INDICE GENERAL

INDICE DE FIGURAS

INDICE DE TABLAS

INTRODUCCIÓN

Como ya es de conocimiento de todos, las computadoras fueron diseñadas o ideadas como una herramienta mediante la cual podemos realizar operaciones de cálculo complicadas en un lapso de mínimo tiempo. Pero la mayoría de las aplicaciones de este fantástico invento del hombre, son las de almacenamiento y acceso de grandes cantidades de información.

La información que se procesa en la computadora es un conjunto de datos, que pueden ser simples o estructurados. Los datos simples son aquellos que ocupan sólo un localidad de memoria, mientras que los estructurados son un conjunto de casillas de memoria a las cuales hacemos referencia mediante un identificador único.

Por lo general se tiene que tratar con conjuntos de datos y no con datos simples (enteros, reales, booleanos, etc.) que por sí solos no dicen nada, ni sirven de mucho, es necesario tratar con estructuras de datos adecuadas a cada necesidad.

Las estructuras de datos son una colección de datos cuya organización se caracteriza por las funciones de acceso que se usan para almacenar y acceder a elementos individuales de datos.

Ejemplo. Suponiendo que se enfrentan a un problema como este: Una empresa que cuenta con 150 empleados, desea establecer una estadística sobre los salarios de sus empleados, y quiere saber cuál es el salario promedio, y también cuantos de sus empleados gana entre $1250.00 y $2500.00.

Si se toma la decisión de tratar este tipo de problemas con datos simples, pronto se percatar del enorme desperdicio de tiempo, almacenamiento y velocidad. Es por eso que para situaciones de este tipo la mejor solución son los datos estructurados.

Existen en la actualidad dos tipos de estructuras de datos las cuales son:

Estructura de Datos estáticas: Son aquellas en las que el espacio ocupado en memoria se define en tiempo de compilación y no puede ser modificado durante la ejecución del programa. Corresponden a este tipo los vectores (arrays) y registros (struct).

Estructuras de Datos Dinámicas: Son aquellas en las que el espacio ocupado en memoria puede ser modificado en tiempo de ejecución. Corresponden a este tipo las listas, árboles y grafos. Estas estructuras no son soportadas en todos los lenguajes. La elección de la estructura de datos idónea dependerá

de la naturaleza del problema a resolver y, en menor medida, del lenguaje. Las estructuras de datos tienen en común que un identificador, nombre, puede representar a múltiples datos individuales.

Vectores (Array)

Es una colección o conjunto de datos o elementos que tienen el mismo tipo de dato. Estos vectores tienen una dimensión definida en tiempo de compilación la cual no será posible modificar en tiempo de ejecución.

Se almacenan en posiciones consecutivas de memoria y reciben un nombre común. Para referirse a un determinado elemento de un vector se debe utilizar un índice, que especifique su posición relativa en el vector.

Los tipos de datos que puede soportar un vector son los tipos de datos primitivos (short, int, float, double, ect) y los tipos de datos definidos por el usuario (enum, struct y class).

Existen diferentes maneras de definir un vector en las cuales se enumeran las mismas

1. Declarar un vector con un tamaño definido

```
tipo_dato nombre_vector[tamaño_vector];
```

El tamaño del vector puede definirse mediante constantes, variables o establecer un número;

Ejemplo. Definir un vector con valores reales de 100 datos

- Definido mediante constante

  ```
  #define TAM 100
  float vector[TAM];
  ```

- Definido mediante variable

  ```
  int tam = 100;
  float vector[tam];
  ```

- Definido mediante valor numérico

  ```
  float vector[100];
  ```

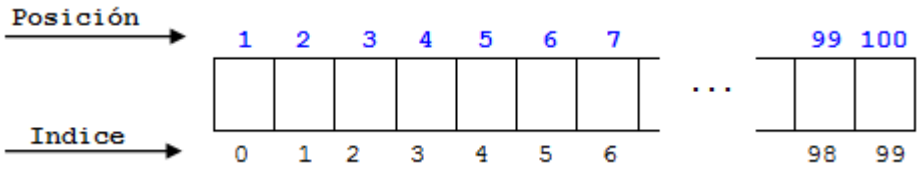


Figura 1. Representación de un vector

2. Declarar un vector con valores constantes

No se define el tamaño del vector pero si se inicializan valores al vector definiéndose con estos valores el tamaño del vector.

Ejemplo. Definir un vector con los siguientes valores (1, 5, 6, 3, 2 y 9)

```
int vector[] = {1, 5, 6, 3, 2, 9}
```

Del ejemplo anterior se observar que el vector tiene un dimensión de 6

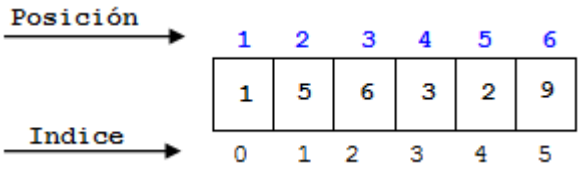


Figura 2. Representación de un vector con valores definidos

Para asignar valores a las casillas de un vector siempre debe estar al lado izquierdo del signo = (igual). El acceso a los datos de un vector se realiza mediante el índice, donde el primer elemento del vector se encuentra en el índice 0 como se muestra en la figura 2.

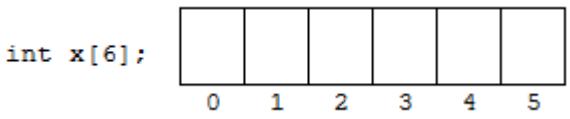


Figura 3. Declaración y creación de un vector

```
x[0] = 10;
x[1] = 5;
x[2] = 30;
x[3] = 2;
```

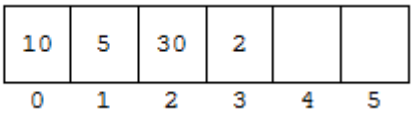


Figura 4. Asignación de valores a un vector

En el caso de que se quiere recuperar los valores de un vector también el acceso se realiza mediante el índice, en este caso el vector con su índice debe estar a la derecha del = (igual). Además las variables a las que se asignará el valor del vector tienen que tener el mismo tipo de dato que el vector. Por tanto como el vector tiene un tipo de dato entero (int) la variable también será entero (int).

```
int a = x[0];
int b = x[1];
int c = x[2];
int d = x[3];
```

Los valores de estas variables son:
a = 10
b = 5
c = 30
d = 2

A la vez se puede asignar y recuperar valores a un vector, significa que los índices de los vectores estarán en ambos lados del = (igual).

```
x[4] = x[0] + x[1];
x[5] = x[2] / x[3];
x[1] = x[4] * 2;
x[0] = x[5] - x[0];
```

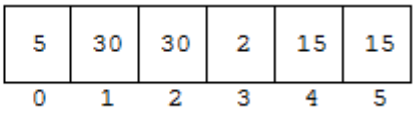


Figura 5. Asignación y recuperación de datos de un vector

Punteros

Un puntero es una variable que apunta o almacena direcciones de memoria de otra variable o creados por la misma variable, que siempre contiene una dirección de memoria (valida, inválida o nula), en la cual puede estar alojada una entidad de un programa o no.

Al momento de ser declarada una variable reserva memoria, la cual tiene una dirección de memoria. Ver figura 6

```
int a;
int b;
int c;
```

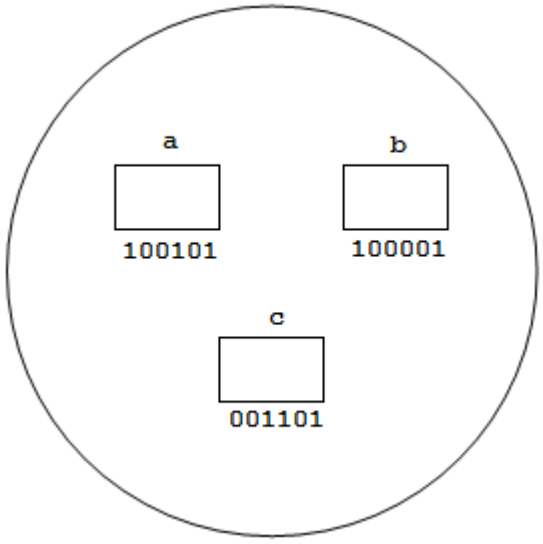


Figura 6. Reserva de memoria

Se puede observar en la figura 6, que las variables a, b y c tienen cada uno su propia dirección de memoria. Las variables punteros son los que apuntan a estas direcciones de memoria que las explicaremos más adelante con más detalle.

De la siguiente manera se declaran las variables punteros:

```
tipo_dato *nombre_puntero;
```

Se debe tener muy en cuenta que al momento de declarar un puntero se debe anteceder el operador ***** (asterisco) al nombre del puntero.

La asignación de direcciones de memoria a las variables punteros se realizan de dos maneras que son:

1. Con direcciones de memoria propia

```
int *px;
px = new int;
```

Figura 7. Asignación de dirección de memoria propia

El uso del operador **new** es muy importante, es la crea y asigna una dirección de memoria a las variables punteros.

2. Con direcciones de memoria de otras variables

```
int a = 10;
int b = 5;
int *pa;
int *pb;
pa = &a;
pb = &b;
```

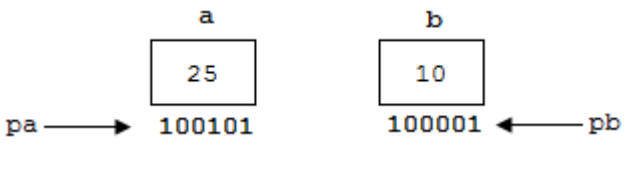


Figura 8. Asignación de memoria de otras variables

En la figura 8, se puede observar que las direcciones de memoria que se asigna a los punteros son de las variables a y b respectivamente.

El & (ampersand) permite extraer la dirección de memoria de una variable.

En la figura 8, el valor del puntero **pa** no es el valor de la variable **a**, sino la dirección de memoria de **a**, es decir, que el valor de **pa** es 100101. Es el mismo caso para el puntero **pb** contiene la dirección de memoria de la variable **b** y no el valor.

Como ya es de conocimiento que las variables punteos no almacenan valores sino direcciones de memoria propias o de otras variables, la pregunta ahora es como acceder a los valores de las direcciones de memoria.

Un puntero a partir de ahora se dividirá en dos partes muy importantes las cuales son: la dirección de memoria como tal, y el contenido de la dirección de memoria (el valor).

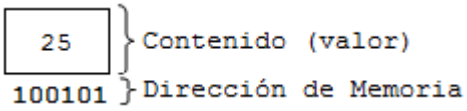


Figura 9. Partes de un puntero

Para poder acceder al contenido un puntero se debe utilizar el operador ***** (asterisco) seguido del nombre del puntero, similar a la declaración de una variable puntero.

```
1.  int a;
2.  int b;
3.  int c;
4.  int *pa;
5.  int *pb;
6.  int *pc;
7.  pa = &a;
8.  pb = b; // error, si intenta asignar el valor de b y no su dirección de
    memoria.
9.  pb = &b;
```

```
10.  pc = new int;
```

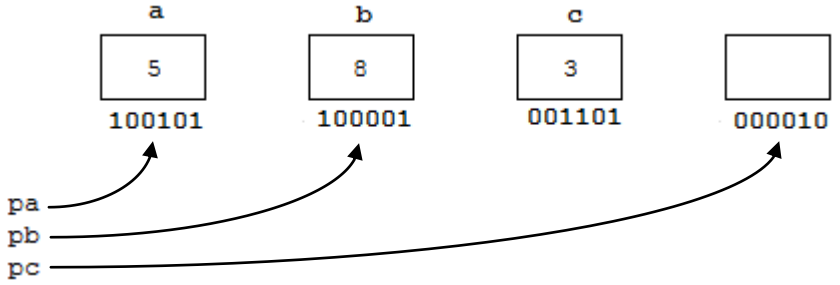


Figura 10. Declaración y asignación de direcciones de memoria

Se observa con detalle la figura 10, los punteros pa recupera la dirección de memoria de la variable a, y pb recupera la dirección de memoria de la variable b, sin embargo pc tiene su propia dirección de memoria.

```
11.  cout<<pa; // imprime la dirección de memoria, es decir, 100101
12.  cout<<pb; // imprime la dirección de memoria, es decir, 100001
13.  cout<<pc; // imprime la dirección de memoria, es decir, 000010
14.  cout<<*pa; // imprime el contenido, es decir, 5
15.  cout<<*pb; // imprime el contenido, es decir, 8
16.  cout<<*pc; // imprime el contenido, es decir, un número aleatorio debido a que
     no tiene ningún valor el contenido de esta dirección de memoria.
```

En las siguientes instrucciones se asignaran valores a los contenidos de los punteros o de las variables

```
17.  *pc = a;
18.  *pa = 2*c;
19.  *pb = a*b;
20.  cout<<a<<b<<c; // Al modificar los contenidos de los punteros, también se
     modificaron los valores de las variables a y b, debido a que los punteros están
     apuntando a las direcciones de memoria de las varibles. Por tanto a = 6, b = 48
     y  c = 3
21.  cout<<*pc; // El contenido de la dirección de memoria del puntero pc es ahora 5
     debido a que el contenido de esta dirección de memoria fue modificada antes de
     ser modificada el valor de la variable a. Por tanto su contenido vale 5.
```

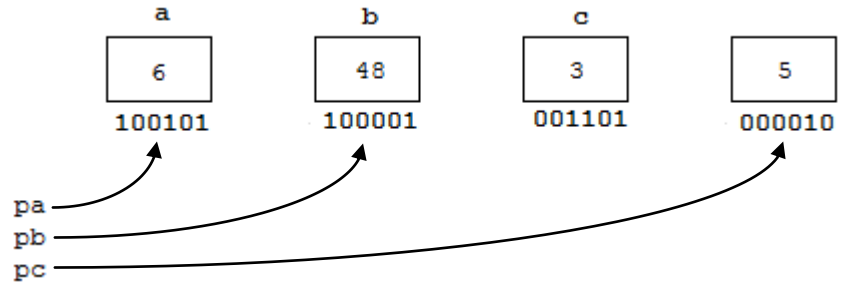


Figura 11. Modificación de contenidos de direcciones de memoria

Cuando las variables punteros fueron asignadas mediante direcciones de memoria propia (**new**), la única manera de poder apuntar o mover a otra dirección memoria primero se debe liberar o eliminar memoria que fue creada. Para realizar esta eliminación se usa la función **delete(variable_puntero)** la cual permitirá eliminar esta memoria y posterior a esto recién se puede

asignar una dirección de memoria diferente, como es el caso del puntero **pc**, tiene una dirección de memoria propia, por tanto primero se debe eliminar la memoria y posterior a esto se puede asignar otra dirección de memoria.

La función **delete** solo se utiliza cuando la variable puntero fue asignada con su dirección de memoria propia, y no así cuando apunta a la dirección de memoria de las variables.

```
22. delete(pc); // elimina la memoria creada
23. pc = pa; // pc apunta a la dirección de memoria a la que apunta pa
24. pb = &c; // pb apunta a la dirección de memoria de la variable c
25. pa = pb; // pa apunta a la dirección de memoria a la que apunta pb
```

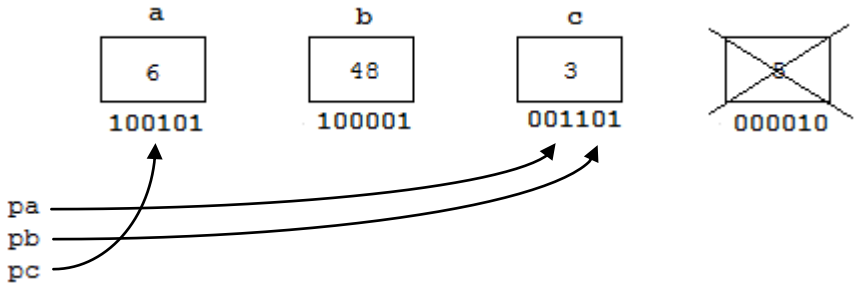


Figura 12. Modificación de direcciones de los punteros

```
26. cout<<a<<b<<c; // los valores son: a = 6, b = 48 y c = 3
27. cout<<*pa<<*pb<<*pc; // los contenidos de las direcciones de memoria
    son: pa = 3, pb = 3 y pc = 6
28. cout<<pa<<pb<<pc; // las direcciones de memoria son: pa = 001101, pb =
    001101 y pc = 100101
```

Se puede observar en la figura 12, y según las instrucciones 27 y 28 los punteros **pa** y **pb** tienen la misma dirección de memoria, por tanto tienen los mismos contenidos.

TEMA 1

PILAS

1.1. Definición

Es una colección lineal, dinámica y homogénea donde los elementos que insertan y eliminan son del mismo extremo. Se usa el concepto de últimos en entrar, primeros en salir derivado del método UEPS ó LIFO (last input, first output) en inglés.

Los elementos que se inserten o eliminen siempre deben ser por encima, es decir, si se desea insertar un nuevo elemento esto se insertara por encima quedando este elemento como el último de la pila, y cuando se desea eliminar el último elemento insertado a la pila será eliminado.

1.2. Representación gráfica

Se representa mediante un vector en forma vertical, en la cual lleva un apuntador denominado tope que determina el estado de la misma, además tiene una capacidad máxima (MAX) de elementos que puede soportar.

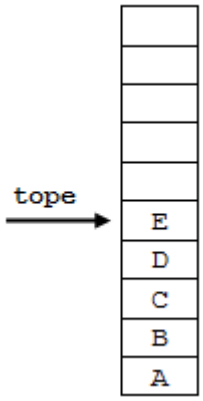


Figura 13. Representación de una pila

1.3. Partes

Tiene dos partes las cuales son:

1.3.1. Vector

Debe ser defina con tamaño exacto, es decir, la máxima capacidad que puede soportar la pila. Es aquí donde se almacena la información de la pila.

```
tipo_dato info[MAX];
```

El tipo de dato de este vector de ahora en adelante llamado campo información puede soportar cualquier tipo de dato ya sean primitivas o definidos por el usuario (ver tabla 1).

Al tratarse de un vector el primer elemento de la pila se encuentra en el índice 0 y el último elemento se encuentra en el índice MAX-1.

Primitivos	Definidos por el usuario
- short - char - int - float - double - long	- enum (enumeraciones) - struct (estructuras) - class (encapsulaciones)

Tabla 1. Tipos de datos

1.3.2. Tope

Es un apuntador que permite determinar el estado actual de la pila. Este apuntador es una variable de un tipo de dato entero variando en un rango desde -1 hasta MAX-1. Cuando este apuntador tiene un valor -1 significa que la pila está vacía y si tiene un valor de MAX-1 significa que la pila está llena.

```
int tope;
```

1.4. Estados

Existen dos estados dentro de una pila, que determina el estado actual de la pila las cuales son:

1.4.1. Vacía

Este estado es cuando en la pila no se encuentra ningún elemento. El apuntador **tope** es el que indica este estado, es decir, cuando el apuntador **tope** tiene el valor de -1 significa que no tiene ningún elemento dentro la pila. (Ver figura 14, a)

1.4.2. Llena

Este estado es cuando en la pila ya no se puede insertarse ningún elemento. El apuntador **tope** es el que indica este estado, es decir, cuando el apuntador **tope** tiene el valor de MAX-1, significa que la pila está llena. (Ver figura 14, b).

