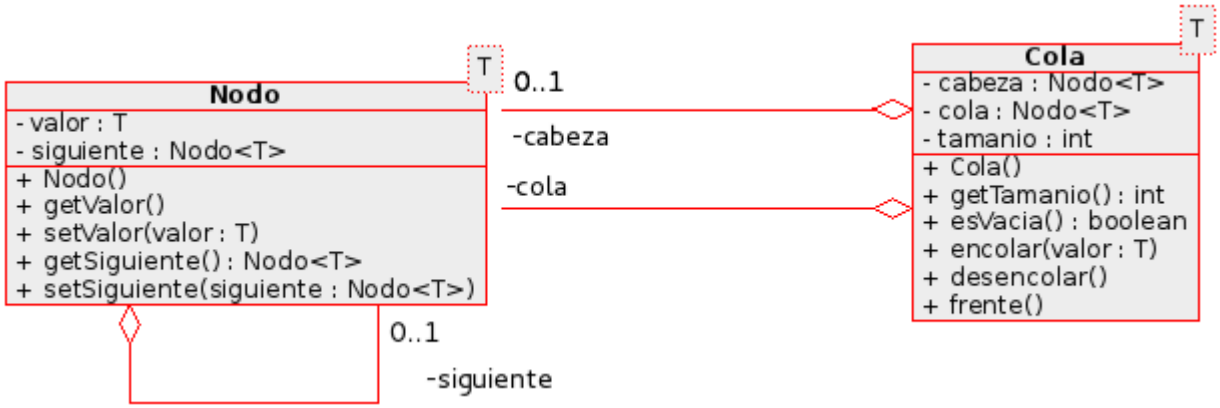


Figura 3.3. Diagrama de clase de la implementación del TAD Cola

almacena la dirección del siguiente nodo. Además, existe una referencia llamada cabeza que apunta al primer elemento que entró (en este caso al dato 10) y una referencia cola que apunta al último dato que entró (o sea el 7).

La Figura 3.3 muestra el diagrama de clases que implementa el TAD Cola. Se puede apreciar básicamente dos clases: *Nodo* y *Cola*. La clase *Nodo* representa cada uno de los nodos enlazados que almacenan los objetos que se encolan. Tiene dos atributos, *valor* que representa el valor que guarda el nodo, en este caso es una referencia a un objetivo de tipo T (siendo T un tipo genérico). El atributo *siguiente*, representa la referencia al siguiente nodo. Los demás son únicamente, constructor y getters y setters de cada atributo. A continuación el código fuente de la clase Nodo.

```
1  package co.unicauca.colas;
2  public class Nodo<T> {
3     //Atributo valor de tipo T. Almacena la referencia al objeto que se guarda
          en el nodo
4     private T valor;
5     //Referencia al siguiente nodo enlazado
6     Nodo<T> siguiente;
7     //Constructor por defecto
8     public Nodo() {
9        valor = null;
10       siguiente = null;
11    }
12    //Devuelve el valor
13    public T getValor() {
14       return valor;
15    }
16    //Modifica el valor
```

```
17    public void setValor(T valor) {
18       this.valor = valor;
19    }
20    //Devuelve el atributo siguiente
21    public Nodo<T> getSiguiente() {
22       return siguiente;
23    }
24     //Modifica el atributo siguiente
25    public void setSiguiente(Nodo<T> siguiente) {
26       this.siguiente = siguiente;
27    }
28 }
```

