Apunte 2 - Estructuras Lineales

Índice general

2. Estructuras lineales **6**
2.1. Pilas . . . 6
2.1.1. Descripción . . . 6
2.1.2. Operaciones del TDA Pila . . . 7
2.1.3. Implementación . . . 7
2.1.3.1. Implementación estática . . . 8
2.1.3.2. Implementación dinámica . . . 9
2.1.4. Análisis de eficiencia del TDA Pila . . . 12
2.2. Colas . . . 13
2.2.1. Definición . . . 13
2.2.2. Operaciones . . . 13
2.2.3. Implementaciones . . . 14
2.2.3.1. Implementación estática . . . 14
2.2.3.2. Implementación dinámica . . . 16
2.2.4. Análisis de eficiencia del TDA Cola . . . 17
2.3. Listas . . . 18
2.3.1. Descripción . . . 18
2.3.2. Operaciones . . . 18
2.3.3. Implementación . . . 19
2.3.4. Análisis de la eficiencia . . . 24

Índice de figuras

2.1. Representación gráfica de una pila . . . 6
2.2. Representación gráfica de las operaciones sobre pila . . . 7
2.3. Representación gráfica de una pila implementada de manera estática . . . 8
2.4. Diagrama de clases UML para pila (implementación estática) . . . 8
2.5. Representación gráfica de una pila implementada de manera dinámica . . . 9
2.6. Representación gráfica de un nodo . . . 10
2.7. Diagrama de clases UML para Nodo (implementación dinámica de estructuras lineales) . . . 10
2.8. Diagrama de clases UML para pila (implementación dinámica) . . . 11
2.9. Cómo se avanza en una estructura lineal . . . 12
2.10. Representación gráfica de una cola . . . 13
2.11. Ejemplo de Cola implementada con un arreglo circular . . . 14
2.12. Ejemplo de cola circular llena . . . 14
2.13. Diagrama de clases UML para Cola (implementación estática) . . . 15
2.14. Ejemplo de Cola implementada de manera dinámica . . . 16
2.15. Poner un elemento en Cola (implementación dinámica) . . . 16
2.16. Sacar un elemento en cola (implementación dinámica) . . . 17
2.17. Diagrama de clases UML para Cola (implementación dinámica) . . . 17
2.18. Representación gráfica de una lista . . . 18
2.19. Ejemplos de las operaciones *localizar* y *recuperar* de TDA Lista . . . 19
2.20. Diagrama de clases UML para Lista (implementación dinámica) . . . 20
2.21. Insertar en Lista en la primera posición . . . 20
2.22. Insertar en Lista en una posición distinta de 1 . . . 21
2.23. Insertar en Lista en una posición distinta de 1 (cont.) . . . 21
2.24. Eliminar en Lista en la primera posición . . . 22
2.25. Eliminar en Lista en posición distinta de 1 . . . 22

Algoritmos

2.1. Código Java de clase Pila y método *constructor* (implementación estática) 8
2.2. Código Java método *apilar* de clase Pila (implementación estática) 9
2.3. Código Java de clase Nodo 10
2.4. Código Java de clase Pila con método *constructor* (implementación dinámica) 11
2.5. Código Java método *apilar* de clase Pila (implementación dinámica) 11
2.6. Código Java método *toString* de clase Pila (implementación dinámica) 12
2.7. Código Java del constructor de la clase Cola (implementación estática) 15
2.8. Código Java del método *sacar* de clase Cola (implementación estática) 15
2.9. Código Java del método *sacar* de clase Cola (implementación dinámica) 17
2.10. Código Java método *insertar* de clase Lista 22

Índice de ejercicios

2.1. Implementación del TDA Pila de manera estática 9
2.2. Implementación del TDA Pila de manera dinámica 13
2.3. Implementación del TDA Cola de manera estática 15
2.4. Implementación del TDA Cola de manera dinámica 18
2.5. Implementación y uso del TDA Lista 23

Apunte 2

Estructuras lineales

Actualizado: 9 de abril de 2021

En esta unidad trabajaremos sobre la siguiente pregunta principal:

> *¿Cuáles son las estructuras de datos más simples, qué se puede hacer con ellas y cómo se pueden implementar de forma eficiente?*

A lo largo de la unidad se presentarán tres tipos de estructuras de datos: Pilas, Colas y Listas; de las cuales se presentarán los conceptos básicos, usos y las operaciones más importantes de cada una, considerando la perspectiva de Tipo de Dato Abstracto (TDA) vista en la materia Desarrollo de Algoritmos.

Luego se presentarán distintas implementaciones, de tipo estático y dinámico, haciendo en cada caso el correspondiente análisis de eficiencia respecto al uso de memoria y al tiempo de ejecución de las distintas operaciones.

2.1. Pilas

2.1.1. Descripción

Una pila es una estructura de datos en la que el último elemento en entrar es el primero en salir. Se denominan estructuras LIFO (Last In, First Out). En esta estructura sólo se tiene acceso al tope o cima de la pila y no existe el concepto de posiciones. Todas las operaciones se realizan en el extremo de la estructura llamado *tope*. El tope de la pila corresponde al elemento que entró en último lugar, es el único elemento "visible" en cada momento y el que saldrá en caso de eliminarse un elemento. La Figura 2.1 muestra una estructura de pila.

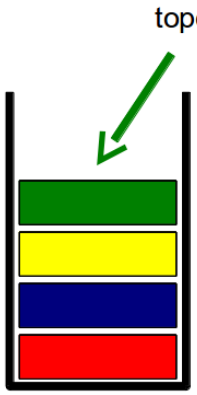
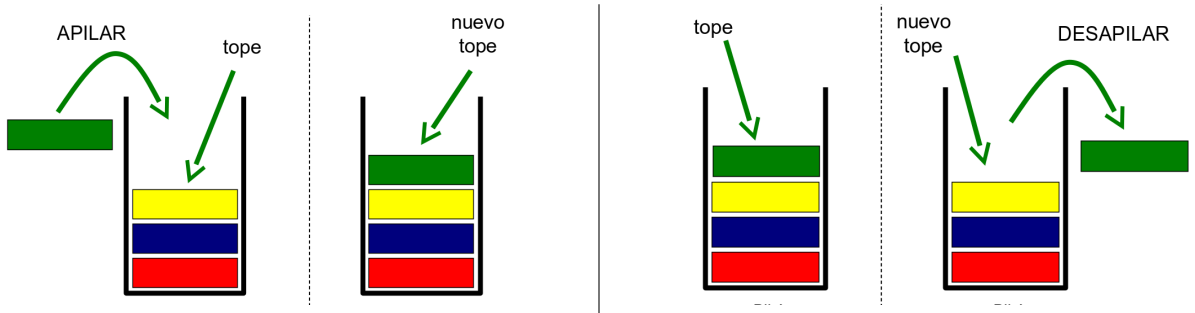


Figura 2.1: Representación gráfica de una pila

A continuación, la Figura 2.2A muestra cómo se realiza la operación de agregar un elemento (apilar) y la Figura 2.2B muestra la operación para sacar un elemento de la pila (desapilar). Como se puede observar, el tope de la pila se modifica en ambas operaciones, apuntando siempre al elemento que queda visible.



(A) Apilar un elemento nuevo (B) Desapilar el tope

Figura 2.2: Representación gráfica de las operaciones sobre pila

Las pilas son utilizadas para solucionar una amplia variedad de problemas en compiladores, sistemas operativos y otros programas en general. Un ejemplo de este tipo de software son los navegadores de Internet: Los browsers usan una pila para almacenar las direcciones de los sitios recientemente visitados y así implementar la funcionalidad "volver atrás" (backspace). También los editores de texto, que proporcionan normalmente un botón "deshacer" que cancela las operaciones de edición recientes en el orden inverso al que fueron hechas. Otro uso común es la pila de recursividad, en donde en cada llamado recursivo se apilan los valores de los parámetros y variables de ese llamado.

En general, utilizaremos pilas cada vez que necesitemos recuperar los elementos en el orden inverso al que han sido agregados.