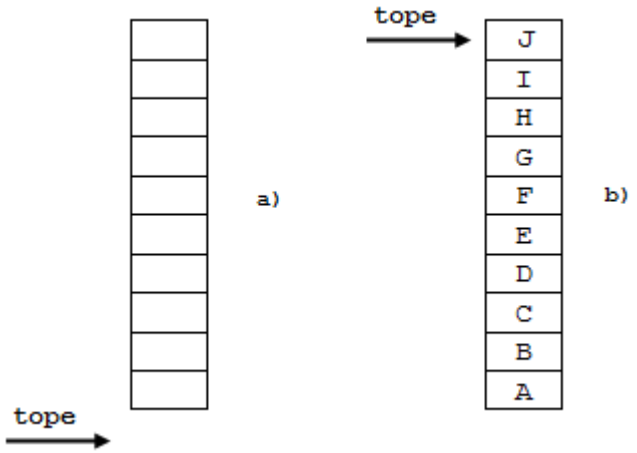


Figura 14. Estados de una pila.
a) Vacía, b) Llena

1.5. Operaciones básicas

Las operaciones básicas de una pila son dos:

1.5.1. Insertar

Esta operación se encarga de insertar o agregar nuevos elementos a la pila siempre por encima. Esta operación se debe realizar mientras la pila no esté lleno, es decir, que el apuntador tope no esté en MAX-1. (Ver figura 15).

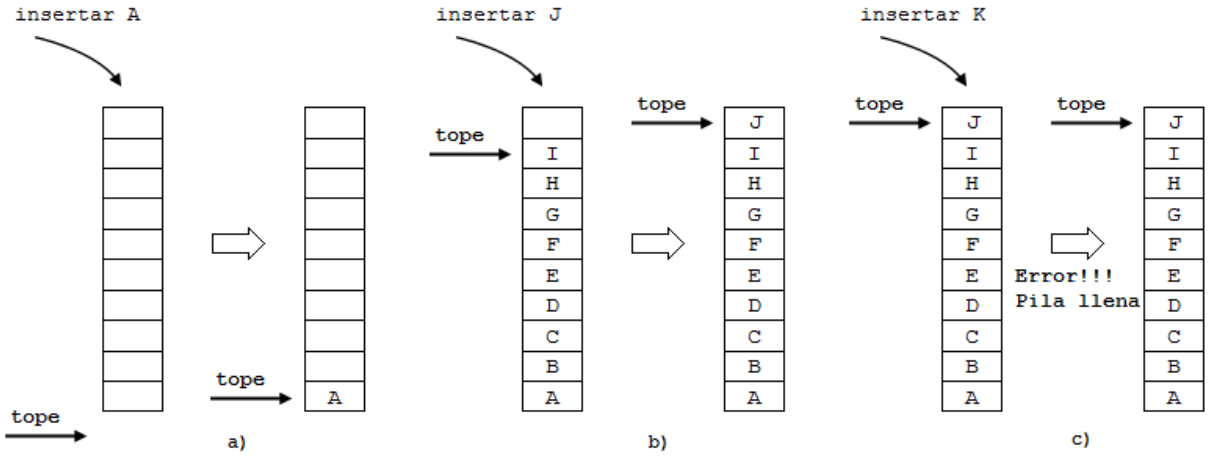


Figura 15. Insertar nuevos elementos
a) Primer elemento, b) Último elemento, c) Pila llena

1.5.2. Eliminar

Este método se encarga de eliminar o quitar los elementos de la pila siempre por encima, es decir, el último elemento insertado. Esta operación se debe

realizar mientras la pila no esté vacío, es decir, que el apuntador tope no esté en -1. (Ver figura 16).

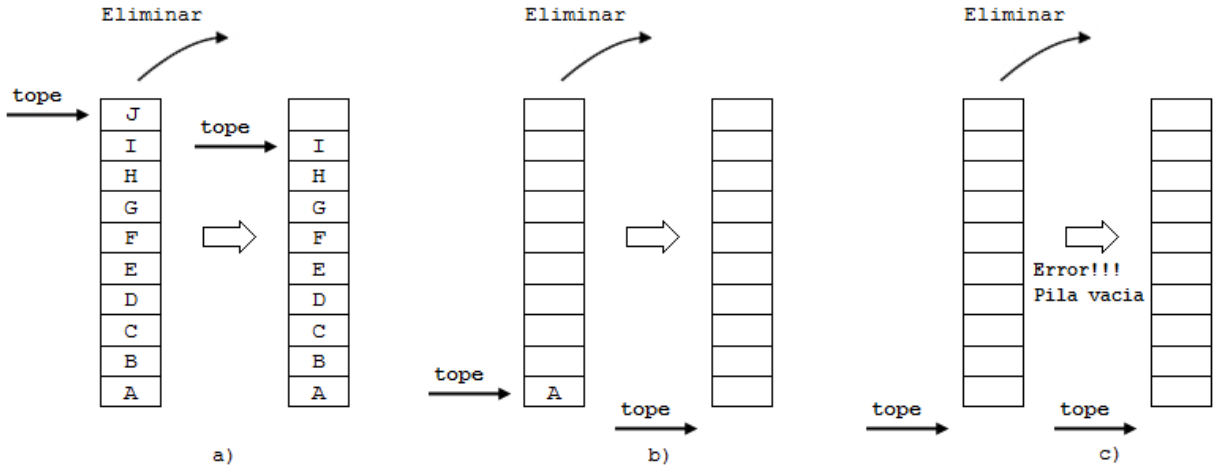


Figura 16. Eliminar un elemento

a) El último elemento insertado, b) El último elemento de la pila, c) Pila vacía

1.6. Ejemplos en la vida real

Al hablar de Pilas se habla de apilaciones de objetos una encima de otra, y como ejemplos se tiene muchos. (Ver figura 17)

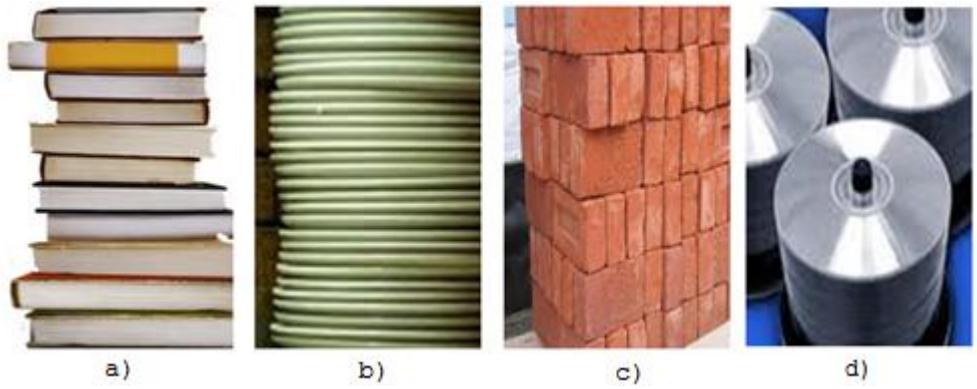


Figura 17. Pilas en vida real

De la figura 17, se observa algunos ejemplos de pilas aplicados en la vida real las cuales son:

a) Pila de libros
b) Pila de platos
c) Pila de ladrillos
d) Pila de CD/DVD

1.6.1. Ejercicios

Buscar y escribir 5 ejemplos de pilas aplicados en la vida real.

1.7. Ejemplos en la informática

De la misma manera se puede listar una serie de ejemplos de pilas aplicados en la informática. (Ver figura 18)

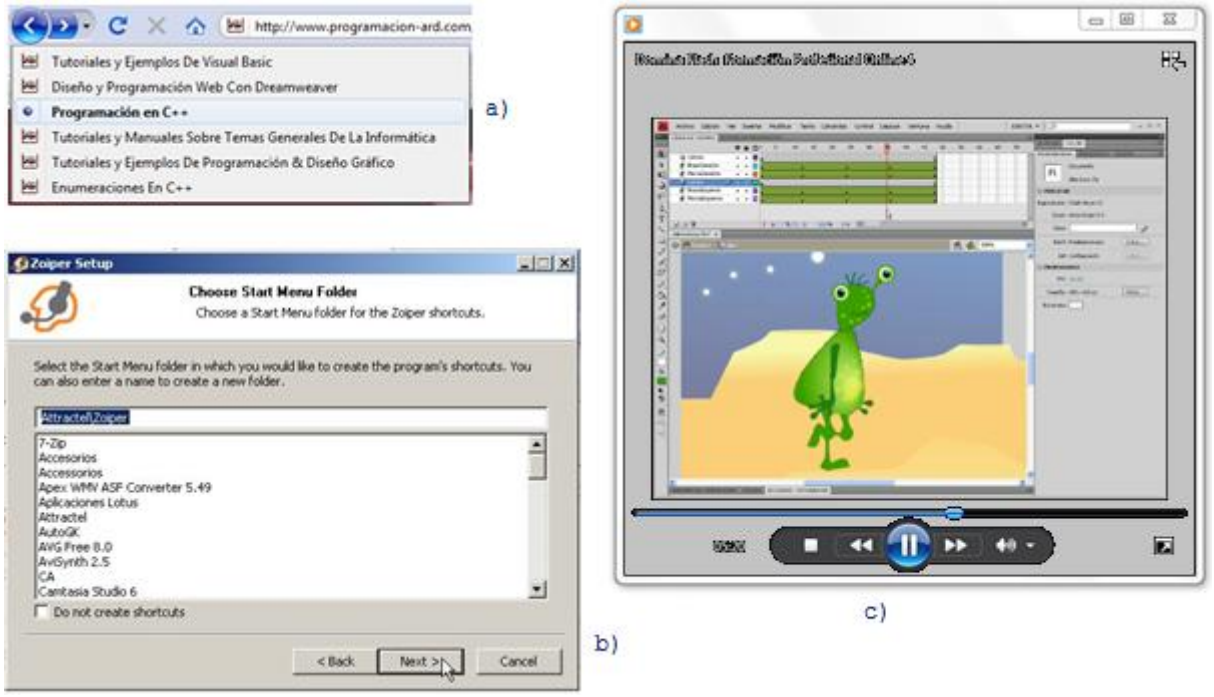


Figura 18. Ejemplos en la informática.

En la figura 18 se observa algunos ejemplos aplicados en la informática, las cuales son:

a) En un navegador, cuando se desea volver atrás o ir adelante después de volver atrás. Siempre vuelve a la útlima página que se visito.
b) Al momento de intalar una aplicación o programa, se sigue los pasos en forma sucesiva presionando sobre el botón siguiente (Next) en el caso de equivocarse en alguno de los pasos se presiona atrás (Back) que volverá al útlimo paso que se realizó.
c) Cuando se reproducen videos, la duración del tiempo de esta reproducción es forma progresiva y ascendentemente en el caso de que se desee retroceder, el tiempo descenderá en forma progresiva.

1.7.1. Ejercicios

Buscar y escribir 5 ejemplos de pilas aplicados en la informática.

1.8. Estructura de datos

La estructura de datos de una pila se representa a partir de las partes de la misma. Esta estructura de datos se la representará mediante la Programación Orientado a Objetos.

Esta estructura se la escribirá en dos archivos las cuales son: los archivos cabecera (*.h) donde se encuentra las declaraciones de las clases y los archivos cuerpos (*.cpp) donde se encuentra las implementaciones de las clases declaradas en los archivos *.h.

```
// Declaración de cabeceras
// clase.h
#define MAX 10
class Pila {
    private:
        char info[MAX];
        int tope;
    public:
        Pila();
        bool vacia();
        bool llena();
        void insertar(char x);
        char eliminar();
        void mostrar();
};

// Definición de un tipo de dato
// en forma de puntero
typedef Pila *pPila;
```

```
// Implementación de clases
// clase.cpp
#include "iostream.h"
#include "clase.h"
Pila::Pila() {
    tope=-1;
}
bool Pila::vacia() {
    return (tope==-1);
}
bool Pila::llena() {
    return (tope==TAM-1);
}
void Pila::insertar(char x) {
    if (!llena())
        info[++tope]=x;
}
char Pila::eliminar() {
    char x=' ';
    if (!vacia())
        x=info[tope--];
    return x;
}

void Pila::mostrar() {

    for (int i=tope; i>=0; i--)

        cout<<(tope-i+1)<<".- "

            <<info[i]<<endl;

}
```

Las partes de una pila forman parte de la estructura de datos que se convierten en atributos privados, de la misma manera los estados y las operaciones básicas de una pila también forman parte de la estructura de datos que se convierten en métodos públicos.

Cada uno de los estados y operaciones básicas de ahora en adelante métodos tienen un objetivo específico como se mencionó en un apartado anterior. A continuación se dará una explicación de cada uno de los métodos.

Pila(); Constructor de la clase, que se encarga de inicializar el apuntador tope en -1. Cada vez que se invoque a este constructor se crea una nueva pila vacía (Ver figura 14, a) asignando una nueva dirección de memoria.

bool vacia(); Este método representa el estado vacía devolviendo el valor de verdad (true) si tope es igual a -1 o falso (false) caso contrario.

bool llena(); Este método representa el estado llena devolviendo el valor de verdad (true) si tope es igual a MAX-1 o falso (false) caso contrario.

void insertar(char x); Este método representa la operación básica de inserción, recibe un parámetro del mismo tipo de dato del campo información en este caso de tipo **char** y no devuelve ningún valor. Este parámetro es el elemento que va a ser insertado en la pila siempre por encima, es por eso que primero se incrementa y luego se asigna el valor del parámetro al vector. (Ver figura 15)

Para realizar esta asignación se debe realizar una previa comprobación para determinar si la pila no está llena invocando al método **llena**.

char eliminar(); Este método representa la operación básica de eliminación, no recibe ningún parámetro y devuelve un valor del mismo tipo de dato del campo información, este valor es elemento que se elimina siempre por encima, es por eso que primero se recupera el valor y luego se decrementa el apuntador tope. (Ver figura 16)

Para recuperar el valor que se está eliminando se debe realizar una previa comprobación de que la pila no esté vacía invocando al método **vacía**.

void mostrar(); Este método muestra los elementos que tiene la pila.

1.9. Operaciones avanzadas

Las operaciones avanzadas son implementadas a partir de las operaciones básicas, estas operaciones pueden ser invertir, unir, etc.

Para implementar las operaciones avanzadas se recomienda analizar las mismas con gráficas y teniendo presente siempre que ya se tiene implementada la estructura de datos. Al implementar estas operaciones ya no se deben trabajar con vectores, sino se debe crear una instancia a partir de la clase Pila como un puntero e ir invocando los métodos necesarios esta la clase para implementar adecuadamente las diferentes operaciones.

De ahora en adelante todas las operaciones avanzadas se implementan dentro de una clase denominada OPila (Operaciones sobre Pilas).

```
// Declaración de la clase OPila (*.h)
// clase.h
class OPila {
    private:
        pPila p;
    public:
        OPila();
        ~OPila();
        void setPila(pPila p);
        pPila getPila();
        void agregar(char x);
        void quitar();
        void imprimir();
             .
             .
             .
};
```

```
// Implementación de la clase OPila (*.cpp)
// clase.cpp
OPila::OPila() {
    p = new Pila();
}
```

Este método es el constructor de la clase, la cual se encarga de crear y asignar una dirección de memoria a la pila **p**. Se debe invocar a este método cuando se crea una nueva instancia de la clase OPila.

```
OPila::~OPila() {
    delete(p);
}
```

Este método es el destructor de la clase, la cual se encarga de liberar la dirección de memoria de la pila **p**. Se debe invocar a este método al final de todas las instrucciones, es decir, cuando ya no se tenga que usar la pila **p**.

```
void OPila::setPila(pPila p) {
    delete(this->p);
    this->p=p;
}
```

Este es un método mutador, la cual se encarga de liberar la dirección de memoria de la pila **p**, y posteriormente asignar a otra dirección de memoria que en este caso viene a ser la dirección de memoria del parámetro **p**.

```
pPila OPila::getPila() {
    return p;
}
```

Este es un método accesorio, la cual se encarga de devolver la dirección de memoria de la pila **p**, mediante el nombre del método. Es muy importante que el tipo de dato del método deba ser igual al del atributo.

```
void OPila::agregar(char x) {
    if (!p->llena())
        p->insertar(x);
    else
        cout<<"Pila llena"<<endl;
}
```

```
void OPila::quitar() {
    if (!p->vacia())
        cout<<"El elemento eliminado es: "<<p->eliminar()<<endl;
    else
        cout<<"Error: Pila vacia"<<endl;
}
void OPila::imprimir() {
    p->mostrar();
}
```

Se puede observar que en la declaración de la clase OPila tiene un atributo único (**p**) que viene a ser una instancia la estructura de datos de una Pila, además que es puntero. Significa que este atributo representa a la pila principal de la clase.

Al ser el atributo **p** una instancia puntero el acceso a los atributos y/o métodos públicos de la clase debe ser mediante el operador **->**.

Todas los anteriores ejemplos y los siguientes (métodos) deben ser invocadas como instancia de la clase OPila en el programa principal **main()**

1.10. Ejemplos.

Para resolver cualquiera de los métodos se tiene que pensar en el concepto de pila, es decir, siempre insertar o eliminar por encima y no así trabajar como vector.

1. Realizar un método para invertir una pila

Se sigue los siguientes pasos para poder invertir una pila. (Ver figura 19)

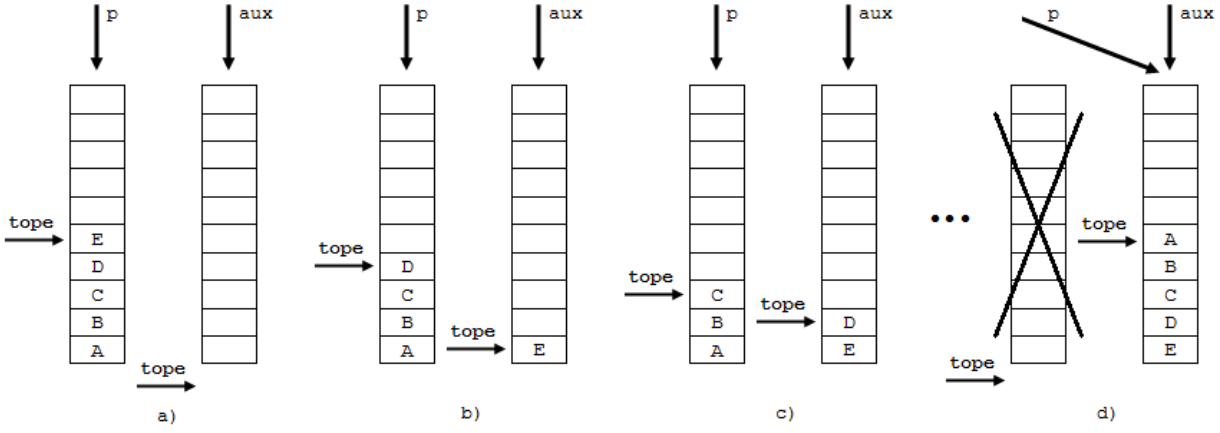


Figura 19. Invertir una pila

a. Crear una nueva pila como se muestra en la figura 19 a). Esta pila también debe ser una instancia de pila puntero denominada **aux.**

b. Usando las operaciones básicas de una pila eliminar un elemento de la pila **p** que en este caso viene a ser el elemento **E,** e insertar este elemento en la pila aux como se muestra en la figura 19 b) y c)
c. Seguir con el paso del inciso b) hasta que la pila **p** esté vacía.
d. Una vez que la pila **p** esté completamente vacía, eliminar o liberar la dirección de memoria de la pila **p** y asignar la dirección de memoria de la pila **aux**, como se muestra en la figura 19 d).

Finalmente devolver la dirección de memoria de la pila **aux** como resultado del método.

El código correspondiente a la inversión de una pila es la siguiente.

```
pPila OPila::invertir(pPila p) {
    pPila aux = new Pila();
    while (!p->vacia())
        aux->insertar(p->eliminar());
    delete(p);
    return aux;
}
```

Se puede observar que el método es parte de la clase OPila.

