
PEL

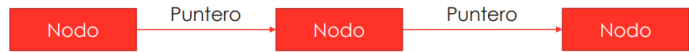


Resumen UF4

U1

Estructura de datos dinámica: Las estructuras de datos dinámicas crecen y se contraen durante la ejecución del programa.

Listas enlazadas: Son colecciones de elementos, dispuestos uno a continuación del otro cada uno de ellos conectado al siguiente mediante un “enlace” o “referencia”. La idea básica consiste en construir una lista de nodos. Estos se componen de dos partes:



- **Campos:** Contiene la información y es un valor de un tipo genérico.
- **Enlaces:** Apunta al siguiente nodo de la lista.

La representación más extendida, es utilizar una caja de dos secciones, en la



que en la primera sección se encuentra el valor del dato, y en la segunda, el puntero que apunta al siguiente nodo. Los enlaces se representan por flechas, indicando que tiene la dirección en memoria del siguiente nodo, situando así los nodos en una secuencia.

El último nodo, debe representarse de forma diferente, para indicar que este no se enlaza con ningún otro. Se puede emplear para ello las siguientes representaciones gráficas:



Clasificación de las listas enlazadas

- **Listas simplemente enlazadas:** Cada nodo contiene un único enlace que se conecta con el nodo sucesor. La lista es eficiente en recorridos directos.
- **Listas doblemente enlazadas:** Cada nodo contiene dos enlaces, uno al predecesor y otro al sucesor. La lista es eficiente tanto en recorridos directos como en recorridos inversos.
- **Listas circulares simplemente enlazadas:** Una lista simplemente enlazada en la que el último elemento se enlaza al primer elemento, de tal modo que la lista puede ser recorrida en modo circular.
- **Listas circulares doblemente enlazadas:** Una lista doblemente enlazada, en la que el último elemento se enlaza al primero y viceversa. Se puede recorrer en modo circular tanto en dirección directa como en inversa.