2.1.2. Operaciones del TDA Pila

Las operaciones básicas del TDA Pila son:

- *constructor vacío*
  // Crea y devuelve la pila vacía.
- *apilar (nuevoElem):boolean*
  // Pone el elemento *nuevoElem* en el tope de la pila. Devuelve *verdadero* si el elemento se pudo apilar y *falso* en caso contrario.
- *desapilar():boolean*
  // Saca el elemento del tope de la pila. Devuelve *verdadero* si la pila no estaba vacía al momento de desapilar (es decir que se pudo desapilar) y *falso* en caso contrario.
- *obtenerTope() : elem*
  // Devuelve el elemento en el tope de la pila. *Precondición*: la pila no está vacía.
- *esVacia() : boolean*
  // Devuelve *verdadero* si la pila no tiene elementos y *falso* en caso contrario.
- *vaciar() : void*
  // Saca todos los elementos de la pila.
- *clone() : Pila*
  // Devuelve una copia exacta de los datos en la estructura original, y respetando el orden de los mismos, en otra estructura del mismo tipo
- *toString() : String*
  // Devuelve una cadena de caracteres formada por todos los elementos de la pila para poder mostrarla por pantalla. *Es recomendable utilizar este método únicamente en la etapa de prueba y luego comentar el código.*

2.1.3. Implementación

Veremos a continuación la implementación de la estructura Pila de dos maneras diferentes: estática y dinámica. En el primer caso se reserva un espacio fijo en memoria, mientras que en el segundo el espacio ocupado crece y decrece durante la ejecución del programa. Gracias al encapsulamiento, los detalles son privados al TDA y quienes la usan desconocen la estructura interna.

2.1.3.1. Implementación estática

Los elementos de la pila se almacenan en un arreglo de tamaño fijo. En una variable de tipo `int` se guarda la posición del elemento tope (o bien la cantidad de elementos). La pila crece y decrece dentro del arreglo previamente definido. Puede dar error de "estructura llena" si se intenta agregar más datos que la capacidad disponible en el arreglo.

En la Figura 2.3 se puede ver la representación de una pila de manera estática. La parte principal de la estructura es un arreglo donde se van almacenando los elementos a medida que llegan. En un atributo o variable *tope* se almacena la posición del elemento que llegó último. El tamaño máximo del almacenamiento se define en una constante para que sea sencillo modificarlo en una sola ubicación en el código.

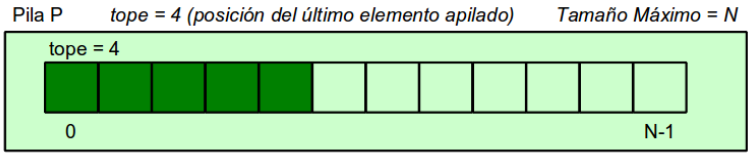


Figura 2.3: Representación gráfica de una pila implementada de manera estática

La representación en UML de la estructura Pila implementada con arreglos se muestra en la Figura 2.4.

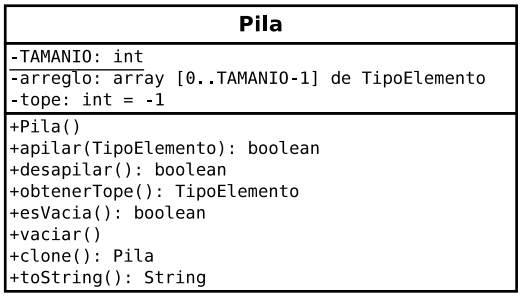


Figura 2.4: Diagrama de clases UML para pila (implementación estática)

A continuación se muestra un ejemplo en Java para el UML definido en la Figura 2.4, donde *TipoElemento* se reemplazó por `Object`[1]. El método *constructor* crea una instancia vacía de la Pila, para lo cual pone el tope en -1 (que es una posición inválida del arreglo). El tamaño (constante TAMANIO) se ha inicializado en 20.

Algoritmo 2.1 Código Java de clase Pila y método *constructor* (implementación estática)

```
public class Pila {

    private Object[] arreglo;
    private int tope;
    private static final int TAMANIO = 20;



    public Pila() {
        this.arreglo = new Object[TAMANIO];
        this.tope = -1;
    }
```

A continuación se muestra el método *apilar* de la clase Pila. En este caso, al tratarse de un espacio fijo, el método verifica si hay espacio disponible para agregar un elemento más, y luego lo agrega al final, actualizando el atributo *tope*.

[1] Para comprender mejor cómo funciona Object como tipo genérico, lea la introducción y sección 1 del Anexo *"Cómo implementar estructuras para un tipo de elemento genérico"*

Algoritmo 2.2 Código Java método *apilar* de clase Pila (implementación estática)

```
public boolean apilar(Object nuevoElem){

    boolean exito;

    if (this.tope+1 >= this.TAMANIO)
        // Error: pila llena
        exito = false;
    else{
        // pone el elemento en el tope de la pila e incrementa tope
        this.tope++;
        this.arreglo[tope]=nuevoElem;
        exito = true;
    }

    return exito;
}
```

Ejercicio 2.1: Implementación del TDA Pila de manera estática

1. Crear un paquete *lineales.estaticas* e implementar la clase Pila, incluyendo todas las operaciones del TDA Pila.
2. Crear un paquete *tests.lineales* e implementar la clase TestPila que permita probar todas las operaciones de la clase Pila anterior con números enteros.
3. En la clase TestPila escriba un método que, dada una pila llena con dígitos (0..9), verifique si la secuencia forma un número capicúa o no. *Nota*: Utilizar una pila auxiliar para facilitar la operación.
4. Extienda la clase TestPila para comprobar que la misma clase Pila también funciona con elementos de tipo String.

La ventaja de implementar una pila en forma estática es que se accede inmediatamente al elemento del tope. Las desventajas son, primero: el arreglo tiene una capacidad máxima fija predefinida, y segundo: se debe reservar espacio (posiciones vacías del arreglo) que no siempre se usa en su totalidad. Estos inconvenientes se eliminan utilizando la implementación dinámica que veremos a continuación.

2.1.3.2. Implementación dinámica

Para implementar la Pila de manera dinámica necesitamos introducir primero el concepto de *Nodo*:

Un nodo es una celda en la que se almacenan dos cosas: un dato y un enlace a un nodo del mismo tipo que él. El enlace es como un "hilo" que une un nodo con el siguiente. De esta manera se arma una estructura con nodos enlazados de manera lineal. En la implementación dinámica de una pila se utiliza dicha estructura. El tope de la pila es un enlace al primer nodo y cada uno de ellos tiene un enlace al siguiente nodo (el que se encuentra por debajo de él en la pila), salvo el último que no tiene ningún nodo por debajo y este enlace es *null*. En la Figura 2.5 se muestra lo descripto.

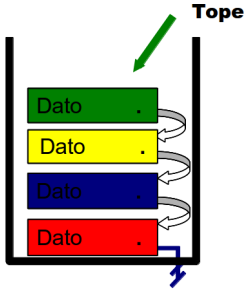


Figura 2.5: Representación gráfica de una pila implementada de manera dinámica

La clase Nodo

A continuación definiremos la clase Nodo para un tipo genérico de elemento. Luego, al implementarlo en Java lo haremos para un tipo especial de elemento como `int` o `String`. Es importante saber que el tipo de dato que se almacena en un nodo puede ser de cualquier tipo, incluidos TDAs definidos por usuarios (por ejemplo: Persona, Alumno, etc). En la Figura 2.6 se muestra la representación gráfica de un nodo que contiene el elemento "124". En el dibujo de la izquierda, el nodo tiene un enlace a otro nodo, mientras que a la derecha se muestra el nodo con un enlace nulo (se cruza la flecha con un par de rayitas, indicando que no está enlazado con otro nodo).

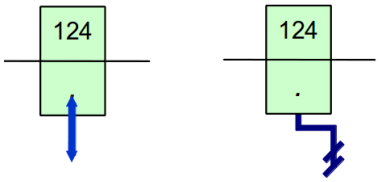


Figura 2.6: Representación gráfica de un nodo

Para poder operar con nodos debemos definir el TDA Nodo con sus datos y operaciones. En la Figura 2.7 se muestra el diagrama UML de clases para el TDA Nodo.

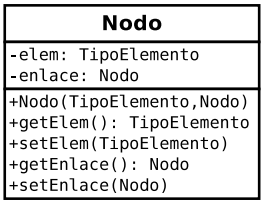


Figura 2.7: Diagrama de clases UML para Nodo (implementación dinámica de estructuras lineales)

A continuación se presenta la implementación de Nodo en Java, para el tipo de elemento `Object`[2].

Algoritmo 2.3 Código Java de clase Nodo

```
class Nodo {

    private Object elem;
    private Nodo enlace;

    // constructor
    public Nodo(Object elem, Nodo enlace) {
        this.elem = elem;
        this.enlace = enlace;
    }

    // modificadoras
    public void setElem(Object elem) {
        this.elem = elem;
    }

    public void setEnlace(Nodo enlace) {
        this.enlace = enlace;
    }

    // observadoras
    public Object getElem() {
        return elem;
    }

    public Nodo getEnlace() {
        return enlace;
    }

}
```

[2]Para comprender mejor cómo funciona Object como tipo genérico, lea la sección 1 del Anexo *"Cómo implementar estructuras para un tipo de elemento genérico"*

La clase Pila

Volviendo a la implementación dinámica de la estructura Pila, hemos dicho que una pila se forma por varios nodos enlazados entre sí, con el tope apuntando al primer nodo de la pila. Las inserciones y eliminaciones se harán siempre sobre el tope de la pila. El diagrama UML de clases para Pila con implementación dinámica se muestra en la Figura 2.8.