2. Insertar ordenadamente un elemento en forma ascendente

Para insertar elementos en forma ordenada, se debe ubicar primero la posición exacta en el cual va a ser insertado.

Los siguientes pasos son los que se debe seguir para insertar elementos en forma ordenada. (Ver figura 20)

El elemento a insertar será **G**.

a. Primero se debe verificar que la pila no esté llena, posteriormente crear una nueva pila **aux** instancia puntero de la clase Pila como se muestra en la figura 20 a).
b. Se va eliminando los elementos de la pila **p** hasta que esta quede vacía o se encuentre un elemento eliminado menor a lo que se desea insertar. En el caso de que el elemento eliminado sea mayor se inserta a la pila **aux** como se muestra en la figura 20 b) y c).
c. Repetir el paso b.
d. En el caso de que el elemento eliminado de la pila **p** sea menor al elemento a insertar, entonces este elemento no debe ser insertado a la pila **aux** y el valor del elemento eliminado se queda guardada en una variable como se muestra en la figura 20 d).

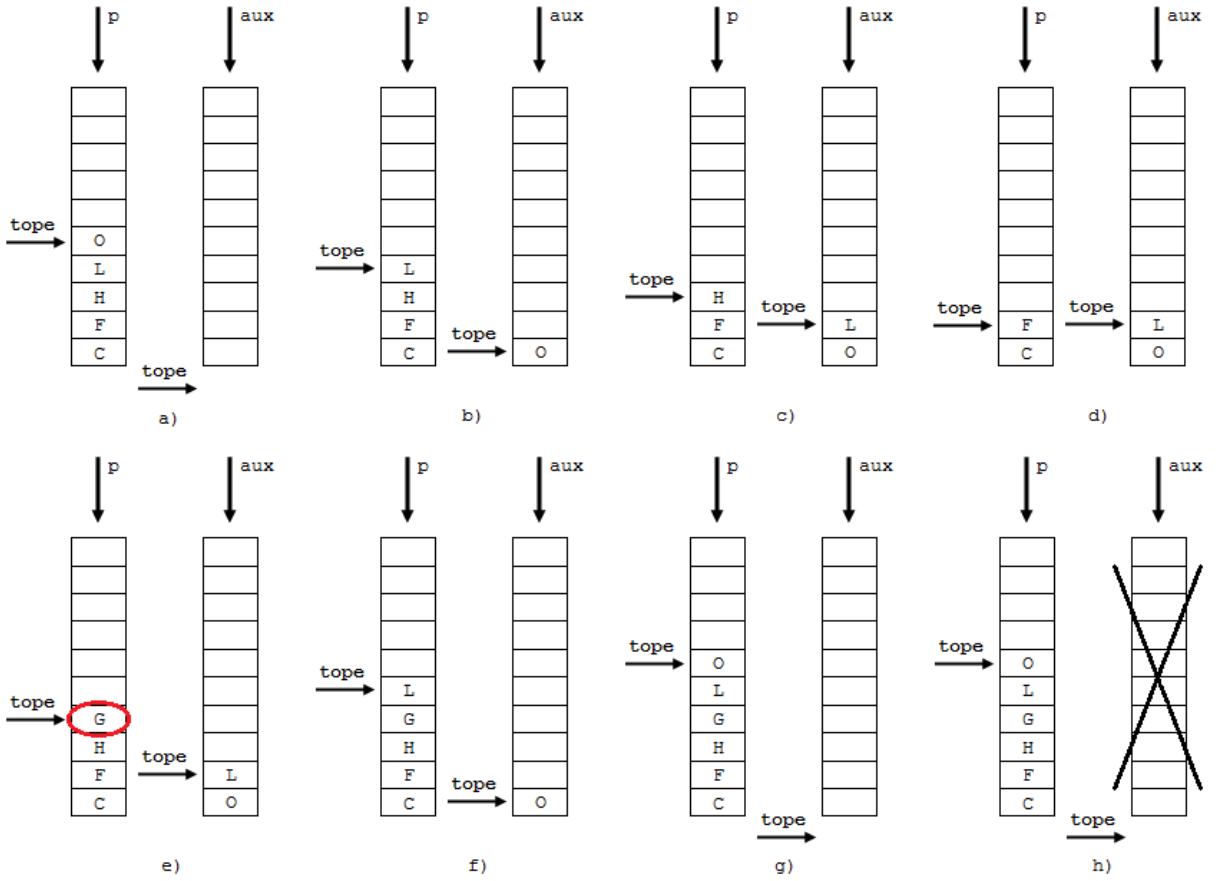


Figura 20. Inserción en forma ordenada

e. El último elemento eliminado de la pila **p** que está guardada en una variable en el paso anterior, se inserta nuevamente en la pila **p** y posteriormente insertar el nuevo elemento, como se muestra en la figura 8 e).
f. Los elementos que quedaron en la pila **aux** se eliminan y se insertar nuevamente en la pila **p** hasta que la pila **aux** quede completamente vacía como se muestra en la figura 20 f) y g).
g. Repetir el paso f.
h. Como se puede observar en la figura 20 h) el elemento ha sido insertado a la posición correcta. Finalmente se debe eliminar o liberar memoria de la pila **aux**.

El código correspondiente al método para insertar un elemento ordenadamente en forma ascendente es el siguiente.

```
void OPila::inser_orden(char x) {
    if (!p->llena()) {
        pPila aux = new Pila();
        char y; bool ins=false;
        while (!p->vacia()&&!ins) {
```

```
            y = p->eliminar();
            if (y>x)
                aux->insertar(y);
            else {
                p->insertar(y);
                p->insertar(x);
                ins = true;
            }
        }
        if (!ins) aux->insertar(x);
        while (!aux->vacia())
            p->insertar(aux->eliminar());
        delete(aux);
    }
}
```

Se puede observar que el método es parte de la clase OPila.

3. Insertar un elemento al principio de la pila

Para insertar un elemento al principio se invertir toda la pila e insertar el nuevo elemento y finalmente invertir la pila nuevamente, donde el nuevo elemento insertado quede al principio de la pila.

Los pasos que se siguen son los siguientes.

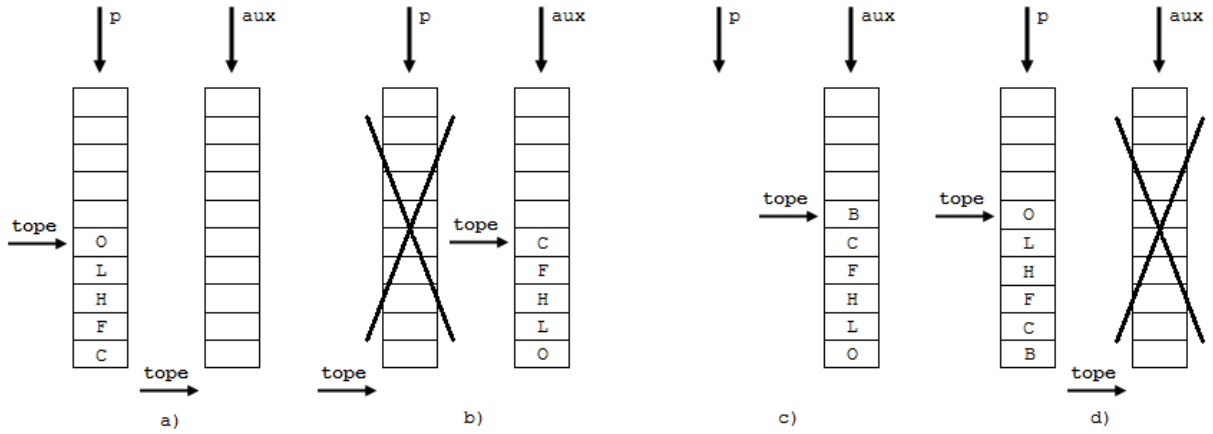


Figura 21. Insertar un elemento al principio

El elemento que se quiere insertar es B.

a. Primero se debe verificar que la pila **p** no esté llena y crear una pila **aux** instancia puntero de la clase Pila como se muestra en la figura 21 a).
b. Es mucho más sencillo invertir la pila **p** sobre la pila **aux** quedando la misma con los elementos invertidos de la pila **p**, además no se debe olvidar que el método invertir elimina o libera la dirección de memoria de la pila a invertir en este caso la pila **p** como se muestra en la figura 21 b).

c. Se debe insertar el nuevo elemento sobre la pila **aux**, como se muestra en la figura 21 c).
d. Finalmente invertir los elementos de la pila **aux** sobre la pila **p**, quedando el último elemento insertado al principio como se muestra en la figura 21 d).

El código correspondiente para realizar este método es el siguiente.

```
void OPila::inser_principio(char x) {
    if (!p->llena()) {
        pPila aux = invertir(p);
        aux->insertar(x);
        p = invertir(aux);
    }
}
```

4. Eliminar el primer elemento de la pila

Esta operación es casi similar al caso anterior, nomas que en este caso se debe eliminar el primer elemento de la pila.

Como se muestra en la figura 22 a) se debe eliminar el elemento C, para esta operación se sigue los siguientes pasos.

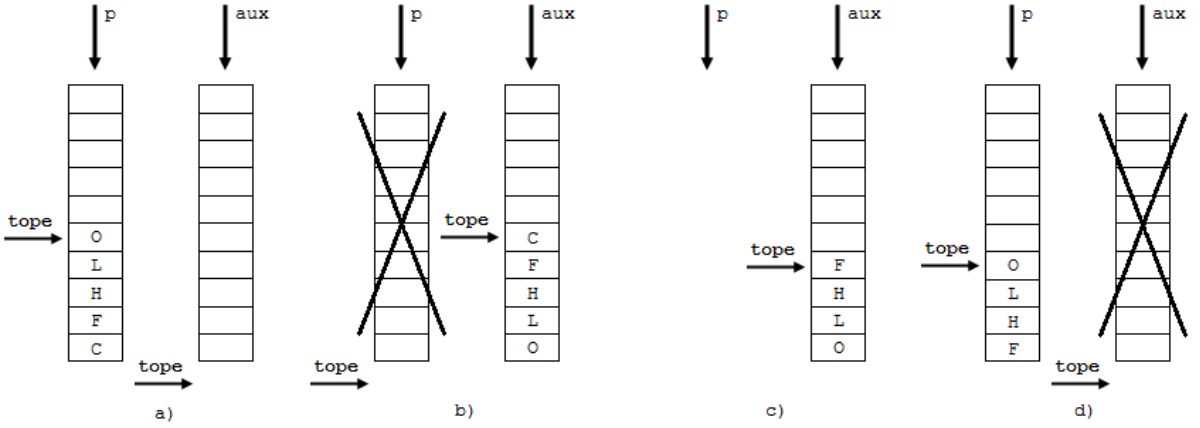


Figura 22. Eliminar el primer elemento de la pila

a. Primero se debe verificar que la pila **p** no esté vacía, posteriormente crear una pila **aux** instancia de la clase Pila como se muestra en la figura 22 a).
b. Invertir los elementos de la pila **p** sobre la pila **aux** quedando como muestra la figura 22 b).
c. Haciendo uso de la operación básica eliminar, se elimina el elemento de la pila **aux** que en este caso viene a ser el elemento C como se muestra en la figura 22 c).

d. Finalmente invertir los elementos de la pila **aux** sobre la pila **p** y se tiene una pila sin el primer elemento como se muestra en la figura 22 d).

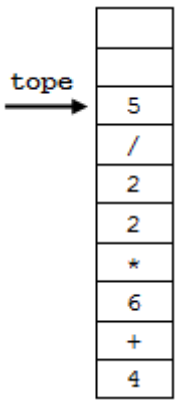
A continuación se tiene el código correspondiente al método.

```
void OPila::eli_principio() {
    if (!p->vacia()) {
        pPila aux = invertir(p);
        cout<<"El primer elemento eliminado es: "<<aux->eliminar()<<endl;
        p = invertir(aux);
    }
}
```

1.11. Ejercicios

Realizar los siguientes métodos

1. Eliminar el primer elemento de una pila igual a un determinado elemento.
2. Eliminar el primer elemento de una pila igual a un determinado elemento y posteriormente inserte este elemento a la pila.
3. Eliminar todos los elementos iguales a un determinado elemento.
4. Eliminar todos los elementos repetidos no consecutivos dejando solo el primer elemento de los repetidos.
5. Eliminar todos los elementos repetidos consecutivos dejando solo el primer elemento de los repetidos.
6. Insertar el primer elemento que menos veces se repite.
7. Insertar un determinado elemento por encima del primer elemento que sea igual a la misma.
8. Unir dos pilas.
9. Intercalar dos pilas
10. Suponiendo que en una pila solo se almacenan números uno por casilla y signos de operación (+, -, * y /), calcular el resultado de la pila en forma numérica.
    De la siguiente figura de pila se tiene como expresión matemática: 4 + 6 * 22 / 5 donde el resultado de esta expresión es 44. La operación se debe realizar de forma comercial, es decir, sin tomar en cuenta la prioridad de signos.



TEMA II

COLAS

2.1. Definición

Es una colección lineal, dinámica y homogénea donde los elementos que se insertan van por un extremo y los que se eliminan van por el otro extremo. Se usa el concepto de primeros en entrar, primeros en salir derivado del método PEPS ó FIFO (first input, first output) en inglés.

Una cola representa la idea que tenemos de cola en la vida real.

2.2. Representación gráfica

Se representa mediante un vector en forma horizontal, en la cual tiene dos apuntadores denominados primero (apunta al primer elemento de la cola) y último (que apunta al último elemento insertado a la cola), además tiene una capacidad máxima (MAX) de elementos que puede soportar.

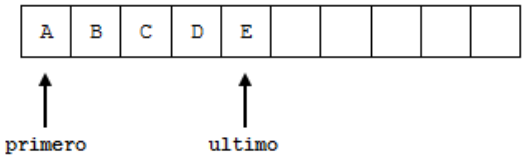


Figura 23. Representación de un Cola

2.3. Partes

Según la figura 23 se puede observar que una cola tiene tres partes las cuales son:

2.3.1. Vector

El vector debe ser defina con tamaño exacto, es decir, la máxima capacidad que puede soportar la cola. Es aquí donde se almacena la información de la cola.

```
tipo_dato info[MAX];
```

El tipo de dato de este vector de ahora en adelante llamado campo información puede soportar cualquier tipo de dato ya sean primitivas o definidos por el usuario (ver tabla 1).

Al tratarse de un vector el primer elemento de la cola se encuentra en el índice 0 y el último elemento se encuentra en el índice MAX-1.

2.3.2. Principio

Es un apuntador que siempre apuntará al primer elemento de la cola. Este apuntador es una variable de un tipo de dato entero que por lo general no se mueve de esta posición a no ser que sea algún tipo diferente de cola.

```
int primero;
```

2.3.3. Ultimo

Es un apuntador que permite determinar el estado actual de la cola. Este apuntador es una variable de un tipo de dato entero variando en un rango desde -1 hasta MAX-1. Cuando este apuntador tiene un valor -1 significa que la cola está vacía y si tiene un valor de MAX-1 significa que la cola está llena.

```
int ultimo;
```

2.4. Estados

Existen dos estados dentro de una cola, que determina el estado actual de la cola las cuales son:

2.4.1. Vacía

Este estado es cuando en la cola no se encuentra ningún elemento. El apuntador último es el que indica este estado, es decir, cuando el apuntador último tiene el valor de -1 significa que no tiene ningún elemento dentro la pila. (Ver figura 24, a).

2.4.2. Llena

Este estado es cuando en la cola ya no se puede insertarse ningún elemento. El apuntador **último** es el que indica este estado, es decir, cuando el apuntador **último** tiene el valor de MAX-1, significa que la cola está llena. (Ver figura 24, b).

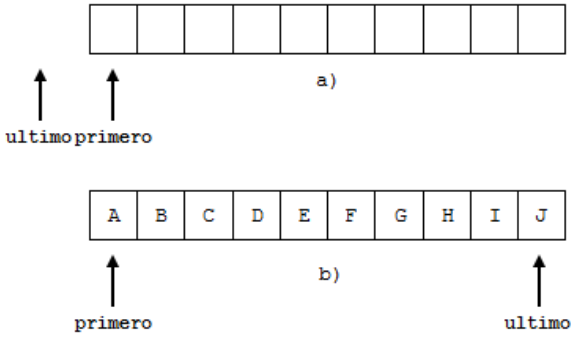


Figura 24. Estados de una Cola

2.5. Operaciones básicas

En toda cola se realiza diferentes operaciones básicas las cuales son:

2.5.1. Insertar

Esta operación se encarga de insertar o agregar nuevos elementos a la cola siempre al final de la cola. Esta operación se debe realizar mientras la cola no esté llena, es decir, mientras el apuntador último no esté en MAX-1. (Ver figura 25).

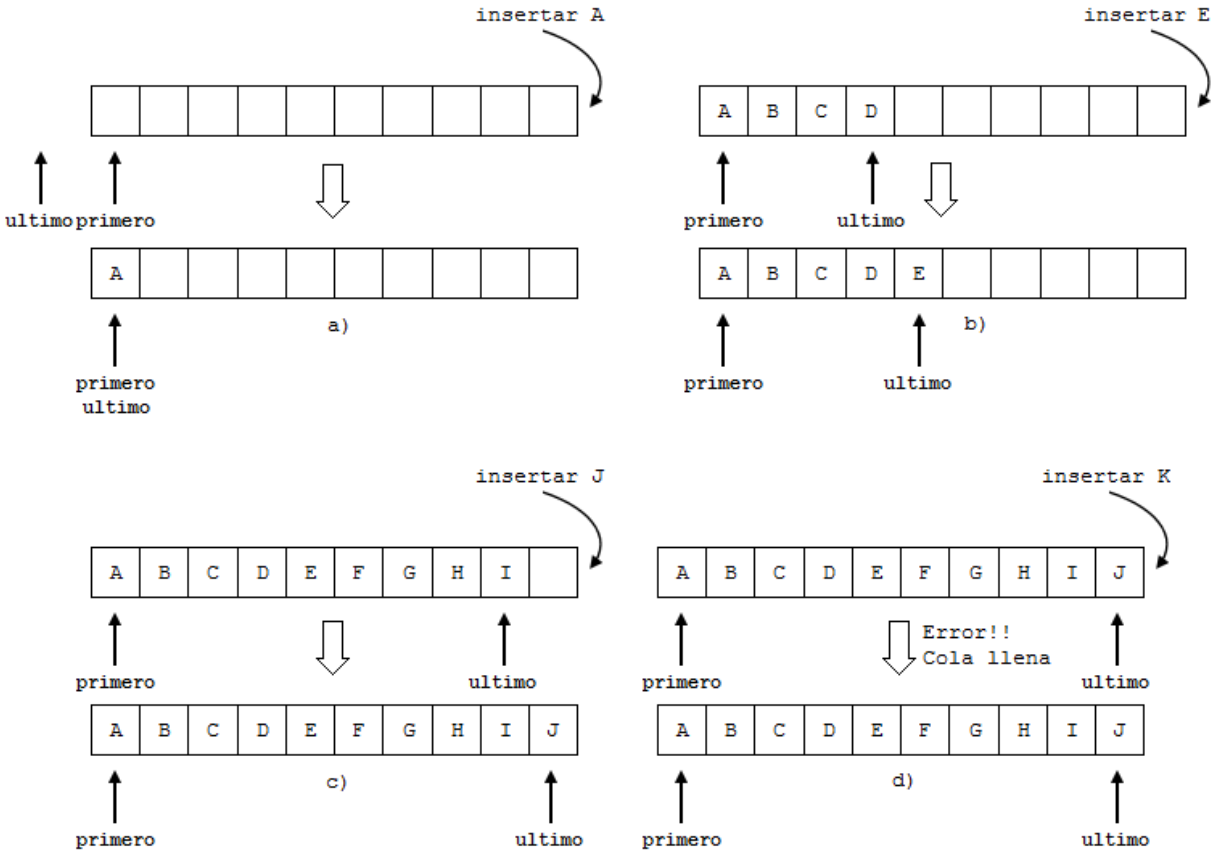


Figura 25. Operación básica insertar

2.5.2. Eliminar

Este método se encarga de eliminar o quitar elementos de una cola siempre por delante o el frente, es decir, el primer elemento insertado. Esta operación se debe realizar mientras la cola no esté vacía, es decir, mientras el apuntador último no esté en -1. (Ver figura 26).