La clase *Cola* representa la cola como tal con sus operaciones principales de *encolar* y *desencolar*. A continuación el código fuente de la clase Cola.

```
1  package co.unicauca.colas;
2  public class Cola<T> {
3     //Atributo cabeza, que apunta al primer elemento de la cola
4     private Nodo<T> cabeza;
5     //Atributo cola, que apunta al ultimo elemento de la cola
6     private Nodo<T> cola;
7     //Almacena el total de elemento de la cola
8     private int tamanio;
9     //Constructor por defecto
10    public cola() {
11       cabeza = null;
12       cola=null;
13       tamanio = 0;
14    }
15    //Devuelve el total de elementos de la cola
16    public int getTamanio() {
17       return tamanio;
18    }
19    //Verifica si la cola esta vacia
20    public boolean esVacia() {
21       return (cabeza == null);
22    }
23    //Encolar un elemento nuevo
24    public void encolar(T valor) {
25       //Crear un nuevo Nodo
26       Nodo<T> nuevo = new Nodo<T>();
27       //Fijar el valor dentro del nodo
28       nuevo.setValor(valor);
29       if (esVacia()) {
30          //cabeza y cola apuntan al nodo nuevo
31          cabeza = nuevo;
32          cola = nuevo;
33       } else {
```

```
34          //Se enlaza el campo siguiente de cola con el nuevo elemento
35          cola.setSiguiente(nuevo);
36          //La nueva cola pasa a ser nuevo
37          cola = nuevo;
38       }
39       //Incrementa el tamanio porque hay un nuevo elemento en la cola
40       tamanio++;
41    }
42    //Elimina el primer elemento de la cola
43    public void desencolar() {
44       //si no es vacia
45       if (!esVacia()) {
46          //verificar si hay un solo elemento en la cola
47          if (cabeza == cola) {
48             cabeza = null;
49             cola = null;
50          } else {
51             //se elimina el primer elemento de la cola, desplazando la cabeza
                   al siguiente nodo
52             cabeza = cabeza.getSiguiente();
53          }
54          //Se disminuye el total de elementos
55          tamanio--;
56       }
57     }
58
59    //Devuelve el primer elemento de la cola
60    public T frente() {
61       if (!esVacia())
62          return cabeza.getValor();
63       else
64          return null;
65    }
66 }
```

En la línea 24 se puede apreciar el método **encolar**. Este método recibe como parámetro el **valor** que quiere encolar. Lo primero que se hace es crear un *nuevo* nodo y fijar su valor (lineas 26 y 28). Una vez creado el nodo, se pregunta si la cola es vacía (linea 29). En caso verdadero las referencias *cabeza* y *cola* apuntan al nodo nuevo (lineas 31 y 32). De lo contrario se enlaza el campo *siguiente* de *cola* con la referencia *nuevo* (linea 35), en seguida la *cola* apunta a *nuevo* (linea 37) para indicar cual fue el último nodo que entró a la cola. Finalmente, la variable *tamanio* se incrementa (indicando que la cola tiene una elemento más). Por ejemplo si a la cola de la Figura 3.2 le aplicáramos la operación encolar(23), nos daría como resultado la cola de la Figura 3.4.

En la linea 43 se aprecia el método **desencolar**. Lo primero que hace es preguntar si la cola no esta vacía (linea 45), en caso afirmativo, si cabeza es igual a cola significa que hay un sólo elemento, por lo tanto, la *cabeza* y *cola* se les asigna null y de esta forma la cola quedaría vacía. Si *cabeza* no es igual a *cola*, significa que hay más de un elemento en la cola, por lo tanto,

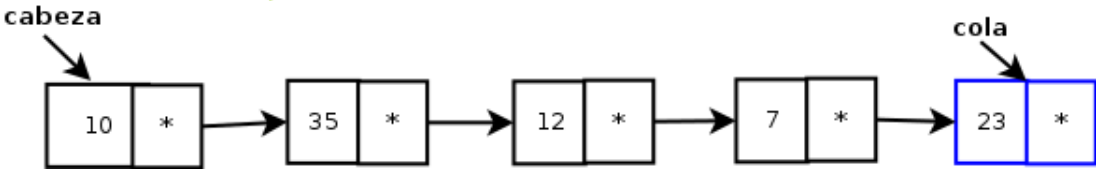


Figura 3.4. Representación de la cola después de encolar el elemento 23

simplemente de desplaza la *cabeza* al siguiente nodo (line 52). Finalmente se decrementa la variable *tamanio* (indicando que hay un elemento menos en la cola).
A continuación el código de un Cliente que instancia la cola que hemos creado. En este caso se almacenan objetos de tipo entero, se encolan algunos números, se imprimen los valores del frente de la cola y se desencolan sus elementos.

```
1  package co.unicauca.colas;
2  public class ClienteMain {
3     public static void main(String[] args) {
4        //Crear una nueva cola de enteros
5        Cola<Integer> cola = new Cola<Integer>();
6        //Se encolan algunos elementos
7        cola.encolar(12);
8        cola.encolar(13);
9        cola.encolar(14);
10       cola.encolar(15);
11       //Se imprime el primer elemento de la cola
12       System.out.println("El primer elemento de la cola es:" + cola.frente());
13
14       cola.desencolar();
15       System.out.println("El primer elemento de la cola es:" + cola.frente().
             toString());
16
17       cola.desencolar();
18       System.out.println("El primer elemento de la cola es:" + cola.frente());
19
20       cola.desencolar();
21       System.out.println("El primer elemento de la cola es:" + cola.frente());
22
23       cola.desencolar();
24       System.out.println("El primer elemento de la cola es:" + cola.frente());
25    }
26 }
```