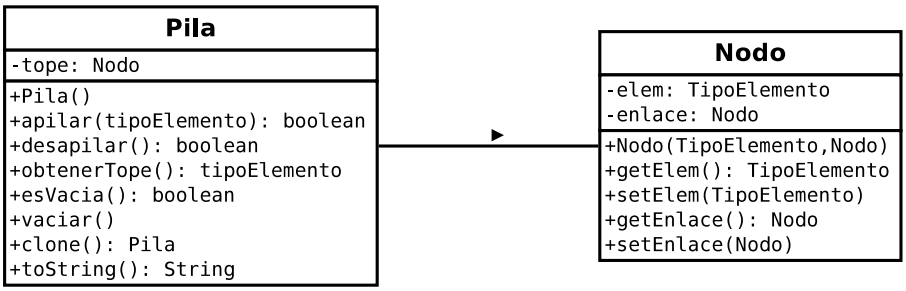


Figura 2.8: Diagrama de clases UML para pila (implementación dinámica)

A continuación se muestra un ejemplo en Java para el UML definido en la 2.8, utilizando Nodo como fue definido en el diagrama de clases UML de la Figura 2.7 e implementado según el Algoritmo 2.3. Como se observa en el código 2.4, el método *constructor* crea una instancia vacía de la Pila, para lo cual pone el tope en null (indicando que no apunta a ningún nodo).

Algoritmo 2.4 Código Java de clase Pila con método *constructor* (implementación dinámica)

```
public class Pila {

    private Nodo tope;

    public Pila() {
        this.tope = null;
    }
```

Por su lado, el método *apilar* crea una nueva instancia de Nodo, en la cual agrega el elemento nuevo, y lo enlaza al primer nodo de la pila (si la pila estaba vacía, el nodo nuevo queda apuntando a null). Luego, actualiza el atributo *tope* para que apunte al último nodo creado. Como la implementación dinámica no tiene restricción de espacio, la pila nunca se llena, por lo cual siempre devuelve true.

Algoritmo 2.5 Código Java método *apilar* de clase Pila (implementación dinámica)

```
public boolean apilar(Object nuevoElem){

    // crea un nuevo nodo delante de la antigua cabecera
    Nodo nuevo = new Nodo(nuevoElem, this.tope);

    // actualiza el tope para que apunte al nodo nuevo
    this.tope = nuevo;

    // nunca hay error de pila llena, entonces devuelve true
    return true;
}
```

A continuación se muestra el método *toString* de la clase Pila con implementación dinámica. Como se explicó en la definición de las operaciones del TDA Pila, el método toString se agrega para facilitar el trabajo durante la depuración o debugging de la clase Pila, y luego se lo comenta o se lo redefine de manera privada. Como se observa en el código (Algoritmo 2.6), primero se verifica que la estructura no sea vacía. Si lo es, se devuelve el texto "Pila vacía". Sino, se utiliza una variable auxiliar de tipo Nodo (*aux*), con lo cual se recorre la estructura desde el tope hacia el fondo de la pila. Para comenzar, se asigna a *aux* el valor del atributo *tope*, es decir que ahora *aux* y *tope* apuntan al mismo nodo. Luego, si *aux* no es nulo, la ejecución entra al ciclo mientras y agrega el valor del nodo a la cadena de salida. A continuación avanza al siguiente nodo con la asignación *aux* = *aux.getEnlace()*. Esta instrucción es básica para recorrer cualquier estructura dinámica lineal. En la Figura 2.9 se muestra paso a paso como cambia *aux* en una pila de 3 elementos.

Algoritmo 2.6 Código Java método *toString* de clase Pila (implementación dinámica)

```
@Override
public String toString(){
    String s = "";

    if (this.tope == null)
        s = "Pila vacia";
    else {
        // se ubica para recorrer la pila
        Nodo aux = this.tope;
        s = "[";

        while (aux != null){
            // agrega el texto del elem y avanza
            s += aux.getElem().toString();
            aux = aux.getEnlace();
            if (aux != null)
                s += ",";
        }
        s += "]";
    }

    return s;
}
```

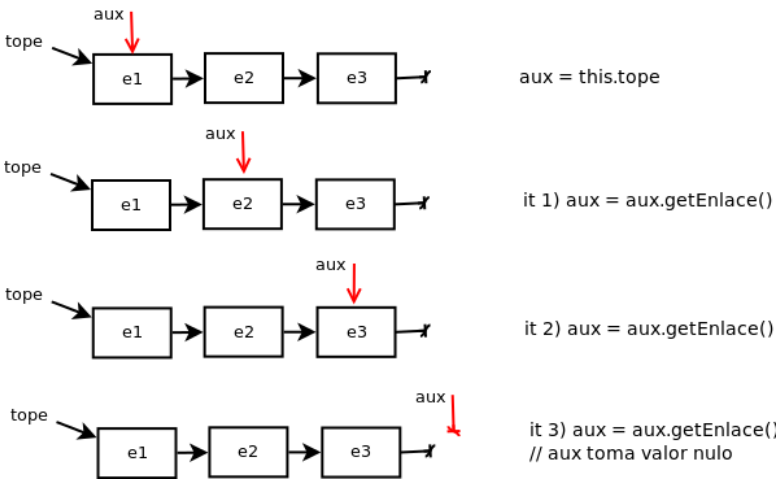


Figura 2.9: Cómo se avanza en una estructura lineal

Respecto a la implementación del método *vaciar*, en las primeras versiones de Java se debía eliminar nodo a nodo para liberar el espacio de memoria. En las versiones actuales no es necesario. Alcanza con setear el atributo *tope* en `null`, ya que el Garbage Collector libera la memoria que no se utiliza de manera recursiva. En caso de implementar pila dinámica en otro lenguaje, se debe analizar cómo trabaja dicho lenguaje para implementar la operación *vaciar* correctamente[3].

2.1.4. Análisis de eficiencia del TDA Pila

Comparando ambas implementaciones, vemos que la cantidad de pasos para obtener el tope de la pila es una operación de O(1) en ambos casos. De la misma manera con las operaciones *apilar* y *desapilar*. Por otro lado, las operaciones *clonar* y *toString* necesitan recorrer la pila completa en ambas implementaciones, visitando todos los nodos, por lo tanto son de O(n).

Respecto al espacio de memoria requerido, la implementación dinámica tiene la ventaja de que ocupa sólo lo que necesita y que este espacio puede crecer y decrecer según la necesidad del usuario. Otra ventaja es que la pila dinámica nunca se llena.

[3] Más información sobre el Garbage Collector de Java: *http://www.oracle.com/technetwork/java/javase/gc-tuning-6-140523.html*

Ejercicio 2.2: Implementación del TDA Pila de manera dinámica

1. Crear un paquete *lineales.dinamicas* e implementar las clases Pila y Nodo, con todas las operaciones especificadas para el TDA Pila.
2. Utilice la clase TestPila del paquete *tests.lineales* anterior (Ejercicio 2.1) para probar que todas las operaciones de la clase Pila funcionan igual que antes (debe lograrlo cambiando solamente el nombre del paquete importado).
3. En el paquete *tests.lineales* agregue clases para probar las operaciones de la clase Pila con elementos del TDA Alumno definido en el Apunte 1 (Introducción y Repaso).

2.2. Colas

2.2.1. Definición

Una cola es una estructura de datos en la que el primer elemento en entrar es el primero en salir. Se denominan estructuras FIFO (First In, First Out). Se tiene acceso al frente de la cola para ver el primer elemento o eliminarlo, y al final para agregar nuevos elementos. El frente de la cola corresponde al elemento que entró primero, es decir el que saldrá en la próxima eliminación. El final de la cola corresponde al elemento que entró en último lugar. La Figura 2.10 muestra esta estructura de forma gráfica.