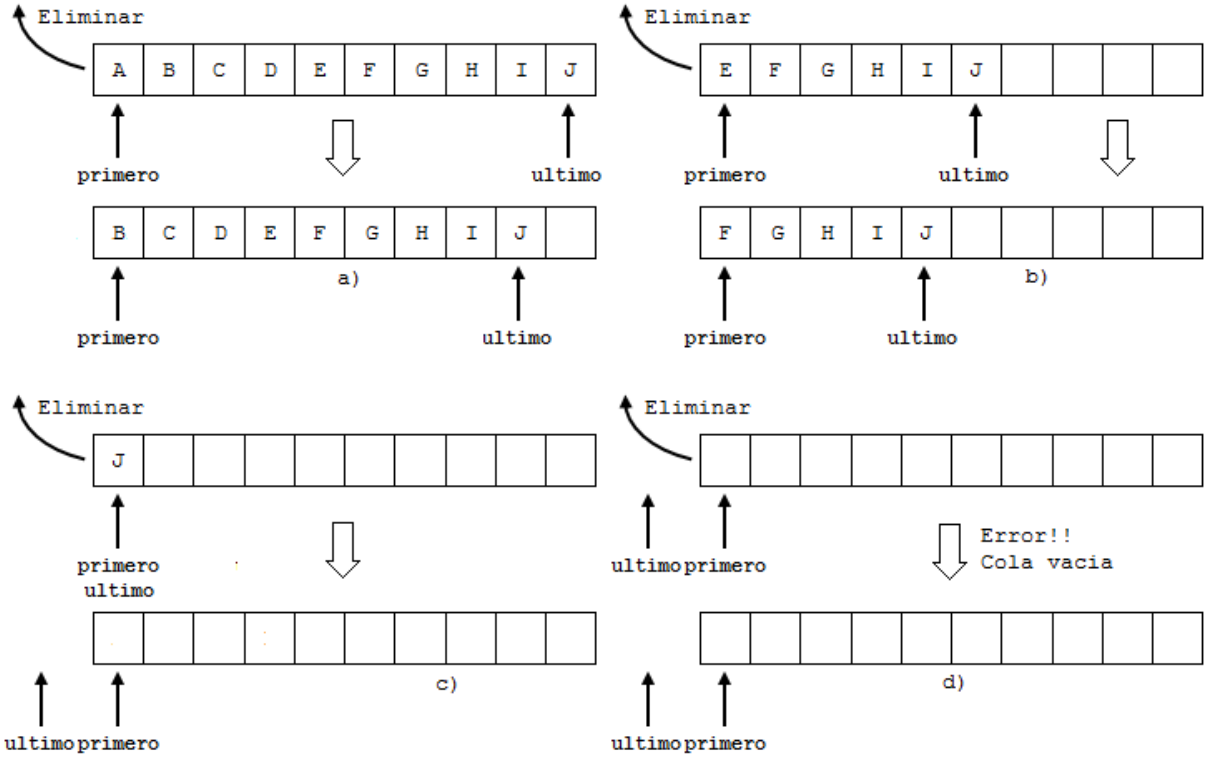


Figura 26. Operación básica insertar

2.6. Ejemplos en la vida real

Se puede encontrar una serie de ejemplos de colas aplicados en la vida real como se muestra en la figura 27.

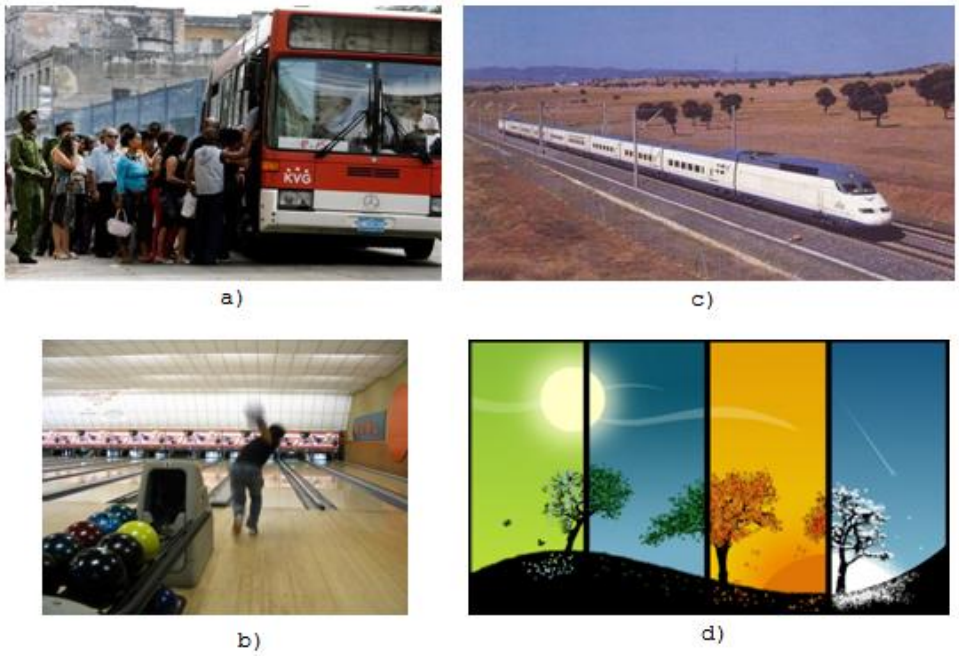


Figura 27. Colas aplicadas en la vida real

a) Una cola para subir a un micro bus, el primero que llega a la fila o cola a partir de ahora será el primero en entrar al bus. El último que llega tendrá que esperar hasta que el resto de la cola ingrese al bus.
b) En un juego de bolos, las bolas para derribar los pinos están ordenadas en el que el primero que llega a la cola será el primer en ser lanzado para derribar los pinos y esta bola regresará pero será insertada al final de la cola de bolas.
c) Las estaciones del año, en cualquier año la primera estación que llega es el verano y debe pasar esta estación para que llegue la siguiente estación la misma estación deberá ponerse a la cola, es decir, al final de las estaciones hasta el próximo año.
d) Una cola de vagones de tren, en donde el primero que parte de su origen es el primero en llegar a su destino.

2.6.1. Ejercicios

Enumerar cinco ejemplos de colas aplicados en la vida real.

2.7. Ejemplos en la informática

Se pueden enumerar también ejemplos aplicados dentro la informática como se puede observar en la figura 38.

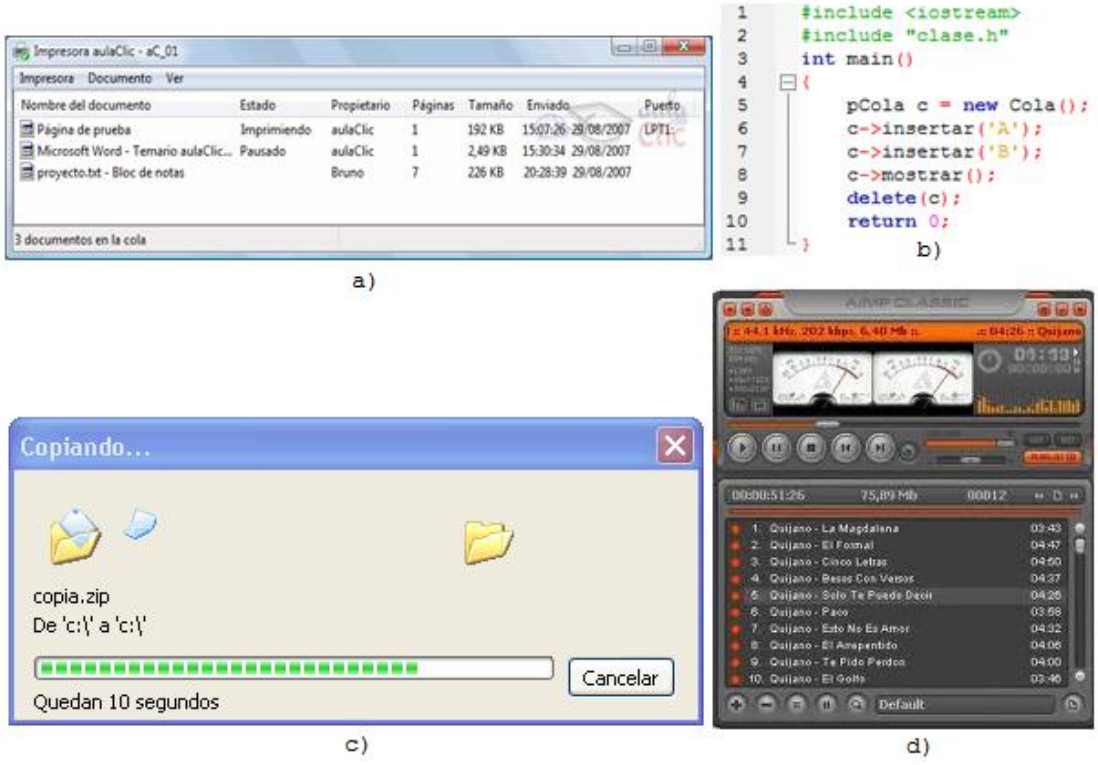


Figura 28. Colas aplicadas en la informática

a) Una cola de impresión, donde el primer documento enviado a imprimir será el primero en ser impreso.
b) En la compilación o intérprete de códigos, no se debe olvidar que al momento de ser compilado o interpretado los códigos siempre se realiza de arriba hacia abajo, es decir, lo primero que es escrito es el primero en ser compilado o interpretado.
c) Trasferencia de datos, el primer dato enviado será el primer dato en llegar a su destino.
d) Reproducción de una lista de música, el primer tema insertado a la lista será el primero en ser reproducida.

2.7.1. Ejercicios

Enumerar cinco ejemplos de colas aplicados en la informática.

2.8. Estructura de datos

La estructura de datos de una cola se representa a partir de las partes de la misma. Esta estructura de datos se representa mediante la Programación Orientado a Objetos.

Esta estructura se escribe en dos archivos las cuales son: los archivos cabecera (*.h: header) donde se encuentra las declaraciones de las clases, definición de constantes entre otros y los archivos cuerpos (*.cpp: source) donde se encuentra las implementaciones de las clases declaradas en los archivos *.h.

```
// Declaración de cabeceras
// clase.h
#define MAX 10
class Cola {
    private:
        char info[MAX];
        int primero;
        int ultimo;
    public:
        Cola();
        bool vacia();
        bool llena();
        void insertar(char x);
        char eliminar();
        void mostrar();
};
typedef Cola *pCola;
```

```
// Implementación de cabeceras
// clase.h
Cola::Cola() {
    primero = 0;
    ultimo = -1;
}
bool Cola::vacia() {
    return (ultimo == -1);
}
bool Cola::llena() {
    return (ultimo == MAX-1);
}
void Cola::insertar(char x) {
    if (!llena())
        info[++ultimo] = x;
}
```

```
char Cola::eliminar() {
    char x = ' ';
    if (!vacia()) {
        x = info[primero];
        for (i=1; i<=ultimo; i++)
            info[i-1] = info[i];
        ultimo--;
    }
    return x;
}
```

```
void Cola::mostrar() {
    for (int i=0; i<=ultimo; i++)
        cout<<info[i]<<"\t";
}
```

Como se puede observar en el código anterior las partes de una cola forman parte de la estructura de datos que se convierten en atributos privados, de la misma manera los estados y las operaciones básicas de una cola también forman parte de la estructura de datos que se convierten en métodos públicos.

Cada uno de los estados y las operaciones básicas de ahora en adelante métodos tienen un objetivo específico como se mencionó en un apartado anterior. A continuación se dará una explicación de cada uno de los métodos.

Cola(); Constructor de la clase, que se encarga de inicializar el apuntador último en -1 y primero que inicia su valor en 0 que apunta al primer elemento de la colas. Cada vez que se invoque a este constructor se crea una nueva cola vacía (Ver figura 24, a) asignando una nueva dirección de memoria.

bool vacia(); Este método representa el estado vacía de la cola devolviendo el valor de verdad (true) si último es igual a -1 o falso (false) caso contrario.

bool llena(); Este método representa el estado llena de la cola devolviendo el valor de verdad (true) si último es igual a MAX-1 o falso (false) caso contrario.

void insertar(char x); Este método representa la operación básica de inserción de una cola, recibe un parámetro del mismo tipo de dato del campo información en este caso de tipo de dato **char** y no devuelve ningún valor. Este parámetro es el nuevo elemento que va a ser insertado en la cola siempre al final, es por eso que primero se incrementa y luego se asigna el valor del parámetro al vector. (Ver figura 25)

Para realizar esta asignación se debe realizar una previa comprobación que determina si la cola no está llena invocando al método **llena.**

char eliminar(); Este método representa la operación básica de eliminación de una cola, no recibe ningún parámetro pero sí devuelve un valor del mismo tipo de dato del campo información, este valor es el elemento que se elimina de la cola que es siempre el primero, es por eso que primero se recupera el valor del primer elemento y luego se recorre el resto de los elementos de la cola hacia adelante, quedando el segundo elemento en el principio y finalmente se decrementa el apuntador último. (Ver figura 26)

Para recuperar el valor que está eliminando debe realizar una previa comprobación de que la cola no esté vacía invocando al método **vacía.**

void mostrar(); Este método muestra los elementos que tiene la cola.

2.9. Operaciones avanzadas

Las operaciones avanzadas son implementadas a partir de las operaciones básicas, estas operaciones pueden ser eliminar un determinado elemento, unir, entre otros.

Para implementar las operaciones avanzadas se recomienda analizar las mismas mediante gráficas y teniendo presente siempre que ya se tiene implementada la estructura de datos. Al implementar estas operaciones ya no se deben trabajar con vectores, sino se debe crear una instancia a partir de la clase Cola como un puntero e ir invocando los métodos necesarios de esta clase para implementar adecuadamente las diferentes operaciones.

De ahora en adelante todas las operaciones avanzadas se implementan dentro de una clase denominada OCola (Operaciones sobre Colas).

```
// Declaración de la clase OCola (*.h)
// clase.h
class OCola {
    private:
        pCola c;
    public:
        OCola();
        ~OCola();
        void setCola(pCola c);
        pCola getCola();
        void agregar(char x);
        void quitar();
        void imprimir();
        .
        .
        .
};
```

```
// Implementación de la clase OCola
// clase.cpp
OCola::OCola() {
    c = new Cola();
}
```

Este método es el constructor de la clase, la cual se encarga de crear y asignar una dirección de memoria a la cola **c**. Se debe invocar a este método cuando se crea una nueva instancia de la clase OPila.

```
OCola::~OCola() {
    delete(c);
}
```

Este método es el destructor de la clase, la cual se encarga de liberar la dirección de memoria de la cola **c** creada en el método constructor. Se debe invocar a este método al final de todas las instrucciones, es decir, cuando ya no se tenga que usar la cola **c**.

```
void OCola::setCola(pCola c) {
    this->c = c;
}
```

Este es un método mutador, la cual se encarga de liberar la dirección de memoria de la cola **c**, y posteriormente asignar a otra dirección de memoria que en este caso viene a ser la dirección de memoria del parámetro **c**.

```
pCola OCola::getCola() {
    return c;
}
```

Este es un método accesorio, la cual se encarga de devolver la dirección de memoria de la cola **p**, mediante el nombre del método. Es muy importante que el tipo de dato del método deba ser igual al del atributo.

```
void OCola::agregar(char x) {
    if (!c->llena())
        c->insertar(x);
    else
        cout<<"Error!!\nCola llena";
}
void OCola::quitar() {
    if (!c->vacia())
        cout<<"El elemento eliminado es: "<<c->eliminar();
    else
        cout<<"Error!!\nCola vacia";
}
void OCola::imprimir() {
    c->mostrar();
}
```

2.10. Ejemplos

Para implementar cualquier operación avanzada y para tener una compresión con una gran facilidad se debe utilizar gráficas, de esta manera se tendrá una mejor comprensión.

1. **Realizar un método para eliminar el primer determinado elemento que se encuentre.**

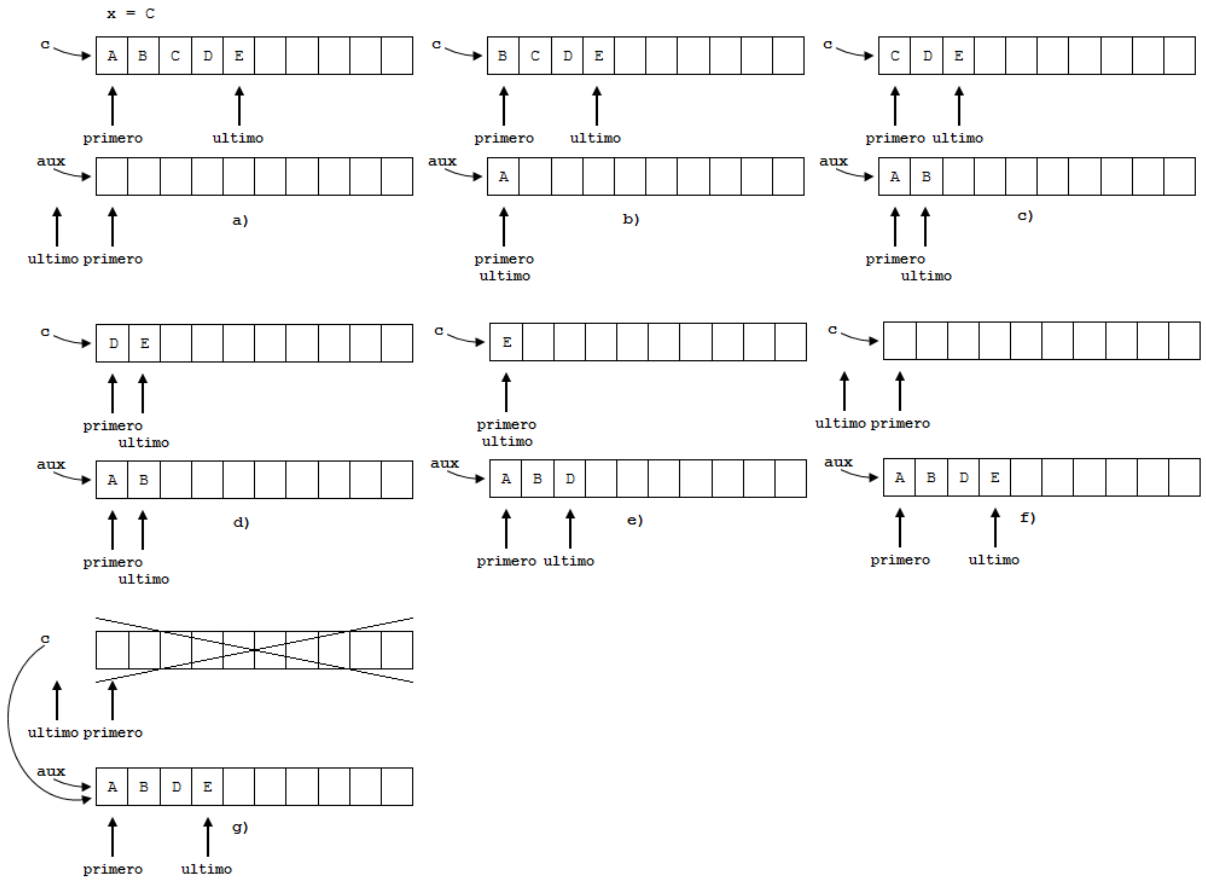


Figura 29. Eliminar un determinado elemento

Antes que nada se realiza la verificación correspondiente de la existencia de elementos dentro la cola, de ser así los pasos para cumplir con el objetivo de este método es (ver figura 29):

a) Crear una cola auxiliar **aux** ver figura 29 a), en las que se irán insertando todos los elementos eliminados de la cola principal **c**.
b) Mientras la cola **c** no esté vacía, eliminar los elementos de cola e insertar a la cola **aux** ver las figuras 29 b) y c).
c) En el caso de que se encuentre al elemento que se desee eliminar, no se debe insertar este elemento dentro la cola **aux**, como se muestra en la figura 29 d).