TAD Lista

```
template <class T> class NodoGenerico
{
protected:
    T dato;
    NodoGenerico <T>* enlace;
public:
    NodoGenerico (T t)
    {
        dato = t;
        enlace = 0;
    }
    NodoGenerico (T p, NodoGenerico<T>* n)
    {
        dato = p;
        enlace = n;
    }
    T datoNodo() const
    {
        return dato;
    }
    NodoGenerico<T>* enlaceNodo() const
    {
        return enlace;
    }
    void ponerEnlace(NodoGenerico<T>* sgte)
    {
        enlace = sgte;
    }
};
```

```
typedef int Dato;
#include "Nodo.h"
class Lista
{
protected:
    Nodo* primero;
public:
    Lista()
    {
        primero = NULL;
    }
    void crearLista();
    //...

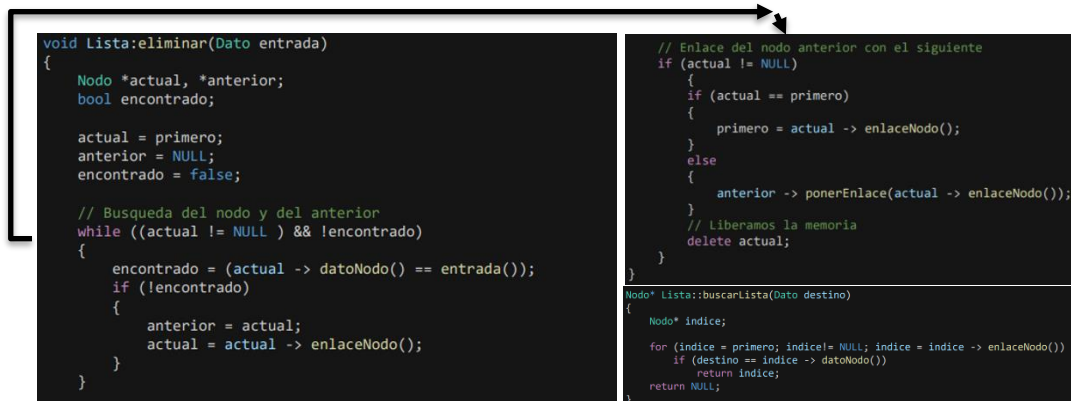
    void Lista::crearLista()
    {
        int x;
        primero = 0;
        cout << "Termina con -1" << endl;
        do {
            cin >> x;
            if (x != -1)
            {
                primero = new Nodo(x, primero);
            }
        } while (x != -1);
    }
```

```
void Lista::insertarCabezaLista(Dato entrada)
{
    Nodo* nuevo ;
    nuevo = new Nodo(entrada);
    nuevo -> ponerEnlace(primeros); // enlaza nuevo con primero
    primero = nuevo; // mueve primero y apunta al nuevo nodo
}

void Lista::insertarUltimo(Dato entrada)
{
    Nodo* ultimo = this -> ultimo();
    ultimo -> ponerEnlace(new Nodo(entrada));
}

Nodo* Lista::ultimo()
{
    Nodo* p = primero;
    if (p == NULL) throw "Error, lista vacia";
    while (p -> enlaceNodo() != NULL) p = p -> enlaceNodo();
    return p;
}

void Lista::insertarLista(Nodo* anterior, Dato entrada)
{
    Nodo* nuevo;
    nuevo = new Nodo(entrada);
    nuevo -> ponerEnlace(anterior -> enlaceNodo());
    anterior -> ponerEnlace(nuevo);
}
```



```

void Lista:eliminar(Dato entrada)
{
    Nodo *actual, *anterior;
    bool encontrado;

    actual = primero;
    anterior = NULL;
    encontrado = false;

    // Búsqueda del nodo y del anterior
    while ((actual != NULL) && !encontrado)
    {
        encontrado = (actual -> datoNodo() == entrada());
        if (!encontrado)
        {
            anterior = actual;
            actual = actual -> enlaceNodo();
        }
    }

    // Enlace del nodo anterior con el siguiente
    if (actual != NULL)
    {
        if (actual == primero)
        {
            primero = actual -> enlaceNodo();
        }
        else
        {
            anterior -> ponerEnlace(actual -> enlaceNodo());
        }
        // Liberamos la memoria
        delete actual;
    }
}

Nodo* Lista:buscarLista(Dato destino)
{
    Nodo* indice;
    for (indice = primero; indice != NULL; indice = indice -> enlaceNodo())
    {
        if (destino == indice -> datoNodo())
            return indice;
    }
    return NULL;
}
    
```

Pila: es una estructura de datos que almacena elementos, siguiendo un orden estricto. Estas se denominan también estructuras **LIFO** (Last-in, First-out). Las pilas solo pueden accederse a sus datos por un único extremo, la **cima** de la pila. Una pila está ordenada cuando hay un elemento al que se puede acceder primero, otro al que se puede acceder en segundo lugar, y así sucesivamente. Las entradas de la pila se deben eliminar en orden inverso al que se situaron. Esto se debe a su propiedad de **LIFO**.

Las pilas pueden implementarse almacenando los elementos en un array, presentando así una longitud fija. Otra forma de implementarlas es mediante listas enlazadas, en donde cada elemento de la lista forma un nodo en la lista.

- **UnderFlow:** En caso de intentar extraer un dato de una pila vacía, se produce una excepción llamada desbordamiento negativo o **underflow**.
- **Overflow:** Si se intenta agregar a un elemento a una pila llena, se produce un desbordamiento u **overflow**.

Esto se puede evitar mediante funciones que comprueban si la pila está llena o vacía.

TAD Pila

El configurar una pila dinámica mediante una lista enlazada, se realiza almacenando cada elemento de la pila como nodo de la lista. Dado que las operaciones de insertar y extraer se realizan por el mismo extremo, las acciones correspondientes mediante la lista se realizarán de la misma forma.

En pila implementada por arrays es estática, dado a que el array es de tamaño fijo. Es por eso que es necesario controlar el tamaño de la pila. La forma habitual de hacer esto, es definir una pila vacía, y a continuación ir introduciendo elementos de modo que el primer elemento se introduce en la posición 0, el segundo en la posición 1, el siguiente a ese en la 2, y así sucesivamente.

Insertar:

1. Verificar si la pila no está llena.
2. Incrementar en 1 el apuntador a la cima de la pila.
3. Almacenar el elemento en la posición del apuntador de la pila.