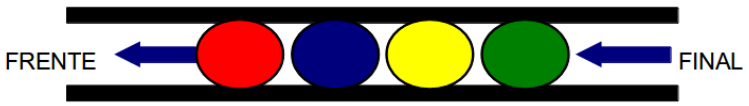


Figura 2.10: Representación gráfica de una cola

Un ejemplo común de cola son las colas de espera de un banco o una caja de supermercado. En sistemas operativos, se utilizan para planificar el uso de los recursos de la computadora, por ejemplo, la cola de trabajos pendientes de impresoras: la petición se pone al final de la cola; cuando la petición llega al frente de la cola, el proceso o trabajo solicitado, se procesa.

2.2.2. Operaciones

Las operaciones básicas del TDA Cola son:

- *constructor vacío*
  // Crea y devuelve una cola vacía
- *poner* (nuevoElem):boolean
  // Pone el elemento al *final* de la cola. Devuelve *verdadero* si el elemento se pudo agregar en la estructura y *falso* en caso contrario.
- *sacar*():boolean
  // Saca el elemento que está en el *frente* de la cola. Devuelve *verdadero* si el elemento se pudo sacar (la estructura no estaba vacía) y *falso* en caso contrario.
- *obtenerFrente*():tipoElemento
  // Devuelve el elemento que está en el *frente*. *Precondición*: la cola no está vacía.
- *esVacia*(): boolean
  // Devuelve *verdadero* si la cola no tiene elementos y *falso* en caso contrario
- *vaciar*():void
  // Saca todos los elementos de la estructura.
- *clone*():Cola
  // Devuelve una copia exacta de los datos en la estructura original, y respetando el orden de los mismos, en otra estructura del mismo tipo

- *toString*() : String
  // Crea y devuelve una cadena de caracteres formada por todos los elementos de la cola para poder mostrarla por pantalla. *Es recomendable utilizar este método únicamente en la etapa de prueba y luego comentar el código.*

2.2.3. Implementaciones

Al igual que en las pilas, se implementarán las colas en forma estática y dinámica, con la diferencia que en la implementación estática de colas, el arreglo debe ser considerado de manera circular, así se evita el corrimiento de los datos en el arreglo, lo cual sería sumamente ineficiente.

2.2.3.1. Implementación estática

Para la implementación estática se supondrá que el arreglo es circular, es decir que sus dos extremos (frente y final) se unen tal como se muestra en la Figura 2.11. Al agregar o quitar elementos de la cola, el frente o el final de la cola se actualizan a una nueva posición del arreglo de la que poseen hasta el momento, sin cambiar de lugar los elementos.

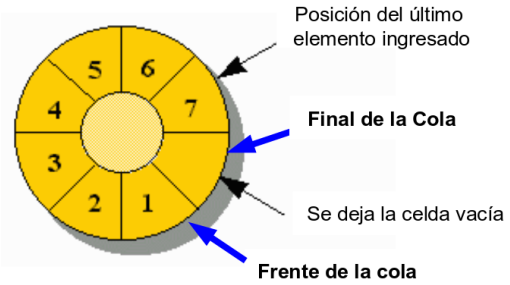


Figura 2.11: Ejemplo de Cola implementada con un arreglo circular

En esta implementación la cola es un arreglo definido de un tamaño fijo. El valor máximo para el tamaño de la cola se almacena como una constante (TAMANIO). El *frente* y el *final* son variables enteras que almacenan la posición del primer y último elemento en el arreglo respectivamente. La cola crece y decrece dentro del espacio disponible (definido por la longitud del arreglo), pero de manera circular. Esto quiere decir que en lugar de mover los elementos, para que el primero se ubique en una posición en particular, lo que se mueve son los "cursores" final y frente. La cola comienza a llenarse desde la posición 0 (cero), avanzando el cursor *final* cuando se agrega, y avanzando el cursor *frente* cuando un elemento se saca de la cola. Cuando uno de los dos debe pasar de la posición TAMANIO-1 a la siguiente (que sería TAMANIO) en realidad pasa a la posición 0.

Para evitar confundir *cola llena* y *cola vacía*, la variable *final* siempre apunta a la posición siguiente a la que contiene el último elemento. Es decir que se permiten cargar como máximo TAMANIO-1 elementos. Cuando la cola está vacía se verifica *frente* = *final*. Si la cola se llena, como en el caso de la Figura 2.12, *final+1* = *frente*.

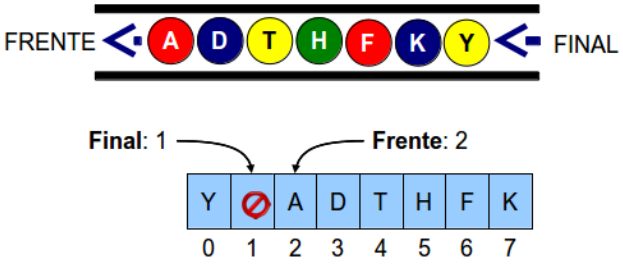


Figura 2.12: Ejemplo de cola circular llena

En la Figura 2.13 se muestra el diagrama de clases en notación UML para una Cola implementada estáticamente con arreglo circular.

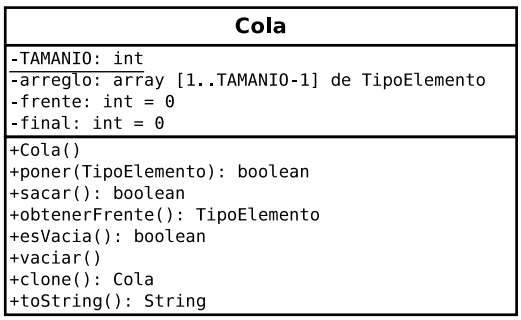


Figura 2.13: Diagrama de clases UML para Cola (implementación estática)

En el Algoritmo 2.7 se muestra el código de declaración de la clase Cola implementada estáticamente con arreglo circular. Fíjese que el atributo se ha llamado *fin* (en lugar de *final*, dado que *final* es una palabra reservada del lenguaje Java). Luego en el Algoritmo 2.8 se muestra como ejemplo la implementación del método *sacar* de la misma clase.

Algoritmo 2.7 Código Java del constructor de la clase Cola (implementación estática)

```
public class Cola {

    private Object[] arreglo;
    private int frente;
    private int fin;
    private static final int TAMANIO = 10;

    public Cola() {
        this.arreglo = new Object[this.TAMANIO];
        this.frente = 0;
        this.fin = 0;
    }
```

Algoritmo 2.8 Código Java del método *sacar* de clase Cola (implementación estática)

```
public boolean sacar() {
    boolean exito = true;

    if (this.esVacia()) // la cola esta vacia, reporta error
    {
        exito = false;
    } else // al menos hay 1 elemento: avanza frente (de manera circular)
    {
        this.arreglo[this.frente] = null;
        this.frente = (this.frente + 1) % this.TAMANIO;
    }

    return exito;
}
```

Ejercicio 2.3: Implementación del TDA Cola de manera estática

1. En el paquete *lineales.estaticas* implementar la clase Cola para elementos genéricos de tipo Object, incluyendo todas las operaciones del TDA Cola.

2. En el paquete *tests.lineales* implementar la clase TestCola que permita probar todas las operaciones de la clase Cola anterior con elementos de algún tipo primitivo y de TDA Fecha definido en el Apunte 1 (Introducción y Repaso).

Al igual que en TDA Pila, la ventaja de implementar una cola en forma estática, usando un arreglo, es que se accede inmediatamente al frente y final; y la desventaja es que el arreglo tiene una capacidad máxima fija, teniendo que reservar espacio de memoria que puede quedar sin ser utilizado.

2.2.3.2. Implementación dinámica

La implementación dinámica de cola se realiza con nodos enlazados entre sí. A diferencia de Pila que guardaba sólo el enlace al tope, el objeto Cola guarda el enlace al frente y al final de la cola. Cada nodo de la Cola tiene referencia al nodo siguiente. La Figura 2.14 muestra gráficamente la estructura dinámica.

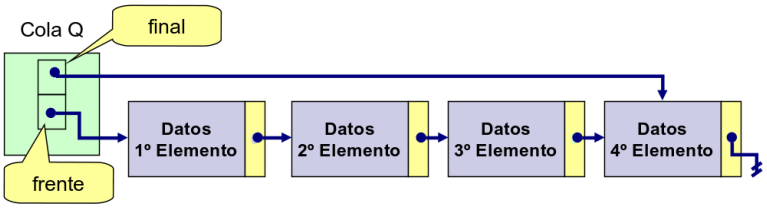


Figura 2.14: Ejemplo de Cola implementada de manera dinámica

En la implementación con nodos enlazados se debe verificar y considerar dos casos especiales, *poner* un elemento en una cola vacía, y eliminar el último elemento de la cola (después de ejecutar la operación *sacar*, la cola debe quedar vacía).