   Además, una vez que se encuentre al elemento a eliminar debe terminar el ciclo.
d) Comenzar con un nuevo ciclo, para terminar de eliminar todos los elementos de la cola **c** e insertarlos en la cola **c**, como se muestran en las figuras 29 e) y f).

e) Finalmente libere la dirección de memoria de la cola **c**, asigne la dirección de memoria de la cola **aux**, como se muestra en la figura 29 g).

A continuación se implementa el código correspondiente al método.

```
void OCola::eli_determinado(char x) {
    if (!c->vacia()) {
        pCola aux = new Cola();
        while (!c->vacia()) {
            char y = c->eliminar();
            if (y==x) break;
            aux->insertar(y);
        }
        while (!c->vacia())
            aux->insertar(c->eliminar());
        delete(c);
        c = aux;
    }
}
```

2. Realizar un método para insertar un elemento al principio de la cola.

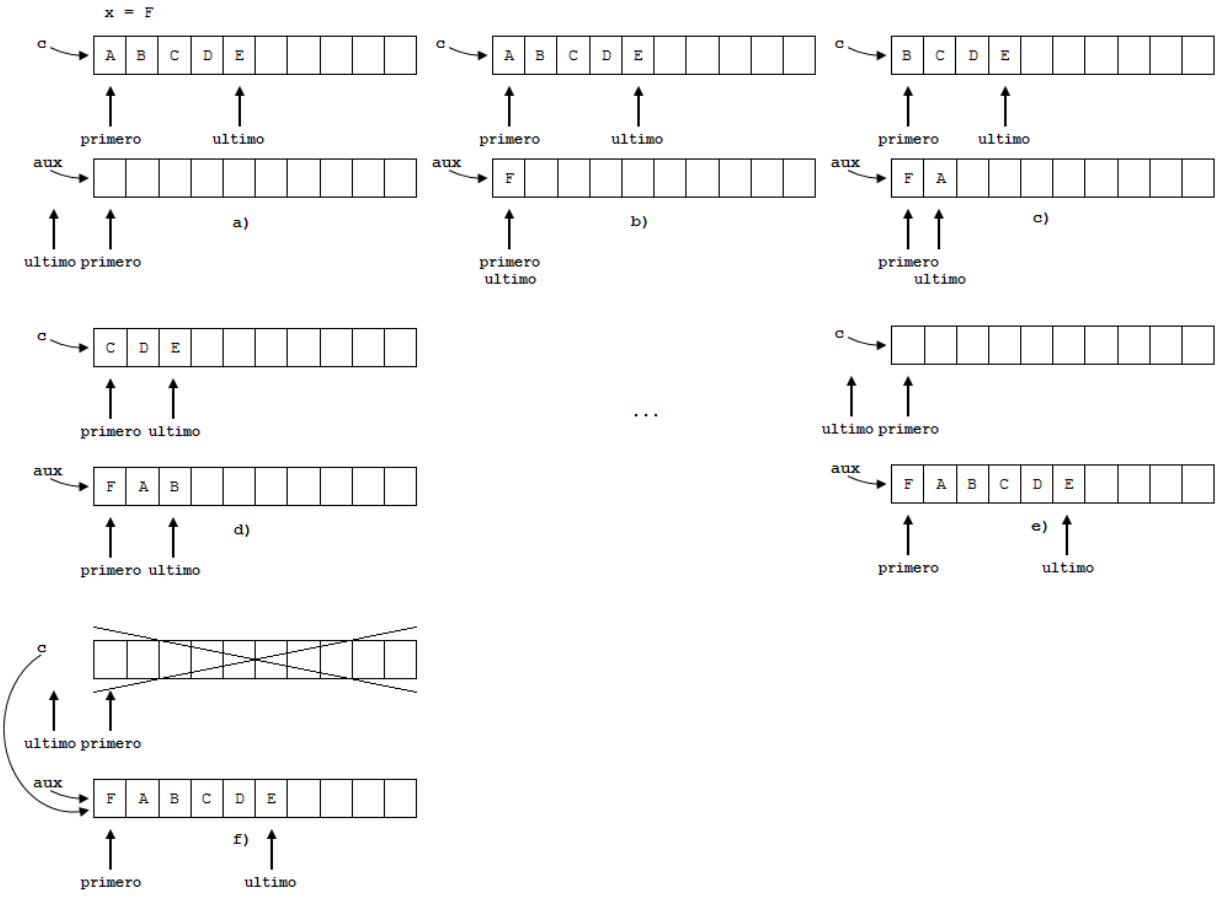


Figura 30. Insertar elemento al principio

Primeramente tiene que cerciorase que la cola no esté llena, solo de esta manera se insertar un nuevo elemento, de ser así los pasos a seguir son:

a) Crear una nueva cola **aux**, como se muestra en la figura 30 a).
b) Insertar el nuevo elemento en la cola **aux**, como se muestra la figura 30 b).
c) Eliminar todos los elementos de la **c** a la cola **aux**, es decir, la cola **c** debe estar vacía y estos elementos deben estar en la cola **aux**. Como se muestran en las figuras 30 c), d) y e).
d) Una vez que quede vacía la cola **c**, todos los elementos se encuentran en la cola **aux**, además que el primer elemento de la cola **aux** es nuevo elemento que se pretende insertar.
   Finalmente libere memoria de la cola **c**, y asigne la dirección de memoria de la cola **aux**, como se muestra en la figura 30 f).

A continuación se implementa el código correspondiente al método.

```
void OCola::ins_principio(char x) {
    if (!c->llena()) {
        pCola aux = new Cola();
        aux->insertar(x);
        while (!c->vacia())
            aux->insertar(c->eliminar());
        delete(c);
        c = aux;
    }
}
```

3. Realizar un método para insertar un determinado elemento al lado izquierdo del elemento que sea igual al elemento, el caso de no encontrarse en la cola insertar al final.

Al momento de analizar el objetivo del método es insertar un elemento a la cola, entonces lo primero que se debe verificar es que existe espacio por lo menos para uno en la cola, es decir, que la cola no esté vacía. Por tanto los pasos a seguir son:

a) Crear una nueva cola **aux**, como una cola auxiliar como se muestra en la figura 31 a).
b) Eliminar todos los elementos de la cola **c** hasta que quede vacía o se encuentre el elemento en esta cola e ir insertando estos elementos en la cola **aux**, como se muestra en la figura 31 b).
c) En el caso de que se encuentre el elemento a insertar en la cola **c**, primero debe insertar el elemento nuevo, posteriormente insertar el elemento eliminado a la cola **aux**, como se muestra en figura 31 c), Caso contrario se debe insertar al final de cola **aux**.
d) Eliminar el resto de los elementos de la cola **c** e insertar a la cola **aux**, como se muestra en las figuras 31 d), e) y f).

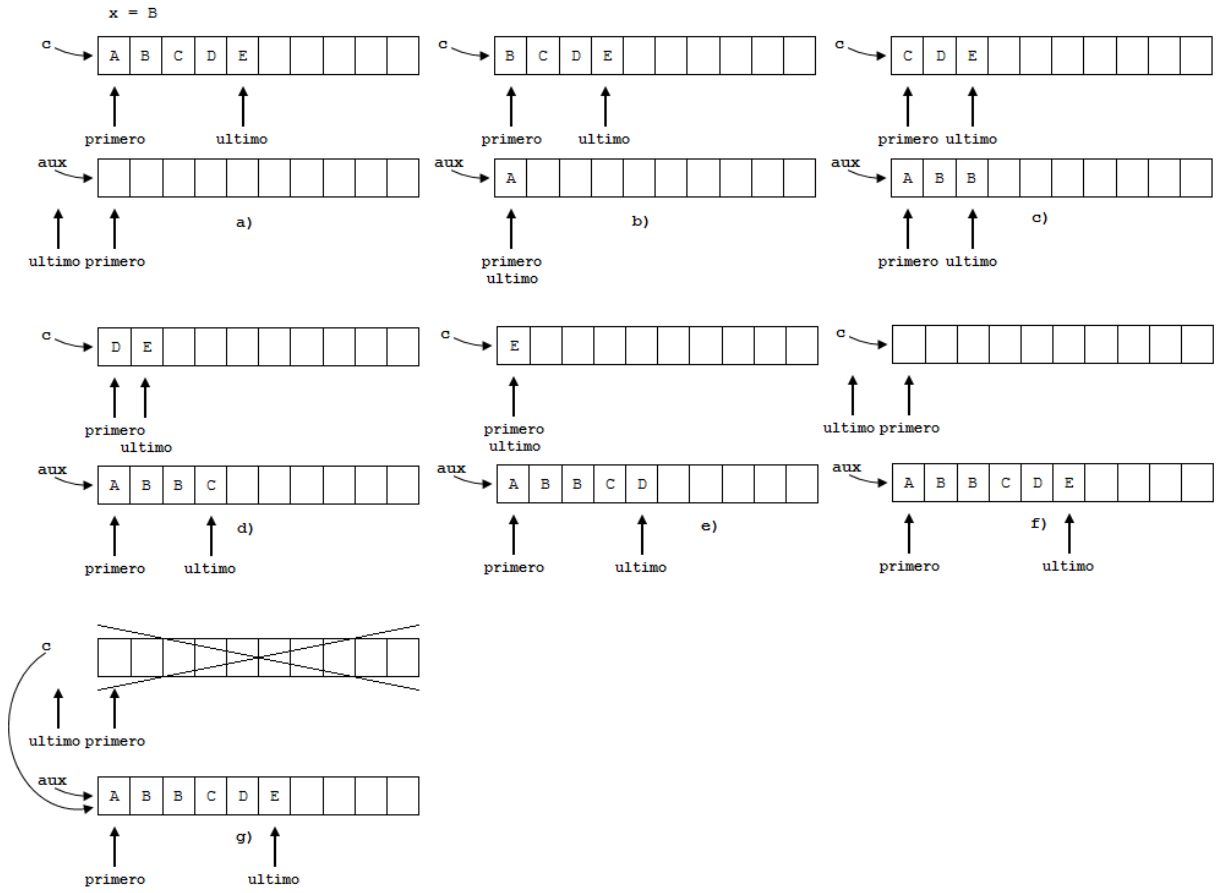


Figura 31. Insertar un elemento igual a algún elemento

e) Finalmente, libere memoria de la cola **c** y asigne la dirección de memoria a la cola **aux**, como se muestra en la figura 31 g).

A continuación se implementa el código correspondiente al método

```
void OCola::ins_izquierdo(char x) {
    if (!c->llena()) {
        pCola aux = new Cola();
        bool insertado = false;
        while (!c->vacia() && !insertado) {
            char y = c->eliminar();
            if (x==y) {
                aux->insertar(x);
                insertado = !insertado;
            }
            aux->insertar(y);
        }
        while (!c->vacia())
            aux->insertar(c->eliminar());
        if (!insertado)
            aux->insertar(x);
        delete(c);
        c = aux;
    }
}
```

4. Realizar un método para eliminar los elementos idénticos a un determinado elemento excepto el primero que se encuentre en la cola.

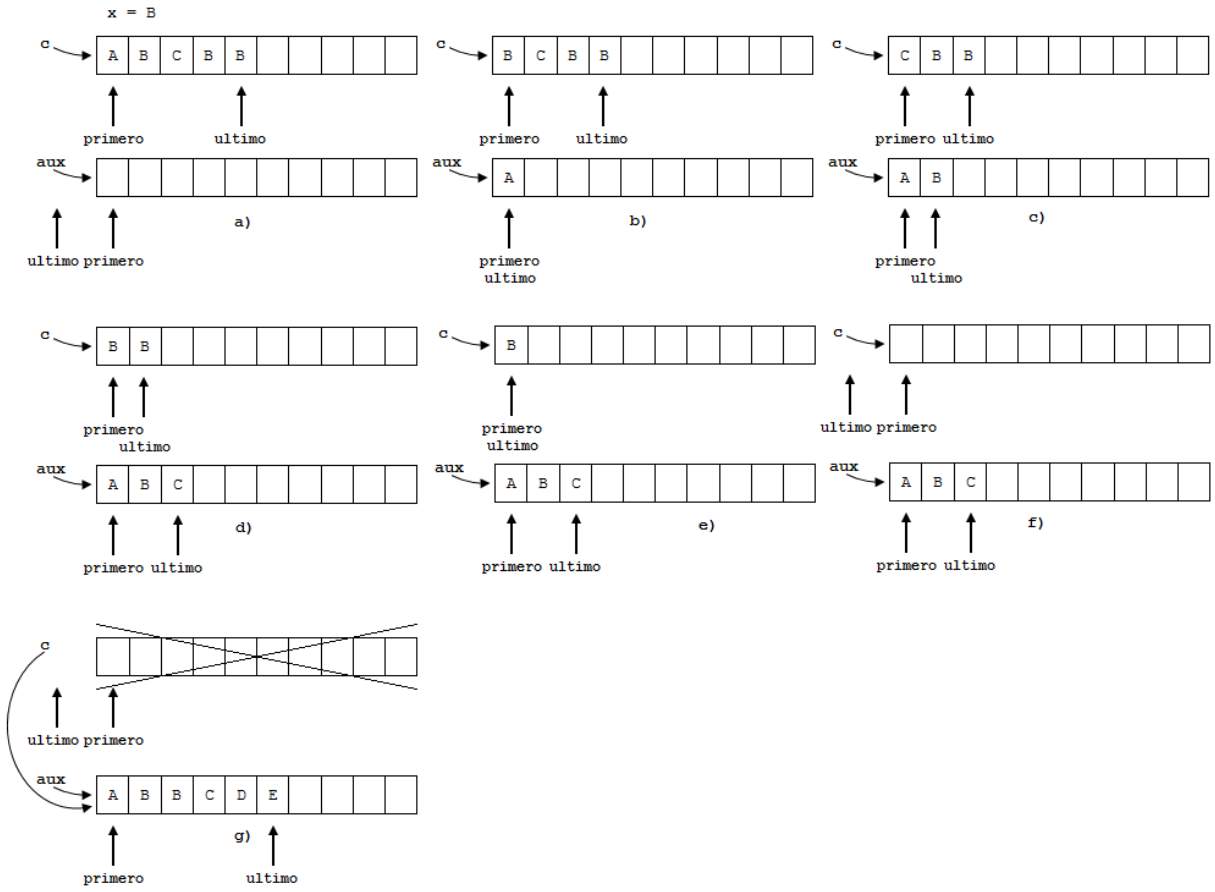


Figura 32. Eliminar elementos idénticos excepto el primero

Como el objetivo principal del método es eliminar elementos iguales a un determinado elemento pero sin eliminar el primero que sea igual al elemento, por tanto lo primero se debe comprobar que la cola **c** no esté vacía. Los pasos a seguir para cumplir el objetivo son:

a) Crear una cola **aux**, la cual es una cola auxiliar y permitirá manipular la cola **c** como se muestra en la figura 32 a).

b) Ir eliminando todos los elementos de la cola **c** e ir insertando las mismas hasta que quede vacía, como se muestra en la figura 32 b).

   A medida que se vaya eliminando los elementos de la cola **c**, se va comprobando con el elemento que se quiere eliminar, si es el primer elemento igual al quiere eliminarse entonces se debe insertar a la cola **aux** como se muestra en la figura 32 c).

c) En el caso de que ya sea el segundo, tercero u otro entonces no se debe insertar en la cola **aux** como se muestra en la figura 32 d).

Para verificar el número de veces que se ha encontrado el elemento a eliminar se contar la misma y así se determina el número de veces que se repite el elemento.

d) Siga con los anteriores pasos hasta que cola **c** quede completamente vacía como se muestra en las figuras 32 e) y f).

e) Finalmente libere la dirección de memoria de la cola **c,** y asigne la dirección de memoria de la cola **aux.**

A continuación se implementa el código correspondiente al método

```
void OCola::eli_primero(char x) {
    if (!c->vacia()) {
        short primero = 0;
        pCola aux = new Cola();
        while (!c->vacia()) {
            char y = c->eliminar();
            if (x==y) {
                if (primero==0) {
                    aux->insertar(y);
                    primero++;
                }
            }
            else
                aux->insertar(y);
        }
        delete(c);
        c = aux;
    }
}
```

2.11. Ejercicios

1. Realizar un método para insertar el elemento que se encuentra al principio de la cola sin eliminar la misma.
2. Realizar un método para eliminar el último elemento de la cola.
3. Realizar un método para insertar un elemento en forma ordenada descendentemente
4. Realizar un método para determinar el número de veces que se repite un determinado elemento.
5. Realizar un método para determinar el número de grupos de existe en una cola, para formar el grupo los elementos deben ser consecutivos sin importar que estos se repitan.
6. Realizar un método para eliminar elementos repetidos no consecutivos.
7. Realizar un método para eliminar elementos repetidos consecutivos.
8. Suponiendo que el campo información de una cola contiene solo caracteres numéricos. Realizar un método para calcular el promedio de la cola.

9. Suponiendo que el campo información de una cola contiene solo caracteres numéricos y el valor de cada campo información representa el tiempo en minutos. Realizar un método para calcular el total de información que es transferida en Mb, sabiendo que la velocidad de transmisión es de 10Kb/seg.
10. Sabiendo que el campo información de una cola contiene solo caracteres numéricos y operadores matemáticos, por lo tanto representa una expresión matemática. Realizar un método para calcular el resultado de esta expresión en forma comercial.

TEMA III

LISTAS ENLAZADAS

3.1. Introducción

Hasta ahora se venido realizando estructuras estáticas de datos como es la pila y cola, pese a crear instancias dinámicas esto no implica a modificar el tamaño de la pila o cola siempre el tamaño de ambos será la misma haciendo que no se pueda modificar en tiempo de ejecución.

Sin embargo una de las aplicaciones más interesante y potentes de la memoria dinámica y de los punteros son, sin duda, las estructuras dinámicas de datos.

En muchas ocasiones se necesitan estructuras que puedan cambiar de tamaño durante la ejecución del programa. Claro que se puede crear *arrays* dinámicos, pero una vez creados, tu tamaño también será fijo, y para hacer que crezcan o disminuyan de tamaño, se debe reconstruir desde el principio.

Las estructuras dinámicas nos permiten crear estructuras de datos que se adapten a las necesidades reales a las que suelen enfrentarse los programas. Pero no sólo eso sino también permitirán crear estructuras de datos muy flexibles, ya sea en cuanto al orden, la estructura interna o las relaciones entre los elementos que las componen.