La salida por consola de este programa sería la siguiente.

```
El primer elemento de la cola es: 12
El primer elemento de la cola es: 13
El primer elemento de la cola es: 14
El primer elemento de la cola es: 15
El primer elemento de la cola es: null
```

3.4 La clase Queue de java

La API de java ya trae implementada la cola mediante la interface *java.util.Queu.*
Las operaciones básicas son...
Entonces la pregunta que nos podemos hacer es, ¿si java ya trae una cola por qué implementar una propia mediante nodos enlazados? La respuesta es simple, porque implementando nuestra propia cola sabremos cómo funciona internamente una cola y habrá más aprendizaje. A continuación se muestra el mismo ejemplo de la sección Implementación del TAD en Java pero utilizando la interface Queue de java.

```
1 /* Ejemplo del uso de la interface Queue*/
2 import java.util.Queue;
3 public class EjemploQueue{
4    public static void main(String arg[]) {
5       //Crea una nueva cola de enteros
6       //FALTA CODIGO
7    }
8 }
```

3.5 Problemas que se resuelven con colas

Las colas se utilizan en sistemas informáticos, transportes y operaciones de investigación (entre otros), donde los objetos, personas o eventos son tomados como datos que se almacenan y se guardan mediante colas para su posterior procesamiento. A continuación se describen problemas del mundo real donde la estructura fundamental se resuelve gracias a una cola.

3.5.1 Simulador del despegue de aviones

Se debe utilizar una cola para simular el despegue de aviones del aeropuerto el Dorado de Bogotá, las políticas que rigen las operaciones aéreas son las siguientes:

- Se tiene estimado que del aeropuerto pueden salir vuelos cada cinco minutos, siempre y cuando no hayan solicitudes de aterrizaje.
- Un vuelo no puede salir del aeropuerto, antes de la hora programada.
- Un vuelo no puede despegar del aeropuerto si se encuentran vuelos pendientes por despegar (se debe respetar la cola de vuelos).
- El aeropuerto conoce con anterioridad la hora de salida de todos los vuelos programados para ese día. Las aerolíneas tienen programados vuelos con 10 minutos de diferencia.
- Los aterrizajes ocurren de manera aleatoria. El aterrizaje tarda diez minutos (es prioritario el aterrizaje de un avión).
- Se asume que la simulación inicia con mínimo 10 solicitudes de despegue.

A continuación un bosquejo de cómo podría ser la simulación.

```
1    public void simular(){
2       //Crea la cola de vuelos
3       Cola<Vuelo> cola = new Cola<Vuelo>();
```

```
4       //Lee los vuelos desde el archivo y los carga en cola
5       cola = leerVuelosDesdeArchivo();
6       //Crea una instancia de tipo Proceso
7       Proceso instancia = null;
8       //Mientras haya procesos en la cola
9       while (!cola.esVacia()){
10         instancia=cola.desencolar();
11         if(hayAterrizaje()){
12            //Simula el aterrizaje
13         }else{
14            //Simula el despeje
15         }
16      }
17   }
```