En la Figura 2.15 se muestra el algoritmo de la operación *poner* de manera gráfica. Los pasos son:

1. Crear un nuevo nodo (nuevoNodo) con el dato a insertar.
2. Enlazar el nodo creado a continuación del nodo final.
3. Hacer que *final* apunte a nuevoNodo (es decir, se enlaza final con el nodo que se acaba de insertar).
4. En caso de que la cola esté vacía, el nuevo nodo deberá elanzarse tanto al frente como al final de la cola.

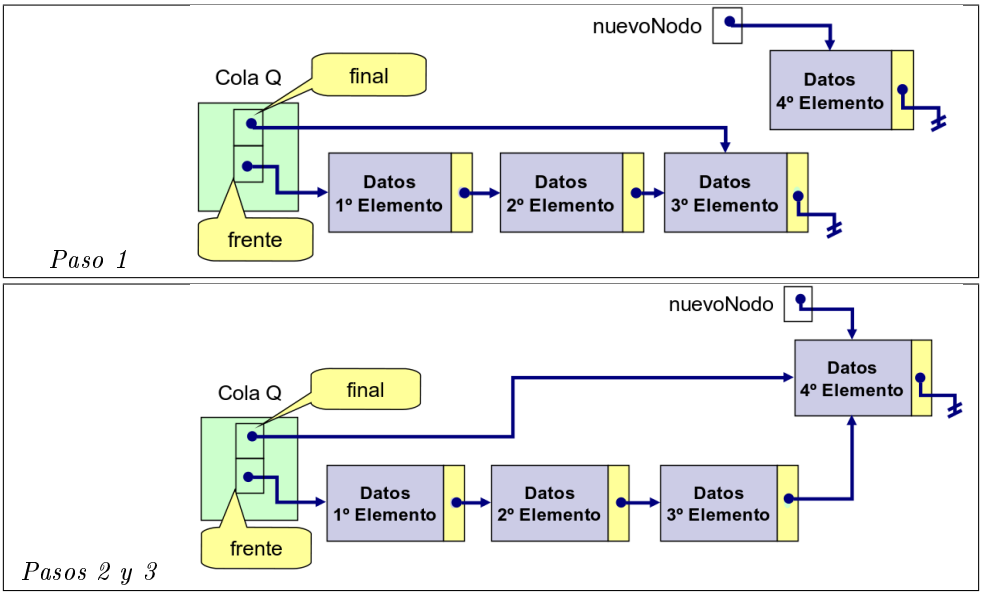


Figura 2.15: Poner un elemento en Cola (implementación dinámica)

Otra operación básica del TDA es *sacar*, que quita el elemento del frente de la cola. En la Figura 2.16 se observa que la operación debe hacer que *frente* avance una posición, apuntando al siguiente nodo. Un caso especial es cuando el elemento a quitar es el último que queda en la cola, debiendo cuidarse de dejar en `null` ambos punteros (*frente* y *final*). Otro caso especial es cuando se intenta sacar un elemento de una cola que se encuentra vacía, en cuyo caso el método debe devolver `false` para indicar el error.

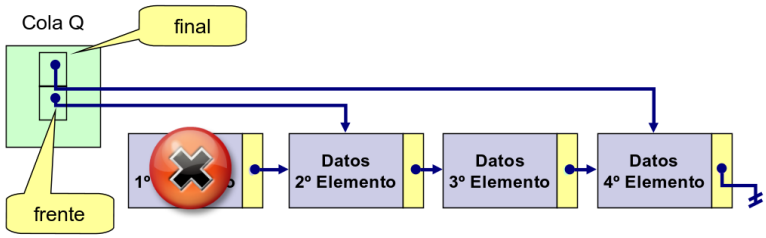


Figura 2.16: Sacar un elemento en cola (implementación dinámica)

En la Figura 2.17 se muestra el diagrama de clases UML para una Cola implementada con nodos enlazados, utilizando la clase Nodo usada anteriormente en la implementación de Pila.

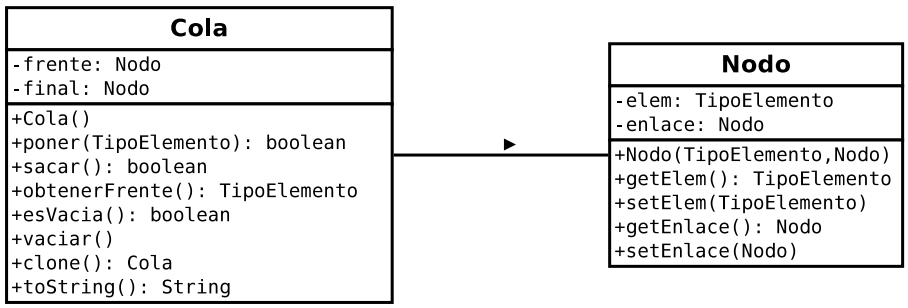


Figura 2.17: Diagrama de clases UML para Cola (implementación dinámica)

A continuación se presenta el código Java de la clase Cola, implementada de acuerdo al diagrama UML presentado en la Figura 2.17.

Algoritmo 2.9 Código Java del método *sacar* de clase Cola (implementación dinámica)

```
public boolean sacar(){
    boolean exito = true;

    if (this.frente == null)
        // la cola esta vacia, reporta error
        exito = false;
    else{
        // al menos hay un elemento:
        // quita el primer elemento y actualiza frente (y fin si queda vacia)
        this.frente = this.frente.getEnlace();
        if (this.frente == null)
            this.fin = null;
    }

    return exito;
}
```

2.2.4. Análisis de eficiencia del TDA Cola

Al igual que en las implementaciones del TDA Pila, en las del TDA Cola se puede observar que la cantidad de pasos para obtener el frente de la cola o para poner en cola es constante, es decir que poseen O(1), tanto en el caso estático como dinámico. Por otro lado, las operaciones clonar y toString necesitan recorrer la cola completa en ambas implementaciones, visitando todos los nodos, por lo tanto son de O(n).

Respecto al espacio de memoria requerido, la implementación dinámica tiene la ventaja de que ocupa sólo el espacio de memoria necesario para almacenar los elementos, con la posibilidad de crecer y decrecer según la necesidad del usuario, y sin el inconveniente de que la estructura pueda quedarse sin espacio.

Ejercicio 2.4: Implementación del TDA Cola de manera dinámica

1. En el paquete *lineales.dinamicas* agregar la clase Cola, incluyendo todas las operaciones del TDA Cola.
2. Utilice la clase TestCola del paquete *tests.lineales* anterior para probar todas las operaciones de la nueva clase Cola (debe lograrlo simplemente cambiando el nombre del paquete importado)
3. En una nueva clase MixLineales en el paquete tests.lineales, implementar el método: generarOtraCola(Cola c1) que recibe por parámetro una estructura de tipo Cola c1 con elementos de tipo char que tiene el siguiente formato: $a_1\$a_2\$a_3\$\ldots\$a_n$, donde cada a_i es una sucesión de letras mayúsculas y a partir de c1 debe generar como salida otra Cola de la forma: $a_1a_1^*\$a_2a_2^*\$\ldots\$a_na_n^*$ donde cada a_i^* es la secuencia de letras de a_i invertida. Ejemplo: Si c1 es : ←A,B,$,C,$,D,E,F←, la operación generarOtraCola devolverá una Cola con el siguiente formato: ←A,B,B,A,$,C,C,$,D,E,F,F,E,D←
NOTA: Para lograr los tramos invertidos de la Cola de salida debe utilizar una estructura Pila auxiliar. La clase MisLineales debe funcionar igual si se compila importando las estructuras Pila y Cola de los paquetes lineales.dinamicas o lineales.estaticas, indistintamente.
4. Diseñe un lote de prueba efectivo para comprobar que el programa del inciso 3 es correcto.
5. Testee el programa usando el lote de prueba diseñado y verifique el funcionamiento tanto con las implementaciones dinámicas como estáticas de Pila y Cola.

2.3. Listas

2.3.1. Descripción

La lista es una secuencia de elementos, que constituye una estructura flexible, que puede crecer o acortarse según sea necesario. Los elementos de una lista se pueden insertar, acceder, borrar y consultar por posiciones, siendo 1 la primer posición. La Figura 2.18 muestra gráficamente la estructura de una lista.