3.1. Definición

Es una estructura de datos donde todos los elementos que forma parte de la lista tienen el mismo tipo de dato (ver tabla 1), y puede crecer o decrecer en tiempo de ejecución según las necesidades del usuario, que estará formada por una secuencia de elementos, donde cada uno de ellos va seguido de otro o de ninguno en el caso de ser el primero de la lista.

Esta secuencia de elementos llamados a partir de ahora nodos, se puede definir a una lista como un conjunto de nodos debidamente enlazados unos con otros, es decir, que estos nodos deben estar siempre unidos (enlazados).

3.2. Representación gráfica

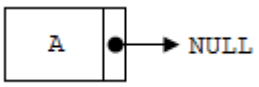


Figura 33. Nodo

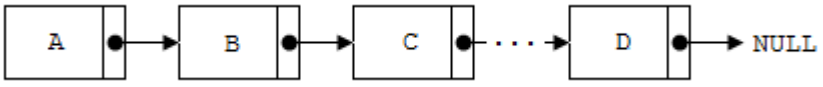


Figura 34. Lista enlazada

La figura 33 representa a un nodo la cual contiene información de la estructura de datos, esta información puede admitir cualquier tipo de dato,

además tiene enlaces las cuales permiten unirse o enlazarse con el resto de los nodos como se muestra en la figura 34. Al conjunto de nodos enlazados se denomina a partir de ahora listas enlazadas.

3.3. Tipos de listas

Dentro de los tipos de listas enlazadas existen cuatro diferentes las cuales son:

3.3.1. Listas simplemente enlazadas (LSE)

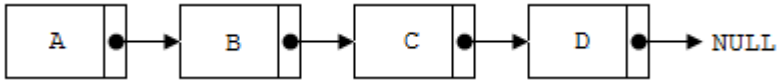


Figura 35. Lista simplemente enlazada

Como se puede observar en la figura 35, estos nodos están enlazados unos con otros, cada uno de estos nodos contiene solo un enlace, donde el enlace del último nodo de la lista apunta a nulo, es decir, que no apunta a nada.

3.3.2. Listas simplemente enlazadas circulares (LSEC)

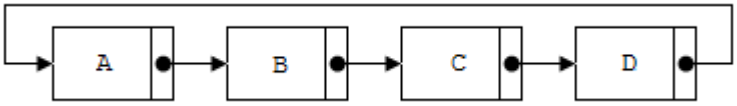


Figura 36. Lista simplemente enlazada circular

Se puede observar en la figura 36, que esta lista es similar a las listas simples enlaces con la diferencia de que el enlace del último nodo apunta al primer elemento.

3.3.3. Listas doblemente enlazadas (LDE)

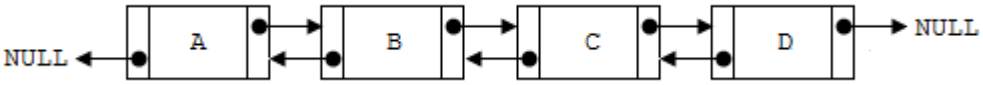


Figura 37. Lista doblemente enlazada

Los nodos de esta lista tienen dos enlaces, uno que apunta al nodo anterior y otro que apunta al nodo siguiente, además se puede observar que el enlace que apunta al siguiente nodo del último nodo de la lista apunta a nulo de igual forma el enlace anterior del primer nodo de la lista también apunta a nulo, es decir, que no apuntan a nada.

3.3.4. Listas doblemente enlazadas circulares (LDEC)

En la figura 38 se observa una lista doblemente circular que es similar a las listas doblemente enlazadas (ver figura 37), con la diferencia que el

enlace siguiente del último nodo de la lista apunta al primer nodo de la lista y el enlace anterior del primer nodo de la lista apunta al último nodo de la lista.

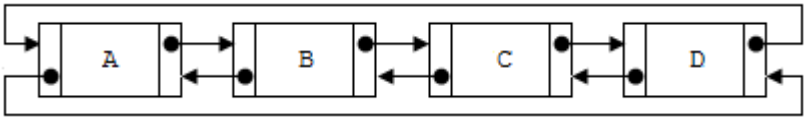


Figura 38. Lista doblemente enlazada circular

3.4. Partes de un Nodo

Las partes de un nodo depende mucho al tipo de lista que se está aplicando o usando para implementar. Sin embargo solo existe dos tipos de nodos las cuales son nodos de simple enlace que también forma parte de las listas simplemente enlazadas circulares, y los nodos de doble enlace que también forma parte de las listas doblemente enlazadas circulares.

3.4.1. Nodo Simple Enlace

Las partes de un nodo simple enlace se pueden observar en la figura 39 las cuales son:

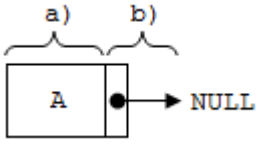


Figura 39. Nodo simple enlace

a) Campo información, la cual es el valor del elemento del nodo que puede tomar cualquier tipo de dato ya sean primitivas o definidas por el usuario.
b) Enlace, es un puntero que permite enlazar con los demás nodos, el tipo de dato de este puntero es de tipo nodo simple.

A partir de este nodo simple enlace se establecen las listas simplemente enlazadas y/o listas simplemente enlazadas circulares, como se muestran en las figuras 35 y 36 respectivamente.

3.4.2. Nodo Doble enlace

Las partes de un nodo doble enlace se pueden observar en la figura 40 las cuales son:

a) Campo información, la cual es el valor del elemento del nodo que puede tomar cualquier tipo de dato ya sean primitivas o definidas por el usuario.

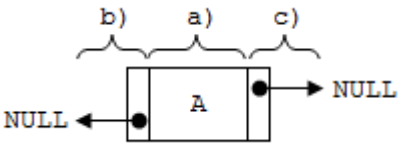


Figura 40. Nodo doble enlace

b) Enlace anterior, es un puntero que permite enlazar los nodos que se encuentran por detrás del nodo, es decir, enlaza a los nodos izquierdos. El tipo de dato de este enlace es de tipo nodo doble.
c) Enlace siguiente, es un puntero que permite enlazar los nodos que se encuentren por delante del nodo, es decir, enlaza a los nodos derechos. El tipo de dato de este enlace es de tipo nodo doble.

A partir de este nodo doble se establecen las listas doblemente enlazadas y/o listas doblemente enlazadas circulares, como se muestran en las figuras 37 y 38 respectivamente.

3.5. Estructura de datos de un nodo

Cada tipo de nodo tiene su estructura de datos, esta estructura de datos es una abstracción de los diferentes tipos de nodos. Por tanto se tiene dos estructuras de datos, es decir, una para un nodo simple (también sirve para nodo simple circular) y otra para un nodo doble (también sirve para un nodo doble circular).

3.5.1. Nodo simple

Según la figura 39 se tiene lo siguiente, también es la estructura de datos de una lista simplemente enlazada y de una lista simplemente enlazada circular.

```
// Declaración de cabecera
// clase.h
class NodoS {
    private:
        char info;
        NodoS *enlace;
    public:
        NodoS(char x);
        char getInfo();
        NodoS *getEnlace();
        void setEnlace(NodoS *N);
};
typedef NodoS *pNodoS;
```

```
// Implementación de cabecera
// clase.cpp
NodoS::NodoS(char x) {
    info = x;
    enlace = NULL;
}
char NodoS::getInfo() {
    return info;
}
NodoS *NodoS::getEnlace() {
    return enlace;
}
void NodoS::setEnlace(NodoS *N) {
    enlace = N;
}
```

3.5.2. Nodo doble

Según la figura 40 se tiene lo siguiente, también es la estructura de datos de una lista doblemente enlazada y de una lista doblemente enlazada circular

```
// Declaración de cabecera
// clase.h
class NodoD {
    private:
        char info;
        NodoD *siguiente;
        NodoD *anterior;
    public:
        NodoD(char x);
        char getInfo();
        NodoD *getSgte();
        NodoD *getAnte();
        void setSgte(NodoD *N);
        void setAnte(NodoD *N);
};
typedef NodoD *pNodoD;
```

```
// Implementación de cabecera
// clase.cpp
NodoD::NodoD(char x) {
    info = x;
    siguiente = NULL;
    anterior = NULL;
}
char NodoD::getInfo() {
    return info;
}
NodoD *NodoD::getSgte() {
    return siguiente;
}
NodoD *NodoD::getAnte() {
    return anterior;
}
void NodoD::setSgte(NodoD *N) {
    siguiente = N;
}
void NodoD::setAnte(NodoD *N) {
    anterior = N;
}
```

3.6. Operaciones avanzadas

Dentro de las operaciones avanzadas se tiene que ver cuatro diferentes tipos de lista, como en el caso de estudio del apartado 3.3., es decir, se implementará las listas simplemente enlazadas, listas simplemente enlazadas circulares, listas doblemente enlazada y listas doblemente enlazadas circulares.

Para cada una de estas operaciones se declarará e implementará una clase en los archivos cabecera (*.h) y recurso (*.cpp).

3.6.1. Operaciones de listas simplemente enlazadas

La declaración de la clase en el archivo cabecera (*.h) es la siguiente para realizar las diferentes operaciones sobre estas listas

```
class Operacion_S_E {
    private:
        pNodoS lista;
    public:
        Operacion_S_E();
        ~Operacion_S_E();
        void mostrar();
        void ins_delante(char x);
        void eli_delante();
        void ins_detras(char x);
```

```
        void eli_detras();
        ...
};
```

Y la implementación de los métodos en el archivo de recursos (*.cpp) de la clase Operacion_S_E se tiene

1. Constructor de la clase

Este método en la que se encarga de inicializar a la lista en nula, es decir, no existen nodos en la lista, como se muestra en la figura 41.

Figura 41. Inicializar lista en nulo

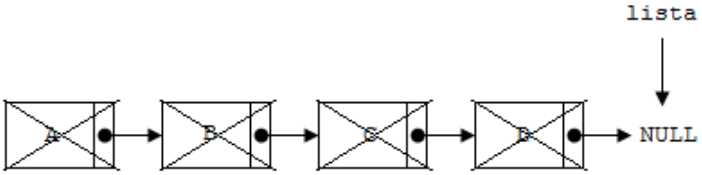


Figura 42. Eliminar todos nodos de una LSE

```
// Constructor de la clase
Operacion_S_E::Operacion_S_E() {
    lista = NULL;
}
```

2. Destructor de la clase

Este método es el se encarga de eliminar todos los nodos de la lista, donde la lista debe quedad en NULL, como se muestra en la figura 42.

Para implementar este método se tiene que recorrer nodo por nodo desde el primer nodo de la lista con el puntero **p** hasta que el puntero **p** quede nulo.

Es necesario trabajar con dos punteros **p** y **q**, en la que **p** se mueve por delante de **q**, y **q** permite liberar o eliminar el nodo. Finalmente se pone el puntero lista en nulo.

```
// Destructor de la clase
Operacion_S_E::~Operacion_S_E() {
    if (lista != NULL) {
        pNodoS p = lista, q = lista;
        while (p!=NULL) {
            q = p;
            p = p->getEnlace();
            delete(q);
        }
        lista = NULL;
    }
}
```

3. Método mostrar

Permite recorrer todos los nodos de la lista mostrando a la vez su información cada uno de los nodos de la lista.

```
// Método que permite mostrar los elementos de la lista
void Operacion_S_E::mostrar() {
    if (lista!=NULL) {
        pNodoS p = lista;
        while (p!=NULL) {
            cout<<p->getInfo()<<"\t";
            p = p->getEnlace();
        }
    }
}
```

4. Método insertar un nodo por delante

Para implementar este método lo primero que se tiene que realizar es crear un nodo al margen de que exista o no más nodos en la lista.

Estos casos se pueden ver en la figura 43, donde 43 a) es el caso cuando existen nodos en la lista y 43 b) pertenece cuando es el primer nodo de la lista.

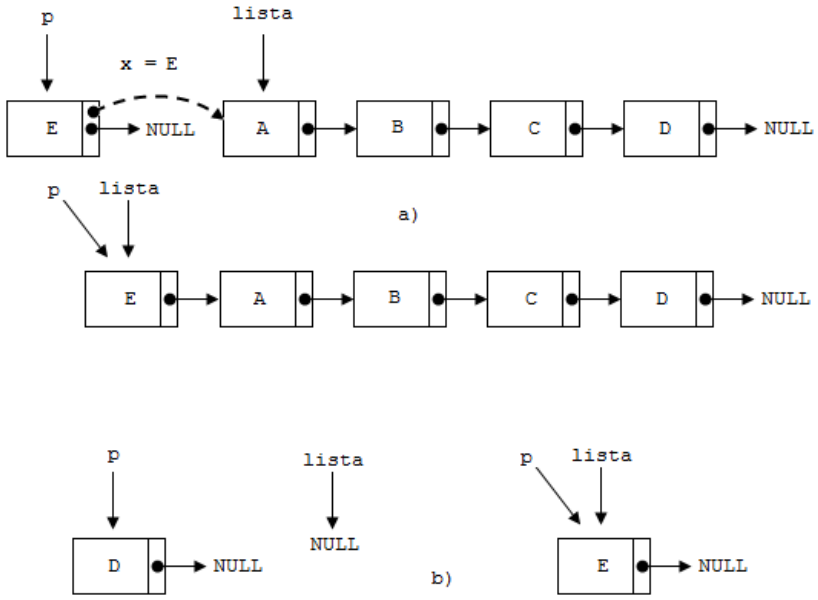


Figura 43. Inserción de un nodo por delante en una LSE

El código correspondiente de este método es

```
// Insertar un nodo por delante de la lista
void Operacion_S_E::ins_delante(char x) {
    pNodoS p = new NodoS(x);
    if (lista!=NULL)
        p->setEnlace(lista);
    lista = p;
}
```

5. Método eliminar el primer nodo de la lista

Lo primero que se debe tener en cuenta es que existan nodos en la lista, de ser así se puede continuar con el método.

Posteriormente se debe verificar si existen varios nodos en la lista de ser así ver la figura 44 a), caso contrario ver la figura 44 b).

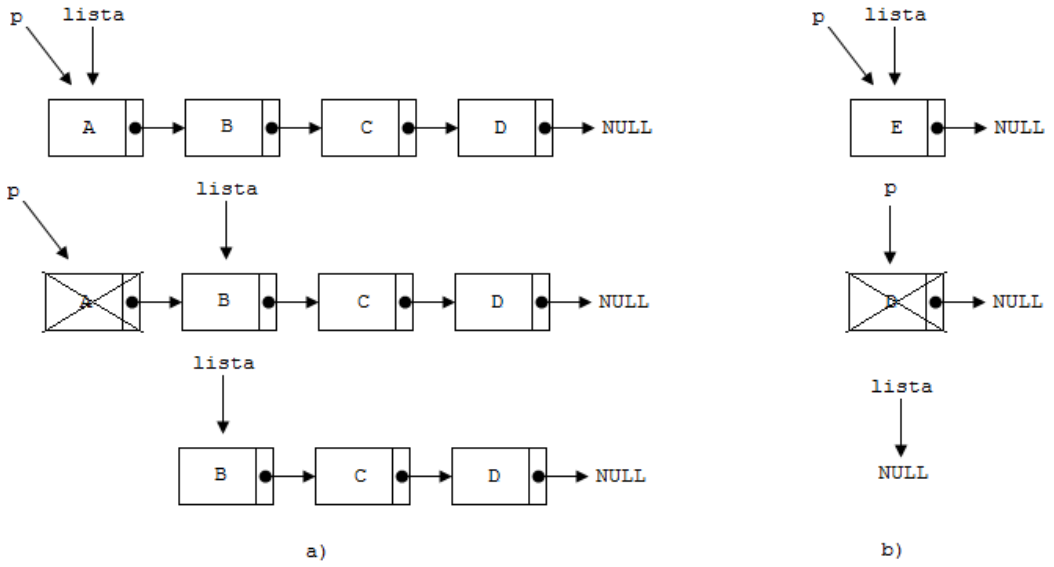


Figura 44. Eliminar el primer nodo en una LSE

La implementación de su código es de la siguiente manera

```
void Operacion_S_E::eli_delante() {
    if (lista!=NULL) {
        pNodoS p = lista;
        if (lista->getEnlace()!=NULL)
            lista = lista->getEnlace();
        else
            lista = NULL;
        delete(p);
    }
}
```

6. Método insertar un nodo al final de la lista

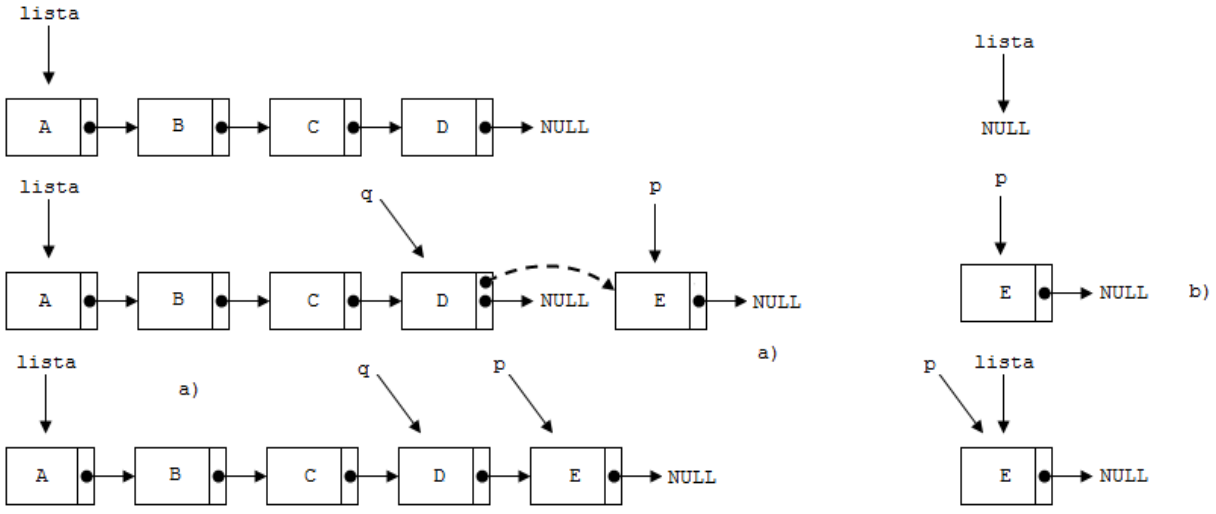


Figura 45. Insertar un nodo por detrás en una LSE

Primero se debe crear un muy al margen de que existan ó no nodos en la lista.

En el caso de que existan nodos en la lista, se debe obtener el último nodo de la lista como se muestra en la figura 45 a) y posteriormente actualizar los enlaces; caso contrario es el primer nodo de la lista como se muestra en la figura 45 b).

El código correspondiente al método es

```
void Operacion_S_E::ins_detras(char x) {
    pNodoS p = new NodoS(x), q = NULL;
    if (lista!=NULL) {
        q = lista;
        while (q->getEnlace()!=NULL)
            q = q->getEnlace();
        q->setEnlace(p);
    }
    else
        lista = p;
}
```

7. Método eliminar el último nodo

Al momento de implementar este método se tiene que verificar que existan nodos en la lista, caso contrario no se puede eliminar ningún nodo.

Una vez comprobada si tiene que obtener el último nodo de la lista y actualizar los enlaces correspondientes, en el caso de que sea el último nodo de la lista, la lista debe quedar en nulo como se muestra en la figura 46.