3.5.2 Simulador de la planificación de procesos round-robin

Utilizando una cola de procesos, simular la planificación de procesos round-robin (turno circular) de un sistema operativo. A continuación se describe su funcionamiento.
Los sistemas operativos modernos permiten ejecutar simultáneamente dos o más procesos. Cuando hay más de un proceso ejecutable, el sistema operativo debe decidir cuál proceso se ejecuta primero. La parte del sistema operativo que toma esta decisión se llama planificador; el algoritmo que usa se denomina algoritmo de planificación.
El algoritmo de planificación de procesos Round Robin es uno de los más sencillos y antiguos REFERENCIA LIBRO SISTEMAS OPERATIVOS. A cada proceso se le asigna un intervalo de tiempo en la CPU, llamado cuanto, durante el cual se le permite ejecutarse. Si el proceso todavía se está ejecutando al expirar su cuanto, el sistema operativo se apropia de la CPU y se la da a otro proceso. Si el proceso se bloquea o termina antes de expirar el cuanto , se hace la conmutación de proceso. Para ello, el planificador mantiene una lista de procesos ejecutables. Cuando un proceso gasta su cuanto, se le coloca al final de la lista. A continuación un bosquejo de cómo podría ser la simulación.

```
1    public void simular(int cuanto){
2       //Crea la cola de procesos
3       Cola<Proceso> cola = new Cola<Proceso>();
4       //Lee los procesos desde el archivo y los carga en cola
5       cola = leerProcesosDesdeArchivo();
6       //Crea una instancia de tipo Proceso
7       Proceso instancia = null;
8       //Simula el tiempo
9       int tiempo=0;
10      //Mientras haya procesos en la cola
11      while (!cola.esVacia()){
12         instancia=cola.desencolar();
13         System.out.println("Tiempo " + tiempo + ": " + instancia.getId() + "
               entra a ejecucion);
14         //Completar el codigo faltante
15      }
16   }
```

3.6 Ejercicios Propuestos

A continuación se plantean los siguientes ejercicios los cuales utilizan la estructura de datos Cola.

1. Elabore una implementación en java del simulador del despegue y aterrizaje de aviones. Un archivo *vuelos.txt* debe contener la programación ordenada por hora de salida de los vuelos de un día cualquiera. El formato debe ser así (numero de vuelo, aerolinea, horas, minutos), por ejemplo:
   10071,AVIANCA,7,0
   10072,AVIANCA,7,10
   10073,AVIANCA,7,20
   10073,SATENA,7,30
   10074,SATENA,7,40
   10075,AVIANCA,7,50
   10076,LAN,8,0
   10077,TAME,8,10
   10078,TAME,8,20

   Un ejemplo de la salida de la simulación podría ser:
   Despegue, vuelo:10071 Hora programada:7:00 Hora real:7:00
   Despegue, vuelo:10072 Hora programada:7:10 Hora real:7:10
   Aterrizaje a las 7:15
   Despegue, vuelo:10073 Hora programada:7:20 Hora real:7:25
   Despegue, vuelo:10073 Hora programada:7:30 Hora real:7:30
   Aterrizaje a las 7:35
   Aterrizaje a las 7:45
   Despegue, vuelo:10074 Hora programada:7:40 Hora real:7:55
   Aterrizaje a las 8:0
   Despegue, vuelo:10075 Hora programada:7:50 Hora real:8:10
   Despegue, vuelo:10076 Hora programada:8:00 Hora real:8:15
   Despegue, vuelo:10077 Hora programada:8:10 Hora real:8:20
   Despegue, vuelo:10078 Hora programada:8:20 Hora real:8:25
   Despegue, vuelo:10079 Hora programada:9:30 Hora real:8:30

2. Elabore una implementación en java del simulador de la planificación de procesos round-robin. El programa debe simular la operación de la ejecución de procesos a partir de la información de un archivo de texto *procesos.txt* que contiene la información básica de cada proceso (El nombre del proceso, el tiempo de ejecución).Un posible contenido de este archivo puede ser:
   P1, 100
   P2,15
   P3,40

   Una posible salida resultado de la ejecución de la simulación para un cuanto de 20 ms, con los tres procesos descritos anteriormente podría sería:
   Tiempo 0: P1 entra a ejecución.

Tiempo 20: P1 se conmuta. Pendiente por ejecutar 80 ms
Tiempo 20: P2 entra a ejecución
Tiempo 35: P2 se conmuta. Pendiente por ejecutar 0 ms
Tiempo 35: P3 entra a ejecución
Tiempo 55: P3 se conmuta. Pendiente por ejecutar 20 ms
Tiempo 55: P1 entra a ejecución
Tiempo 75: P1 se conmuta. Pendiente por ejecutar 60 ms
Tiempo 75: P3 entra a ejecución
Tiempo 95: P3 se conmuta. Pendiente por ejecutar 0 ms.
Tiempo 95: P1 entre en ejecución
Tiempo 155: P1 termina su ejecución
Total tiempo de ejecución de todos los procesos:155 ms
Total de conmutaciones: 4

Bibliografía

[1] G. Kappel, B. Pröll, S. Reich, and W. Retschitzegger, *Web Engineering: The Discipline of Systematic Development of Web Applications*. Wiley, 2006.