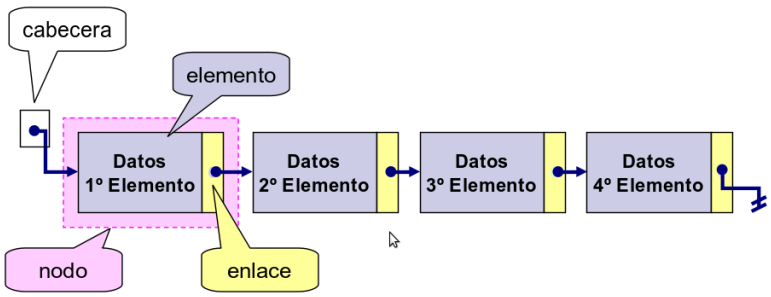


Figura 2.18: Representación gráfica de una lista

2.3.2. Operaciones

Las operaciones básicas para definir el TDA Lista son las siguientes:

- *constructor vacío*
  // Crea y devuelve una lista vacía.
- *insertar* (elem, pos) : boolean
  // Agrega el elemento pasado por parámetro en la posición *pos*, de manera que la cantidad de elementos de la lista se incrementa en 1. Para una inserción exitosa, la posición recibida debe ser $1 \leq pos \leq longitud(lista) + 1$. Devuelve *verdadero* si se puede insertar correctamente y *falso* en caso contrario.
- *eliminar* (pos) : boolean
  // Borra el elemento de la posición *pos*, por lo que la cantidad de elementos de la lista disminuye en uno. Para una eliminación exitosa, la lista no debe estar vacía y la posición recibida debe ser $1 \leq pos \leq longitud(lista)$. Devuelve *verdadero* si se pudo eliminar correctamente y *falso* en caso contrario.

- *recuperar* (pos) : elemento
  // Devuelve el elemento de la posición *pos*. La precondición es que la posición sea válida.
- *localizar* (elem) : int
  // Devuelve la posición en la que se encuentra la primera ocurrencia de *elem* dentro de la lista. En caso de no encontrarlo devuelve -1.
- *vaciar*() : void
  // Quita todos los elementos de la lista. *El manejo de memoria es similar al explicado anteriormente para Cola y Pila dinámicas.*
- *esVacia*() : boolean
  // Devuelve *verdadero* si la lista no tiene elementos y *falso* en caso contrario.
- *clone*() : lista
  // Devuelve una copia exacta de los datos en la estructura original, y respetando el orden de los mismos, en otra estructura del mismo tipo
- *longitud*() : int
  // Devuelve la cantidad de elementos de la lista.
- *toString*(): String
  // Crea y devuelve una cadena de caracteres formada por todos los elementos de la lista para poder mostrarla por pantalla. *Es recomendable utilizar este método únicamente en la etapa de prueba y luego comentar el código.*

A continuación se analizan dos operaciones básicas del TDA Lista:

- *Recuperar* es la operación análoga a obtenerTope en Pila u obtenerFrente en Cola, con la diferencia que la estructura lista no tiene un orden predeterminado de atención, por lo que se puede acceder a cualquiera de sus elementos por medio de su posición. Por ejemplo, en la Figura 2.19 recuperar(3) devuelve el elemento en la tercera posicón, es decir 500.
- *Localizar* devuelve la posición de un elemento en la lista. Si no lo encuentra, devuelve -1. Si el elemento se encuentra más de una vez en la estructura, se devuelve la posición de la primer aparición del mismo comenzando desde la posición 1. Por ejemplo, en la Figura 2.19 localizar(390) devuelve la posición 2.

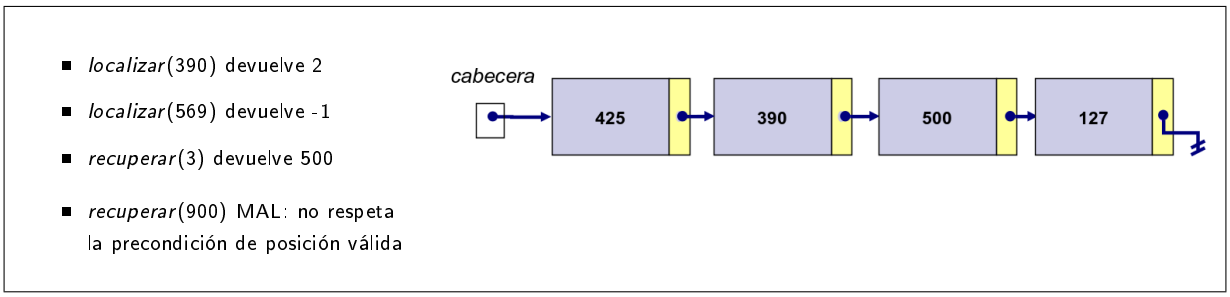


Figura 2.19: Ejemplos de las operaciones *localizar* y *recuperar* de TDA Lista

Otra operación importante del TDA Lista es *longitud*(), que devuelve la cantidad de elementos que posee la lista. Por ejemplo, esta operación es necesaria para saber si la posición en la que se desea insertar o eliminar un elemento es válida o no. Para implementarlo, se debe recorrer desde la cabecera hasta el último nodo, contando cada uno de ellos.

2.3.3. Implementación

En este apunte se desarrollará la implementación dinámica de la estructura lista, ya que es la representación natural según el propósito del TDA Lista. Al igual que en la implementación dinámica de pilas y colas, los nodos tienen un lugar donde guardan datos y un enlace al próximo nodo.

En la Figura 2.20 se presenta el diagrama de clases UML propuesto para la implementación dinámica de listas.

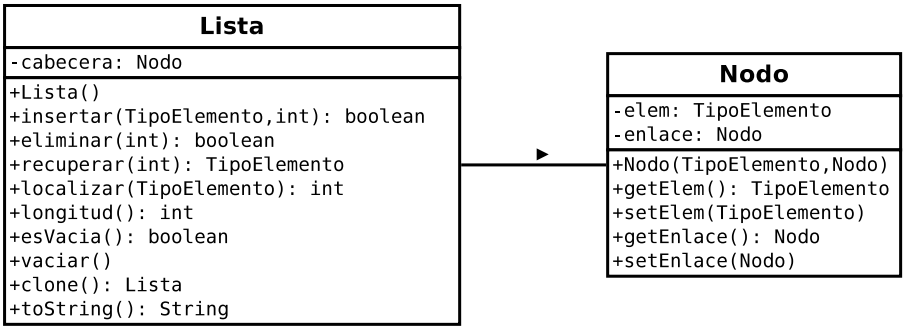


Figura 2.20: Diagrama de clases UML para Lista (implementación dinámica)

Para la mayoría de las operaciones del TDA Lista en esta implementación (por ejemplo, para localizar un elemento) siempre se comienza accediendo desde la cabecera al primer elemento, luego moviéndose al segundo, y así sucesivamente. La manera de avanzar de un nodo al siguiente es utilizando un puntero auxiliar de tipo Nodo, que se inicializa apuntando a la *cabecera*, y se avanza nodo a nodo mediante la asignación *aux=aux.getEnlace()*, como fue explicado para la operación *toString* de Pila en la Figura 2.9.

En la implementación de las operaciones *insertar* y *eliminar* del TDA Lista se considera como caso especial el tratamiento de la primer posición. A continuación, la Figura 2.21 muestra gráficamente los pasos necesarios para insertar un nodo en la primer posición de la lista.