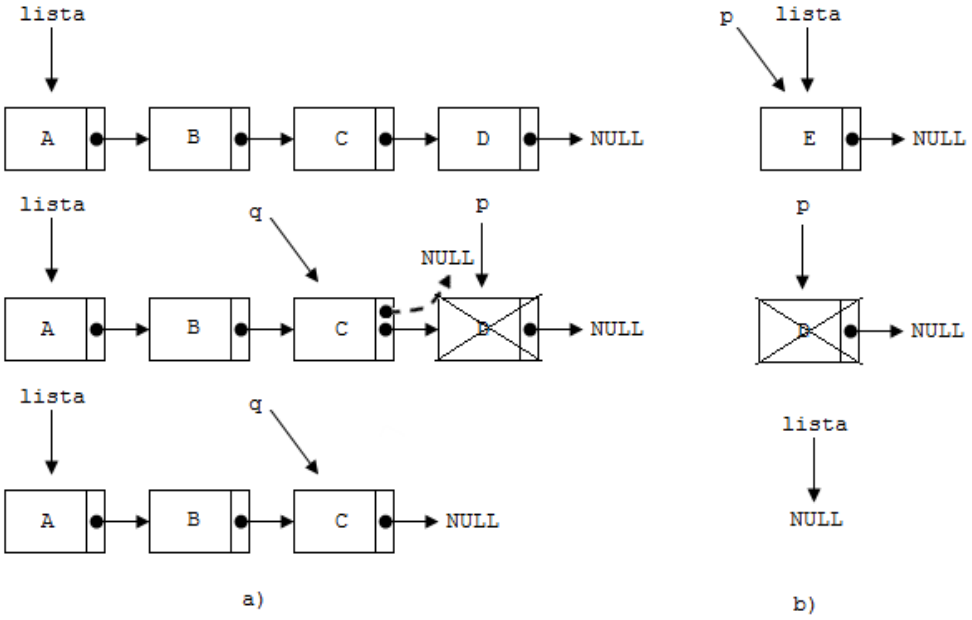


Figura 46. Eliminar el último nodo en una LSE

```
void Operacion_S_E::eli_detras() {
    if (lista!=NULL) {
        pNodoS p = lista, q = NULL;
        while (p->getEnlace()!=NULL) {
            q = p;
            p = p->getEnlace();
        }
        if (q==NULL)
            lista = NULL;
        else
            q->setEnlace(NULL);
        delete(p);
    }
}
```

3.6.2. Operaciones de listas simplemente enlazadas circulares

La estructura de datos que se utiliza para este tipo de lista, es la misma que se utiliza en las listas simplemente enlazadas, por tanto

La declaración de la clase Operacion_S_E_C en el archivo cabecera (*.h) es la siguiente para realizar las diferentes operaciones.

```
class Operacion_S_E_C {
    private:
        pNodoS lista;
    public:
        Operacion_S_E_C();
        ~Operacion_S_E_C();
        void mostrar();
        void ins_delante(char x);
        void eli_delante();
        void ins_detras(char x);
        void eli_detras();
        ...
};
```

Y la implementación de los métodos en el archivo de recursos (*.cpp) de la clase Operacion_S_E_C es la siguiente

1. Constructor de la clase

El constructor permite inicializar la lista en nulo quedando la primera vez la lista sin ningún nodo como se muestra en la figura 41.

```
Operacion_S_E_C::Operacion_S_E_C() {
    lista = NULL;
}
```

2. Destructor de la clase

Permite eliminar todos los nodos de la lista, quedando al final la lista en nulo como se muestra en la figura 47.

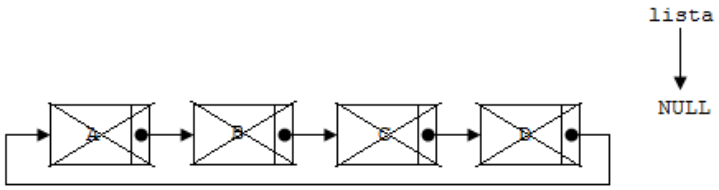


Figura 47. Eliminar todos los nodos de una LSEC

Y código de implementación correspondiente es el siguiente

```
Operacion_S_E_C::~Operacion_S_E_C() {
    if (lista!=NULL) {
        pNodoS p = lista->getEnlace(), q = NULL;
        while (p!=lista) {
            q = p;
            p = p->getEnlace();
            delete(q);
        }
        lista = NULL;
        delete(p);
    }
}
```

3. Método para mostrar todos los nodos de la lista

Este método permite mostrar la información que contiene cada uno de los nodos de la lista.

Su código de implementación es el siguiente

```
void Operacion_S_E_C::mostrar() {
    if (lista!=NULL) {
        pNodoS p = lista;
        do {
            cout<<p->getInfo()<<"  ";
            p = p->getEnlace();
        } while (p!=lista);
    }
}
```

4. Método para insertar un nodo por delante

Para poder insertar un nodo por delante a una lista simplemente enlazada circular, primero se debe crear el nodo al margen de que existan nodos en la lista o no.

Posteriormente se debe verificar si es el primer nodo que se inserta (ver figura 48 b)) o no (ver figura 48 a)).

A la vez se debe obtener el último nodo de la lista siempre y cuando la lista no sea nula.

Su código correspondiente para insertar un nodo por delante es

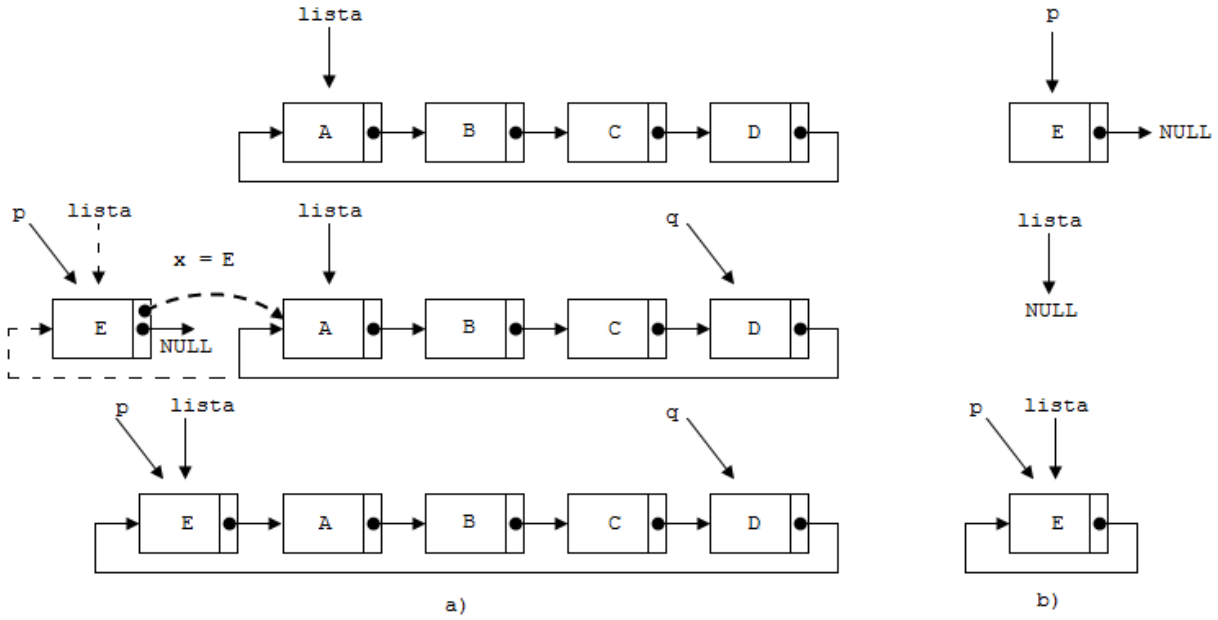


Figura 48. Insertar un nodo por delante en una LSEC

```
void Operacion_S_E_C::ins_delante(char x) {
    pNodoS p = new NodoS(x), q = NULL;
    if (lista!=NULL) {
        q = lista;
        while (q->getEnlace()!=lista)
            q = q->getEnlace();
        p->setEnlace(lista);
        q->setEnlace(p);
    }
    else
        p->setEnlace(p);
    lista = p;
}
```

5. Método para eliminar un nodo por delante

Para eliminar un nodo de la lista se debe tener en cuenta que debe existir nodos en la lista, para lo cual primero se debe ubicar un puntero en el último nodo de la lista y otro puntero en el antepenúltimo nodo para actualizar los enlaces (ver figura 49 a)). Si es el último y único nodo la lista debe quedar nulo (ver figura 49 b)).

```
void Operacion_S_E_C::eli_delante() {
    if (lista!=NULL) {
        pNodoS p = lista, q = lista;
        while (p->getEnlace()!=lista)
            p = p->getEnlace();
        if (p==lista) lista = NULL;
        else {
            p->setEnlace(lista->getEnlace());
            lista = lista->getEnlace();
        }
        delete(q);
```

```
    }
}
```

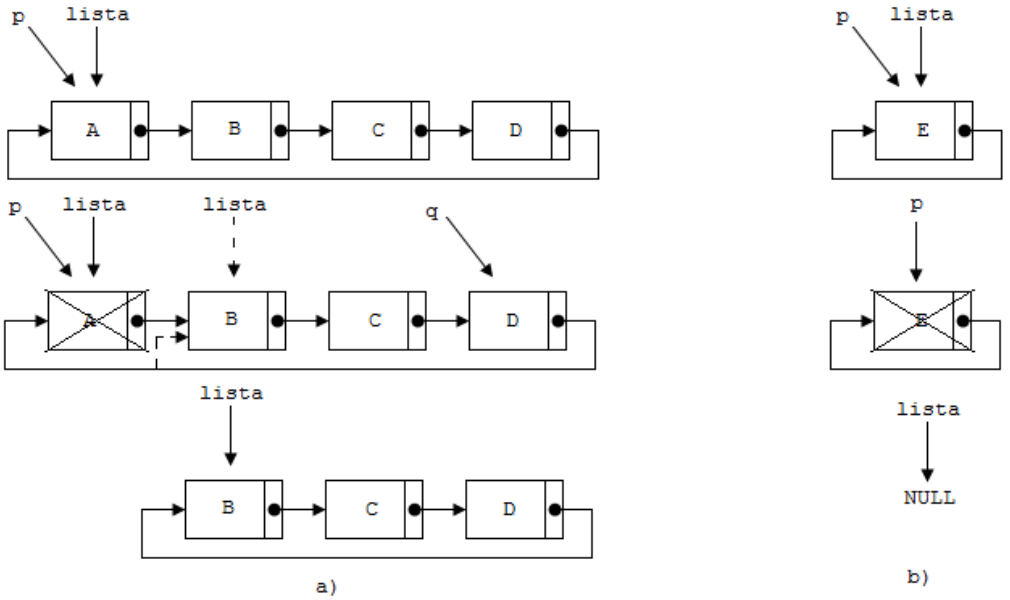


Figura 49. Eliminar un nodo por delante en una LSEC

6. Método para insertar un nodo por detrás

Para insertar un nodo por detrás primero tiene que crear un nuevo nodo, y verificar si existen nodos en lista, en el caso de existir se debe obtener el último nodo de la lista (ver figura 50 a)); al no existir ningún nodo es el primero (ver figura b)).

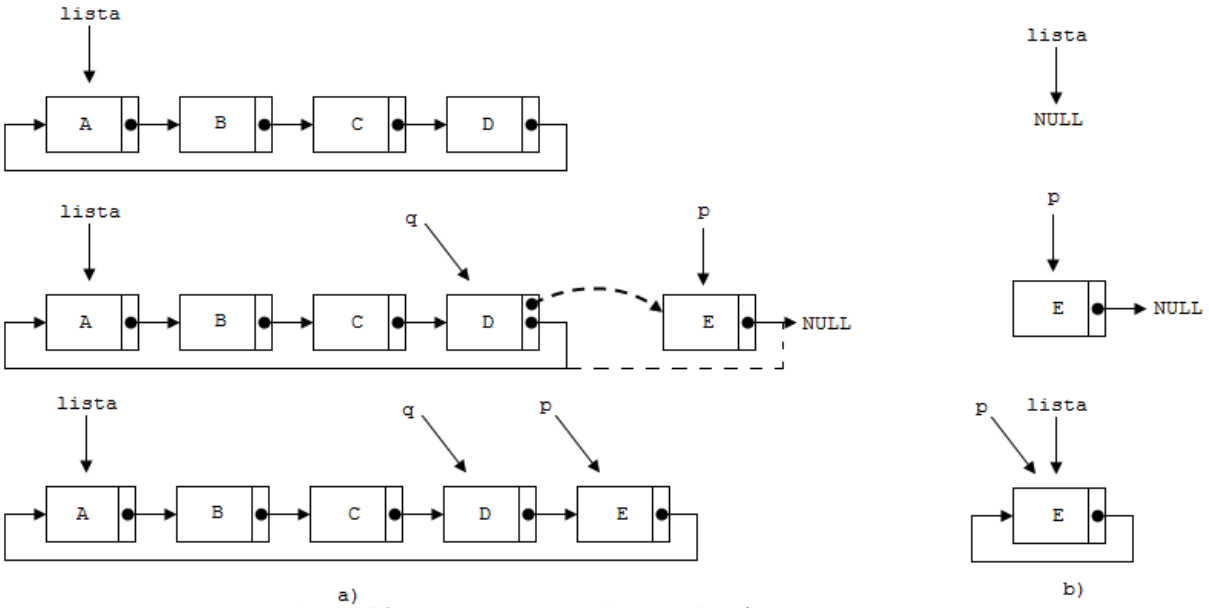


Figura 50. Insertar un nodo por detrás en una LSEC

Su código correspondiente es

```
void Operacion_S_E_C::ins_detras(char x) {
    pNodoS p = new NodoS(x), q = NULL;
```

```
    if (lista!=NULL) {
        q = lista;
        while (q->getEnlace()!=lista)
            q = q->getEnlace();
        p->setEnlace(lista);
        q->setEnlace(p);
    }
    else {
        p->setEnlace(p);
        lista = p;
    }
}
```

7. Método para eliminar el último nodo

Para eliminar el último nodo de una lista se debe verificar que existan nodos en la lista y se debe obtener el último y penúltimo nodo de la lista para actualizar posteriormente (ver figura 51 a)), en el caso de que sea el último nodo de la lista y único entonces la lista debe quedar en nulo (ver figura 51 b)).

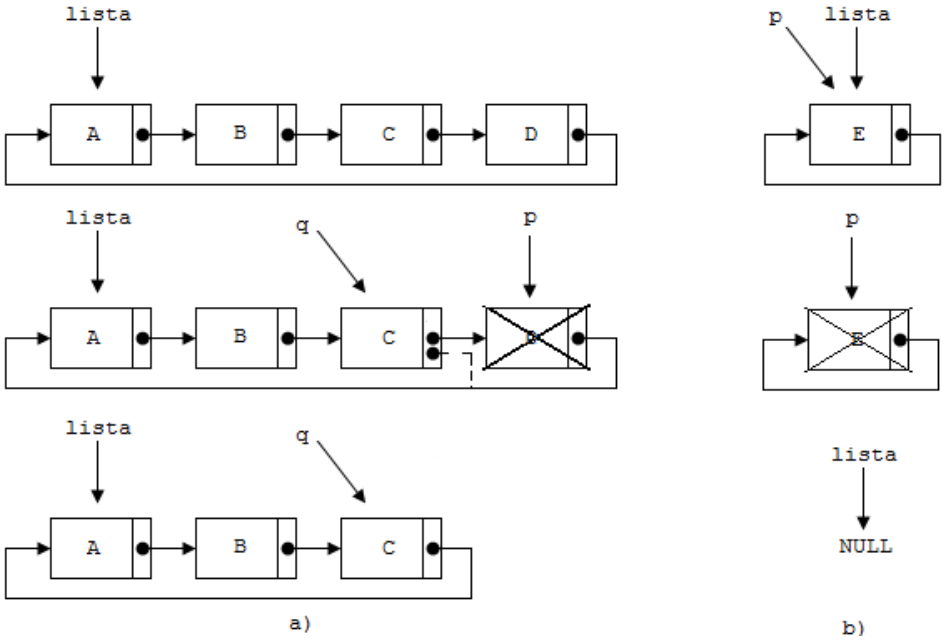


Figura 51. Insertar un nodo por delante en una LSEC

```
void Operacion_S_E_C::eli_detras() {
    if (lista!=NULL) {
        pNodoS p = lista, q = NULL;
        while (p->getEnlace()!=lista) {
            q = p;
            p = p->getEnlace();
        }
        if (p==lista) lista = NULL;
        else q->setEnlace(lista);
        delete(p);
    }
```

```
}
```

3.6.3. Operaciones de listas doblemente enlazadas

La estructura de datos de una lista doblemente enlazada es la misma que la de un nodo doble (ver apartado 4.5.2.).

La declaración de la clase Operacion_D_E en el archivo cabecera (*.h) es la siguiente para realizar las diferentes operaciones sobre estas listas

```
class Operacion_D_E {
    private:
        pNodoD lista;
    public:
        Operacion_D_E();
        ~Operacion_D_E();
        void mostrar();
        void ins_delante(char x);
        void eli_delante();
        void ins_detras(char x);
        void eli_detras();
        ...
};
```

Y la implementación de los métodos en el archivo de recursos (*.cpp) de la clase Operacion_D_E es la siguiente

1. Constructor de la clase

Este método permite inicializar a lista en nulo como se muestra en la figura 41.

```
Operacion_D_E::Operacion_D_E() {
    lista = NULL;
}
```

2. **Destructor de la clase**

Permite eliminar todos los nodos de la lista como se muestra en la figura 52.

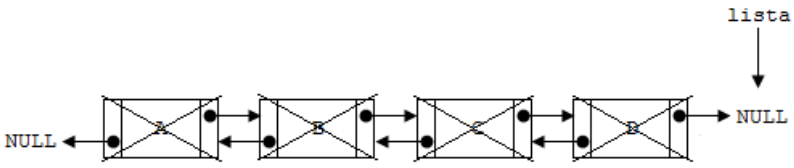


Figura 52. Elimina todos los nodos de una LDE

```
Operacion_D_E::~Operacion_D_E() {
    if (lista!=NULL) {
        pNodoD p = lista, q = NULL;
        while (p!=NULL) {
            q = p;
            p = p->getSgte();
            delete(q);
```

```
        }
        lista = NULL;
    }
}
```

3. Método para mostrar todos los nodos de la lista

Este método permite mostrar todos los elementos de una lista

```
void Operacion_D_E::mostrar() {
    pNodoD p = lista;
    while (p!=NULL) {
        cout<<p->getInfo()<<"  ";
        p = p->getSgte();
    }
}
```

4. Método para insertar un nodo por delate

Al momento de insertar un nuevo nodo por delante se debe crear el nodo posteriormente y verificar que existan nodos en la lista.

En el caso de que existan nodos en la lista actualizar los enlaces anteriores de lista para que apunte al nuevo nodo, y el enlace siguiente del nuevo nodo apunte a lista (ver figura 53 a)).

Caso contrario es el primer nodo de la lista (ver figura 53 b)).

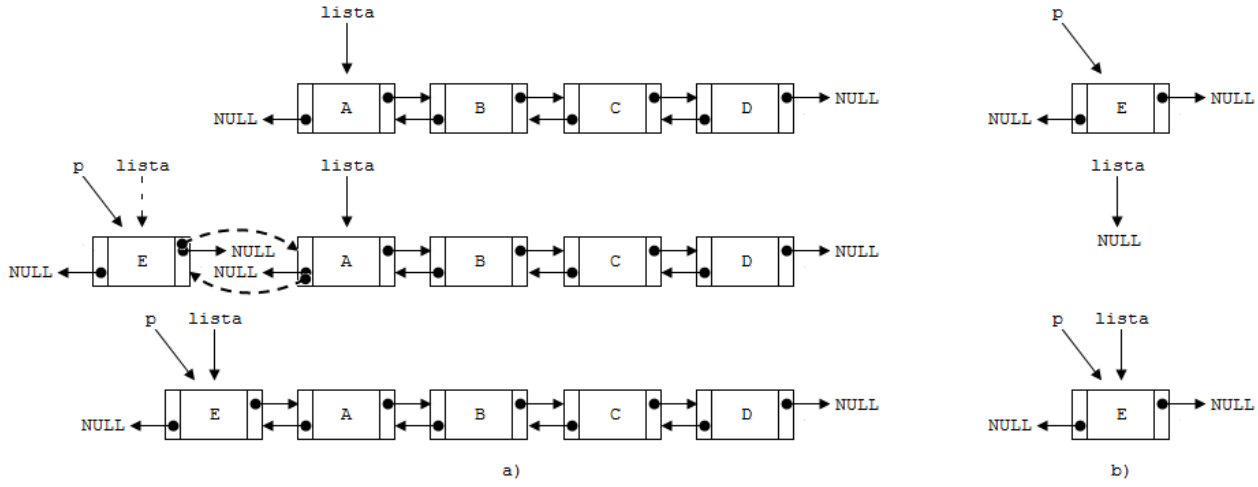


Figura 53. Insertar un nodo por delante en una LSE

```
void Operacion_D_E::ins_delante(char x) {
    pNodoD p = new NodoD(x);
    if (lista!=NULL) {
        p->setSgte(lista);
        lista->setAnte(p);
    }
    lista = p;
}
```

5. Método para eliminar un nodo por delante

Para eliminar un nodo por delante primero se debe verificar de que existan nodos en la lista, en la caso de existir nodos en la lista se debe dejar un puntero donde apunta lista y recorrer lista al siguiente (ver figura 54 a)) salvo a que sea el único nodo de la lista de ser así la lista debe quedar en nulo (ver figura 54 b)).

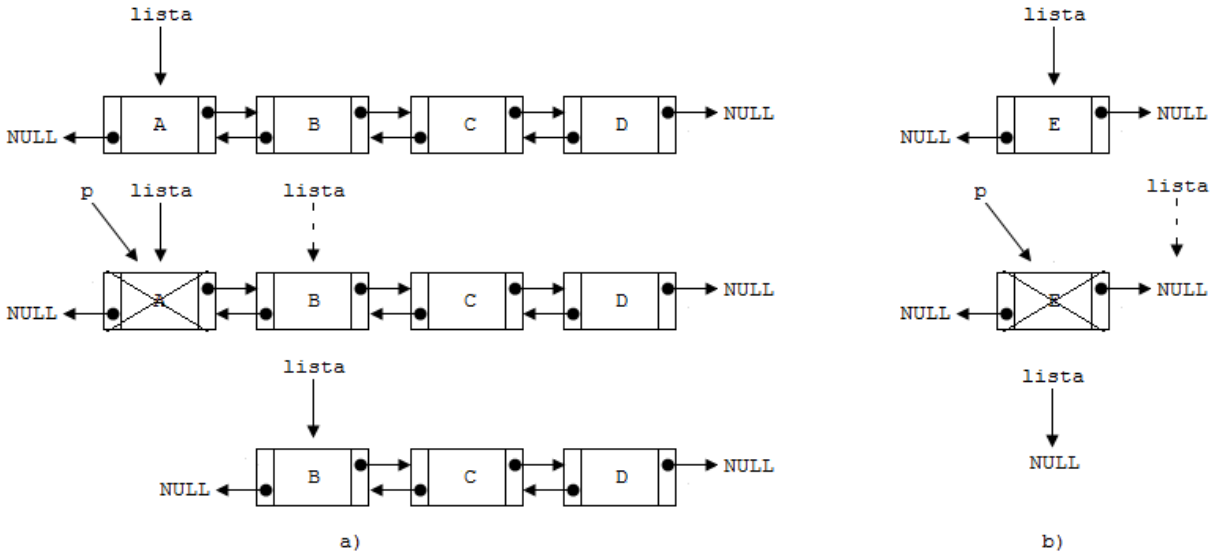


Figura 54. Eliminar un nodo por delante en una LDE

```
void Operacion_D_E::eli_delante() {
    if (lista!=NULL) {
        pNodoD p = lista;
        if (lista->getSgte()==NULL)
            lista = NULL;
        else
            lista = lista->getSgte();
        delete(p);
    }
}
```

6. Método para insertar un nodo al final de la lista

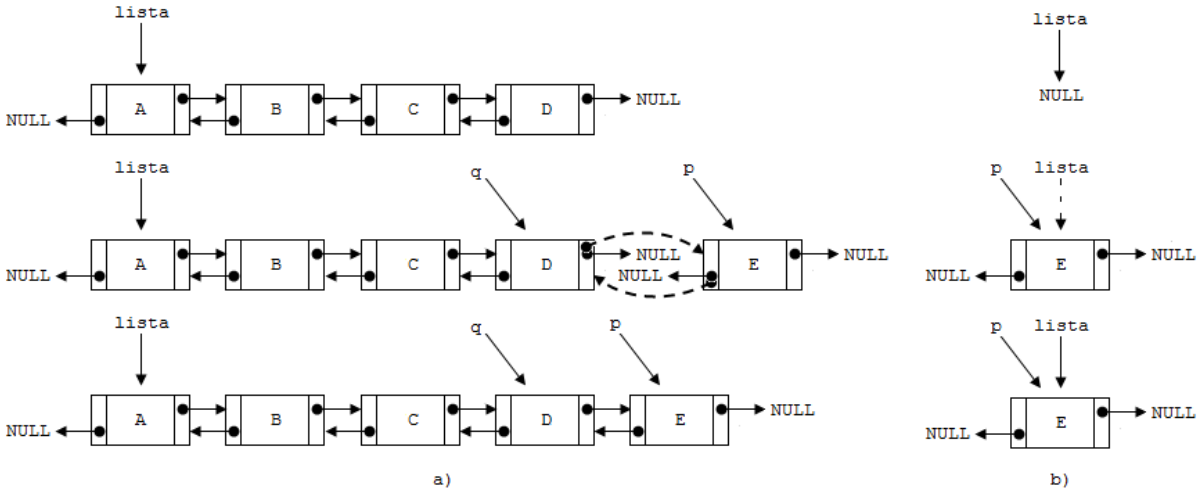


Figura 55. Insertar un nodo al último en una LDE

Para insertar un nodo al final de la lista debe crear un nuevo nodo, posteriormente verifique si existen nodos en la lista.

En el caso de existir nodos en la lista se debe obtener el último nodo de la lista, en la cual el enlace siguiente del último nodo debe apuntar al nuevo nodo y el enlace anterior del nuevo nodo debe apuntar al último nodo (ver figura 55 a)); caso contrario es el primer nodo de la lista (ver figura 55 b)).

```
void Operacion_D_E::ins_detras(char x) {
    pNodoD p = new NodoD(x), q = NULL;
    if (lista!=NULL) {
        q = lista;
        while (q->getSgte()!=NULL)
            q = q->getSgte();
        q->setSgte(p);
        p->setAnte(q);
    }
    else
        lista = p;
}
```

7. Método para eliminar nodos por detrás

Al momento de eliminar el último nodo de la lista primero se debe verificar que existan nodos en la lista.

En el caso de existir nodos en la lista se debe obtener el último y penúltimo nodo de la lista, donde el enlace siguiente del penúltimo nodo apunta a nulo (ver figura 56 a)); en el caso de que sea el último y único nodo la lista debe quedar en nulo (ver figura 56 b)).

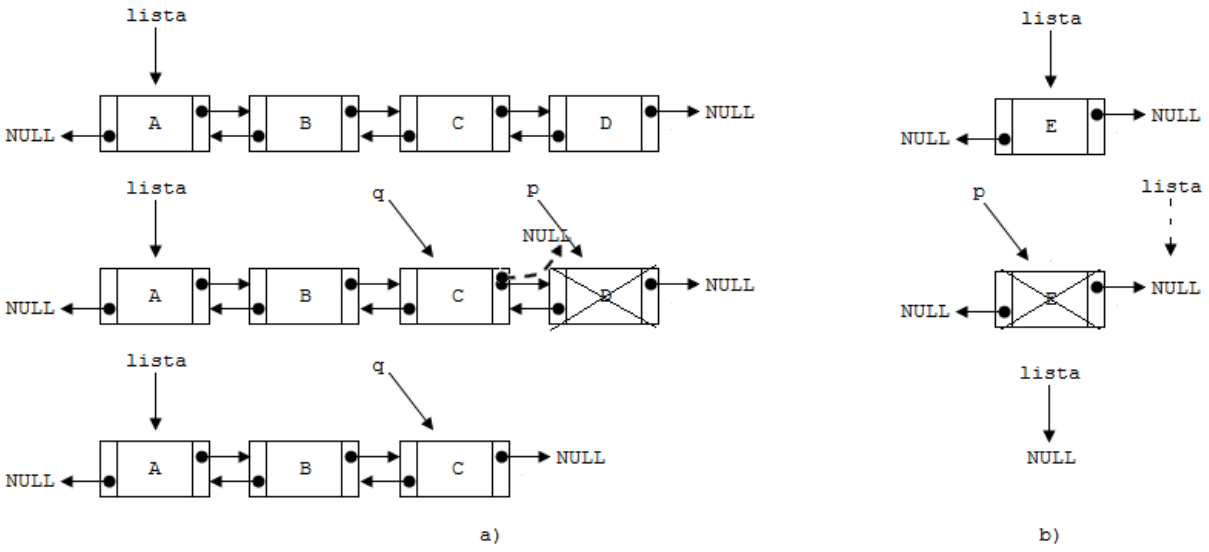


Figura 56. Eliminar el último nodo de una LDE

```
void Operacion_D_E::eli_detras() {
    if (lista!=NULL) {
        pNodoD p = lista, q = NULL;
        while (p->getSgte()!=NULL)
            p = p->getSgte();
        if (p==lista)
            lista = NULL;
        else {
            q = p->getAnte();
            q->setSgte(NULL);
        }
        delete(p);
    }
}
```

3.6.4. Operaciones de listas doblemente enlazadas circulares

Ya se tiene declarado la estructura de datos de una lista doblemente enlazada circular que es la misma de una lista doblemente enlazada circular que se hizo el estudio en el anterior apartado.

La declaración de la clase Operacion_D_E_C en el archivo cabecera (*.h) es la siguiente para realizar las diferentes operaciones sobre estas listas

```
class Operacion_D_E_C {
    private:
        pNodoD lista;
    public:
        Operacion_D_E_C();
        ~Operacion_D_E_C();
        void mostrar();
        void ins_delante(char x);
        void eli_delante();
        ...
};
```

Y la implementación de los métodos en el archivo de recursos (*.cpp) de la clase Operacion_D_E_C es la siguiente

1. Constructor

Es el método que permite inicializar la lista nulo (ver figura 41).

```
Operacion_D_E_C::Operacion_D_E_C() {
    lista = NULL;
}
```

2. Destructor

Permite eliminar todos los elementos de la lista (ver figura 57).

```
Operacion_D_E_C::~Operacion_D_E_C() {
    if (lista!=NULL) {
        pNodoD p = lista, q = lista->getAnte(), r = NULL;
        while (p!=q) {
```

```
            r = p;
            p = p->getSgte();
            delete(r);
        }
        delete(p);
    }
}
```

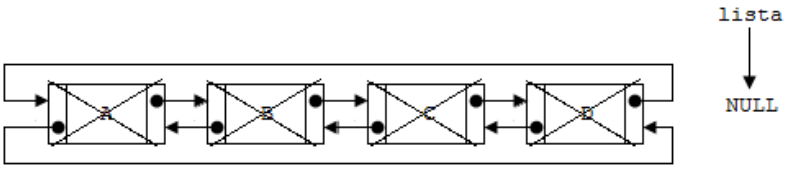


Figura 57. Elimina todos los nodos de una LDEC

3. Método mostrar elementos de la una lista

Este método permite mostrar toda la información de cada uno de los nodos de la lista.

```
void Operacion_D_E_C::mostrar() {
    if (lista!=NULL) {
        pNodoD p = lista;
        do {
            cout<<p->getInfo()<<"  ";
            p = p->getSgte();
        } while (p!=lista);
    }
}
```

4. Método para insertar un nodo al principio

Lo que se tiene que hacer es crear un nuevo nodo y verificar si existen nodos en la lista.

En el caso de que existan nodos en la lista debe obtener el último nodo en la cual actualiza los enlaces, es decir, el enlace anterior de lista apunta al nuevo nodo, el enlace siguiente del último nodo apunta al nuevo nodo, el enlace anterior del nuevo nodo apunta al último nodo y finalmente el enlace siguiente del nuevo nodo apunta a lista (ver figura 58 a)); caso contrario es el primer nodo de la lista por lo tanto los enlaces del nuevo nodo se apunta a sí mismo (ver figura 58 b)).

Cualquiera sea el caso lista apunta al nuevo nodo de la lista.

Y su implementación respectiva para este método es el siguiente

```
void Operacion_D_E_C::ins_delante(char x) {
    pNodoD p = new NodoD(x), q = NULL;
    if (lista!=NULL) {
        q = lista->getAnte();
        lista->setAnte(p);
```

```
        p->setSgte(lista);
        q->setSgte(p);
        p->setAnte(q);
    }
    else {
        p->setAnte(p);
        p->setSgte(p);
    }
    lista = p;
}
```

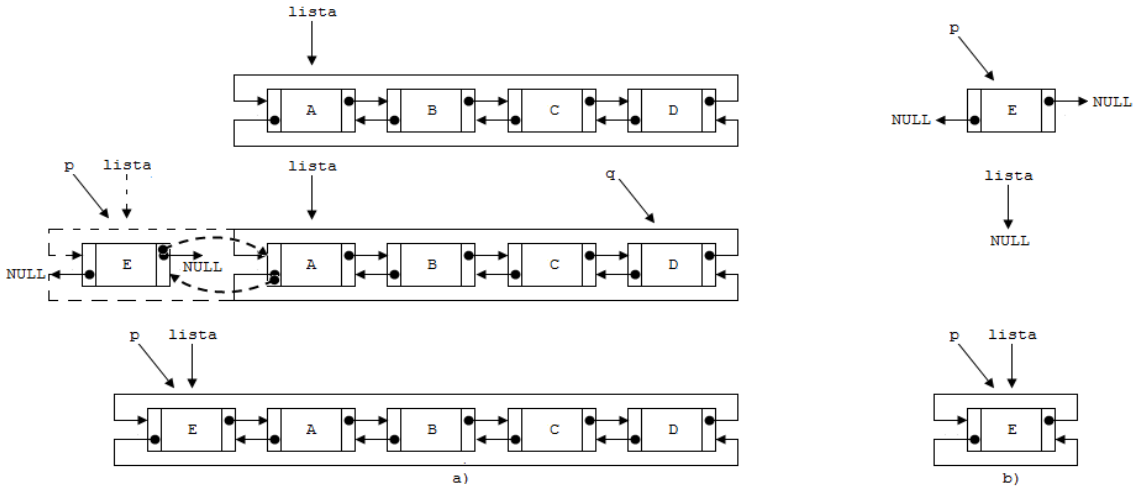


Figura 58. Insertar un nodo al principio en una LDEC

5. Método para eliminar el primer elemento de la lista

Antes de eliminar el primer nodo se debe verificar que existan nodos en la lista, de existir elementos en la lista se puede eliminar el nodo para eso se debe ubicar el primero y último nodo de la lista.

Si es el último y único nodo la lista de quedar en nulo (ver figura 59 a)), caso contrario debe avanzar lista al siguiente nodo son las siguientes actualizaciones que son el enlace siguiente del primer nodo apunta de lista y el enlace anterior de lista apunta al último nodo (ver figura 59 b)).

Su código correspondiente al método es lo siguiente

```
void Operacion_D_E_C::eli_delante() {
    if (lista!=NULL) {
        pNodoD p = lista, q = lista->getAnte();
        if (p==q) lista = NULL;
        else {
            lista = lista->getSgte();
            lista->setAnte(q);
            q->setSgte(lista);
        }
        delete(p);
    }
}
```

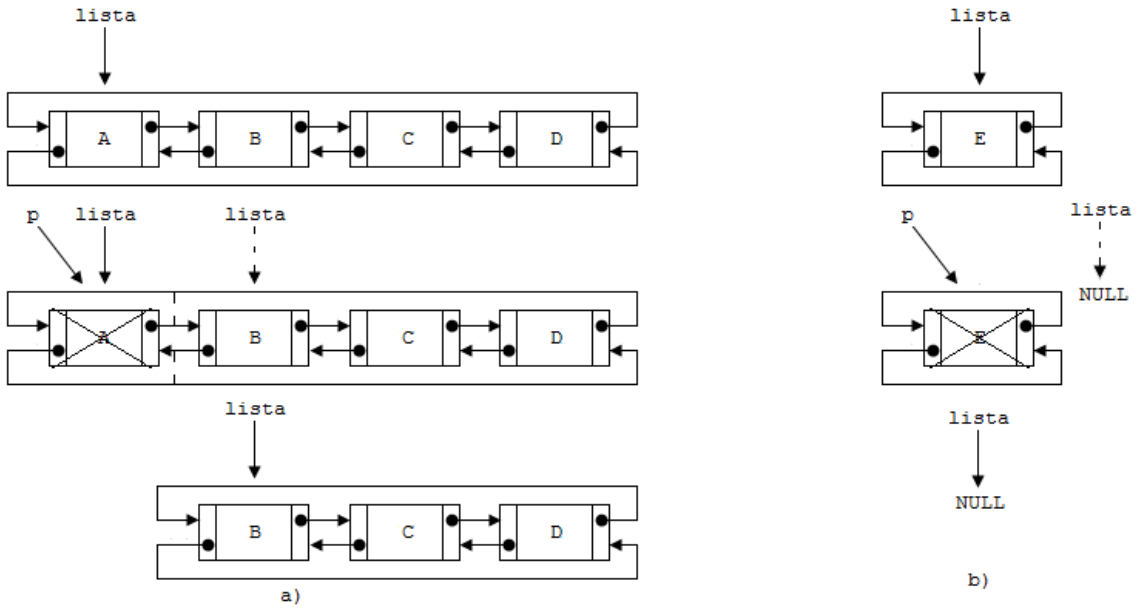


Figura 59. Eliminar el primer nodo de una LDEC

3.7. Ejercicios

Para cada una de los tipos de listas enlazadas, implemente los siguientes ejercicios en sus clases correspondientes.

1. Realizar un método para buscar un determinado elemento. En el caso de que exista en la lista devuelva la dirección de memoria del nodo, caso contrario devuelva NUL.
2. Realizar un método para contar las veces que se repite un determinado nodo.
3. Realizar un método para contar el nodo que más veces se repite en la lista.
4. Realizar un método para obtener el carácter promedio de la lista.
5. Realizar un método para insertar elementos en forma ordenada.
6. Realizar un método para eliminar todos los elementos iguales a un determinado elemento, excepto el primero que se encuentre.
7. Realizar un método para eliminar elementos repetidos no consecutivos, excepto el primero que se repita.
8. Realizar un método para intercalar dos listas.
9. Realizar un método para eliminar elementos repetidos consecutivos, excepto el primero que se repita.
10. Suponiendo que la información que almacena los nodos de la lista expresiones matemáticas. Realizar un método para verificar si los paréntesis están balanceados. Ej. ((x-25)/5+5*x)/(5-x), es balanceada.

11. Suponiendo que la información que almacena los nodos de la lista son solamente caracteres numéricos. Realizar un método para calcular el resultado de la expresión en forma comercial. Ej. 0-25/5+5*0/5-0; resultado -1.