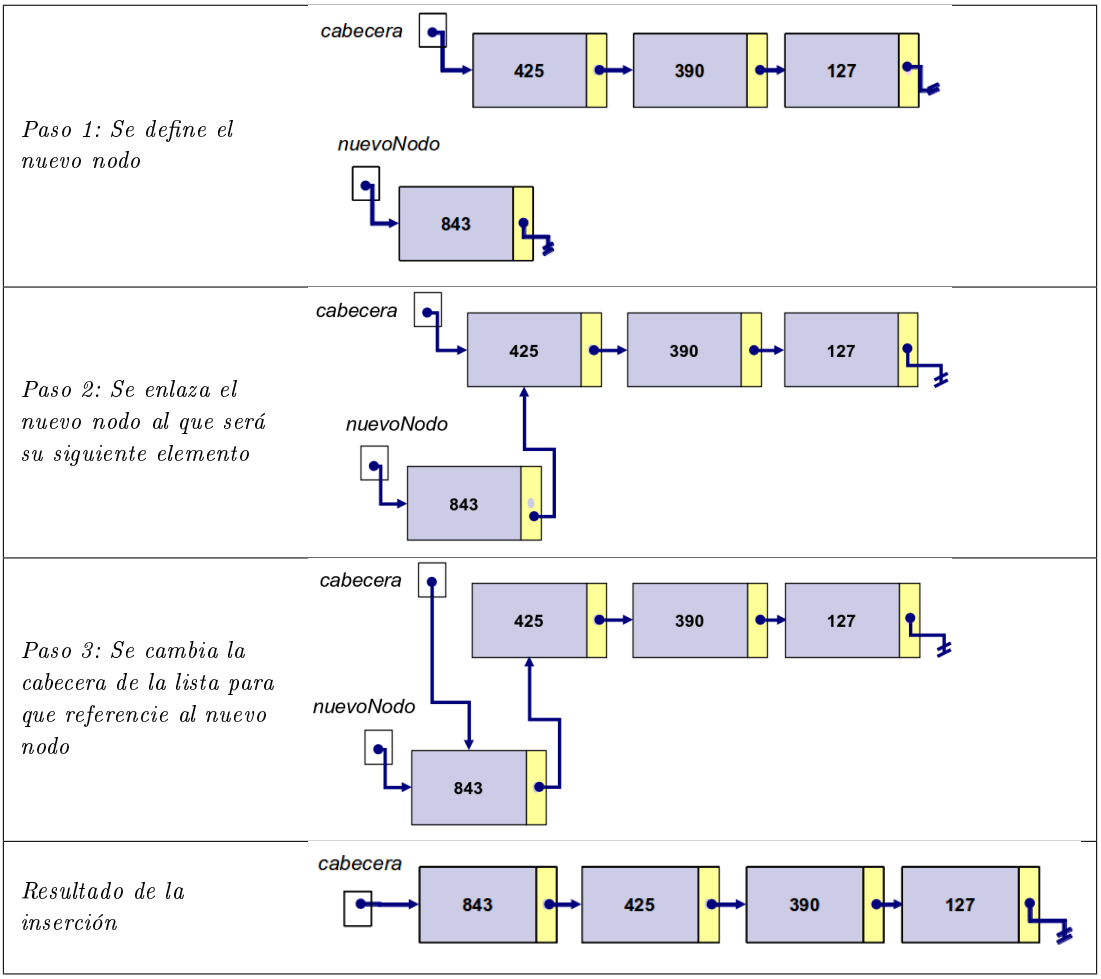


Figura 2.21: Insertar en Lista en la primera posición

Tener en cuenta que *nuevoNodo* es una variable local al método insertar, por lo que se libera al finalizar dicho método.

En la Figura 2.22 se muestra paso a paso el proceso de inserción de un elemento en la posición 3 de la lista. Cuando se inserta un elemento en una posición diferente a 1 se utliza una variable auxiliar (temp) para obtener la referencia al nodo que se ubicará en la posición anterior al nodo nuevo.

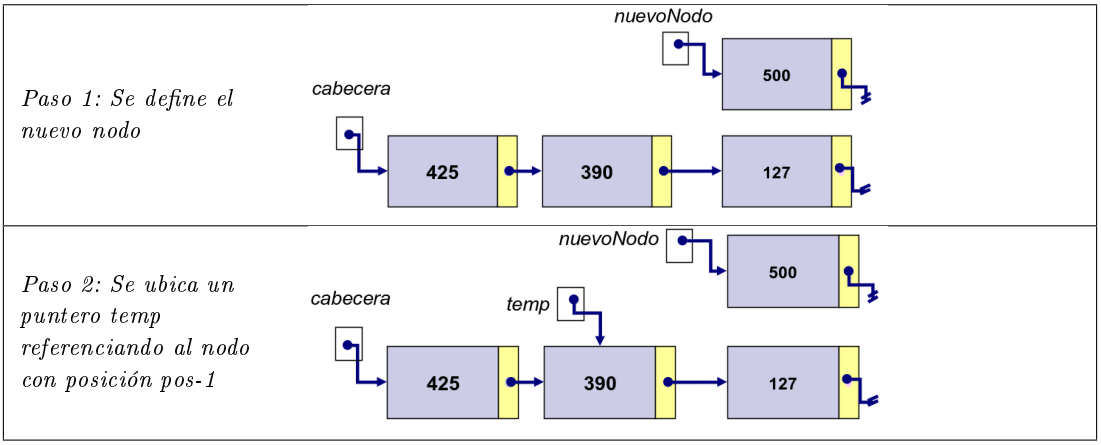


Figura 2.22: Insertar en Lista en una posición distinta de 1

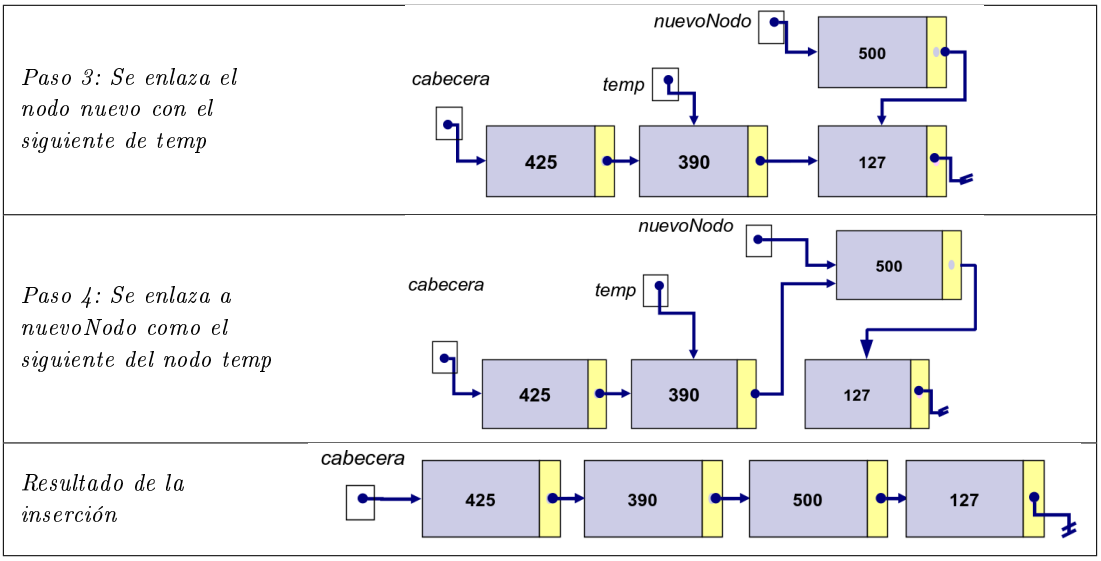


Figura 2.23: Insertar en Lista en una posición distinta de 1 (cont.)

A continuación se presenta el código del método *insertar* de la clase Lista, que implementa el TDA Lista para elementos de tipo Object, de acuerdo al diagrama UML presentado en la Figura 2.20.

Algoritmo 2.10 Código Java método *insertar* de clase Lista

```
public boolean insertar(Object nuevoElem, int pos) {
    // inserta el elemento nuevo en la posicion pos
    // detecta y reporta error posicion invalida
    boolean exito = true;

    if (pos < 1 || pos > this.longitud() + 1) {
        exito = false;
    } else {
        if (pos == 1) // crea un nuevo nodo y se enlaza en la cabecera
        {
            this.cabecera = new Nodo(nuevoElem, this.cabecera);
        } else {   // avanza hasta el elemento en posicion pos-1
            Nodo aux = this.cabecera;
            int i = 1;
            while (i < pos - 1) {
                aux = aux.getEnlace();
                i++;
            }
            // crea el nodo y lo enlaza
            Nodo nuevo = new Nodo(nuevoElem, aux.getEnlace());
            aux.setEnlace(nuevo);
        }
    }
    // nunca hay error de lista llena, entonces devuelve true
    return exito;
}
```

Como se dijo anteriormente, para eliminar un elemento también se debe diferenciar el caso en que este sea el primer elemento de la lista o no. La Figura 2.24 muestra paso a paso este caso.

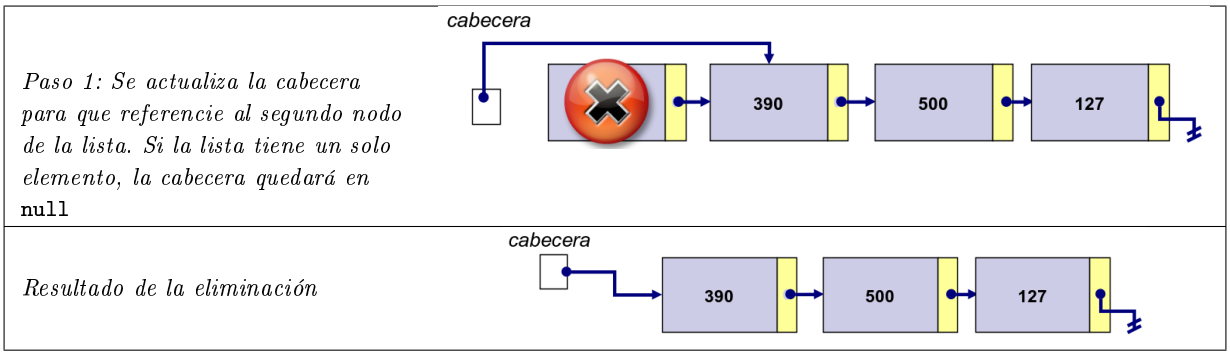


Figura 2.24: Eliminar en Lista en la primera posición

A continuación, la Figura 2.25 muestra la eliminación de un elemento en la posicion 3 (posición distinta de 1).

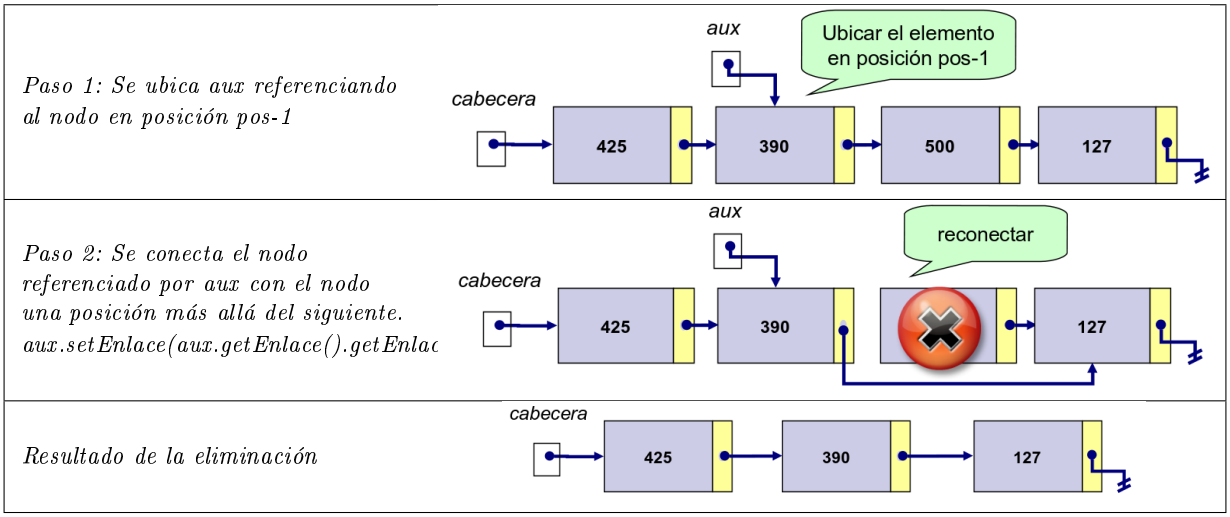


Figura 2.25: Eliminar en Lista en posición distinta de 1

Ejercicio 2.5: Implementación y uso del TDA Lista

Ejercicios obligatorios:

1. Agregar la clase Lista en el paquete *lineales.dinamicas* e implementar todas las operaciones de TDA Lista para elementos de tipo `Object`.

2. Agregar la clase TestLista en el paquete *test.lineales* que pruebe todas las operaciones de la clase Lista.

3. Agregar una clase PruebaLista al paquete *test.lineales* con los siguientes métodos y opciones de menú para probarlos:

   a) *concatenar*: recibe dos listas L1 y L2 y devuelve una lista nueva con los elementos de L1 y L2 concatenados. Ej: si L1=[2,4,6] y L2=[5,1,6,7] debe devolver [2,4,6,5,1,6,7]

   b) *comprobar*: recibe una lista L1 cargada con dígitos (números enteros de 0 a 9) y verifica si los elementos que contiene tienen la forma cadena0cadena0cadena* (donde cadena* es cadena invertida). Ej: si L1=[9,6,5,0,9,6,5,0,5,6,9], cadena=965, luego cadena*=569, entonces la lista L1 cumple con la condición deseada.
   Atención: la longitud de cada cadena no se conoce de antemano, hay que identificarla por la primera posición de 0 en la lista.
   Nota: Utilizar una Pila y una Cola como estructuras auxiliares.

   c) *invertir*: recibe una lista L y devuelve una lista nueva con los elementos de L invertidos. Ej: si L1=[2,4,6] debe devolver [6,4,2]

4. Extienda el TDA Lista agregando las siguientes operaciones a la clase Lista correspondiente, cuidando la eficiencia:

   a) *invertir*: modifica la lista original para que los elementos aparezcan en orden invertido, haciendo un único recorrido de la estructura y sin usar estructuras auxiliares ni otras operaciones del TDA. Ej: si L=[1,2,3,4] debe devolverla modificada como [4,3,2,1]

   b) *eliminarApariciones*(TipoElemento x): elimina todas las apariciones de elementos iguales a x, haciendo un único recorrido de la estructura y sin usar otras operaciones del TDA. Ej: si L1=[1,2,1,3,4] debe devolver [2,3,4]. En los casos de prueba considere que el elemento a eliminar puede estar repetido varias veces en cualquier posición.

5. Analice el orden de eficiencia O de los métodos de los ejercicios 3 y 4

Ejercicios adicionales:

1. En la clase PruebaLista anterior agregar los siguientes métodos y más opciones al menú para probarlos:

   a) *intercalar*: que dadas dos listas L1 y L2, devuelva una lista nueva con los elementos de L1 y L2 intercalados. Por ejemplo, si L1=[1,3,5] y L2=[2,4,6,7] debe devolver [1,2,3,4,5,6,7]

   b) *contar*: que dada una lista que admite elementos repetidos, cuente cuántas veces aparece un *elemento* dado dentro de la lista.

      1) En forma iterativa.
      2) En forma recursiva.

   c) *esCapicúa*(): que verifica si los elementos en una lista son capicúa.

2. Analice cómo extender la clase Lista para hacer más eficientes las operaciones del ejercicio anterior y evalúe cómo cambia la eficiencia cuando se implementan dentro del TDA.

3. En la clase MixLineales en del paquete test.lineales, agregar el método: generarLista(Lista lis) que recibe por parámetro una estructura de tipo Lista lis con elementos de tipo char que tiene el siguiente formato: $a_1 * a_2 * a_3 * \ldots * a_n$, donde cada a_i en una sucesión de letras mayúsculas y a partir de lis debe generar como salida otra Lista de la forma: $a_1 a_1$"$a_1 * a_2 a_2$"$a_2 * \ldots . * a_n a_n$"a_n donde cada a_i" es la secuencia de letras de a_i invertida. Ejemplo: Si lis es AB*C*DEF, la operación generarOtraLista devolverá una Lista con el siguiente formato: ABBAAB*CCC*DEFFEDDEF
   NOTA: Para lograr los tramos invertidos de la Lista de salida debe utilizar una Pila auxiliar y para lograr los tramos no invertidos debe utilizar una Cola auxiliar.

2.3.4. Análisis de la eficiencia

Como se ha explicado en las secciones anteriores, casi todas las operaciones de Lista (excepto *vaciar* o *esVacia*) necesitan, en el peor de los casos, recorrer la lista por completo (por ejemplo, para la operación *insertar* el peor caso es que la posición sea *longitud()+1*), por lo que el orden de las operaciones es de O(n).

Una mejora que merece la pena en la implementación dinámica es almacenar en la clase Lista un atributo entero que guarde la cantidad de elementos existentes en la estructura, siendo cero la longitud de una lista vacía. Esta modificación permitiría saber la longitud de la lista en O(1) en lugar de O(n).

Otras mejoras conocidas de implementaciones de Lista son:

- Agregar a todos los nodos un enlace al nodo anterior. De esta manera la lista queda doblemente enlazada, siendo posible desplazarse al nodo anterior y al siguiente en O(1).
- La mejora anterior suele combinarse con el agregado en la clase Lista de un enlace al último elemento (similar a la implementación dinámica de Cola). De esta manera, insertar en la última posición de la lista, que es de O(n) en la implementación simple, se convierte en una operación de O(1) al igual que insertar en la cabecera.
- Aprovechando todas las mejoras mencionadas, la eficiencia de la implementación del método insertar, eliminar y recuperar se pueden bajar de O(n) a la mitad, calculando si la posición deseada está más cerca de la cabecera o del final, y moviéndose por los enlaces al siguiente o al anterior según corresponda, minimizando los recorridos. En esta implementación, el peor de los casos es la posición de la mitad, que es el máximo de saltos posibles.

Las ventajas de eficiencia en ejecución que proporciona la implementación de lista doblemente enlazada y con doble cabecera se ve opacada por la dificultad al programar y testear dicha implementación, sin embargo vale la pena el esfuerzo si mantener datos en listas largas es crucial para resolver el problema.