Quitar:

1. Verificar si la pila no está vacía.
2. Leer el elemento de la posición del apuntador de la pila.
3. Decrementar en 1 el apuntador de la pila.

El configurar una pila dinámica mediante una lista enlazada, se realiza almacenando cada elemento de la pila como nodo de la lista. Dado que las operaciones de insertar y extraer se realizan por el mismo extremo, las acciones correspondientes mediante la lista se realizarán de

la misma forma. La estructura que tiene la pila implementada mediante listas enlazadas es muy similar a las propias listas. Los elementos de la pila serán los nodos de la lista, y las operaciones del tipo pila implementada con listas son las mismas que la pila implementada con arrays.

```
#include <iostream>
template <class T>
class GenericStack
{
    class Node
    {
    public:
        Node *next;
        T element;
        Node(T node)
        {
            element = node;
            next = nullptr;
        };
    };

    Node *top;

public:
    GenericStack() { top = nullptr; }
    bool const isEmpty() { return top == nullptr; }
    void Push(T element)
    {
        Node *node;
        node = new Node(element);
        node->next = top;
        top = node;
    }
    T Pop()
    {
        if (isEmpty())
            throw "Empty stack, cant remove.";
        T aux = top->element;
        top = top->next;
        return aux;
    }

    void Clear()
    {
        Node *node;
        while (!isEmpty())
        {
            node = top;
            top = top->next;
            delete node;
        }
    }

    T Top()
    {
        if (isEmpty())
            throw "Empty stack.";
        return top->element;
    }

    void Reverse()
    {
        Node *prev, *curr, *succ;
        curr = prev = top;
        curr = curr->next;
        prev->next = nullptr;
        while (curr != nullptr)
        {
            succ = curr->next;
            curr->next = prev;
            prev = curr;
            curr = succ;
        }
        top = prev;
    }

    int Size()
    {
        int count = 0;
        Node *current = top;
        while (current != nullptr)
        {
            count++;
            current = current->next;
        }
        return count;
    }

    void Print()
    {
        if (isEmpty())
            return;
        T p = Top();
        Pop();
        Print();
        std::cout << p << " ";
    }

    ~GenericStack() { Clear(); }
}
```

Cola: Es una estructura de datos que almacena elementos, siguiendo un orden estricto. Estas se denominan también estructuras **FIFO** (First-in, First-out). Tienen numerosas aplicaciones en el mundo real, como colas de mensajes, colas de tareas, colas de prioridades, etc.

Una cola es una estructura de datos que almacena los elementos, de forma que el acceso a dichos datos se hace por uno de los extremos de la lista. En las colas, los elementos se insertan en la parte final de la lista, y se extrae por la parte inicial:

- **Insertar:** Añade un elemento al final de la cola.
- **Quitar:** Elimina o extrae un elemento del inicio de la lista.

TAD Cola

La implementación del TAD cola con lista enlazada, emplea dos punteros de acceso a la lista: frente y final. Estos dos punteros son los extremos por donde se incluyen y extraen los elementos de la cola. La implementación de una cola genérica se realiza con dos clases, la clase **nodoCola** y la clase **colaGenerica**. Esta clase **colaGenerica**, define las variables de acceso frente y final, y las correspondientes operaciones de un TAD cola.

```

#include <iostream>
template <class T>
class GenericQueue
{
    class Node
    {
    public:
        Node *next;
        T element;
        Node(T t)
        {
            element = t;
            next = nullptr;
        }
    };
    Node *start;
    Node *end;

public:
    GenericQueue()
    {
        start = end = nullptr;
        std::cout << "creating a queue..." << std::endl;
    }
    bool isEmpty() const { return start == nullptr; };
    void Push(T element)
    {
        Node *node;
        node = new Node(element);
        std::cout << "Pushing :" << node->element << std::endl;
        if (isEmpty())
            start = node;
        else
            end->next = node;
        end = node;
    }
    T Pop()
    {
        if (isEmpty())
            throw "Empty queue, cant remove.";
        T aux = start->element;
        std::cout << "Popping :" << aux << std::endl;
        Node *node = start;
        start = start->next;
        delete node;
        return aux;
    }

    void Clear()
    {
        while (start != nullptr)
        {
            Node *node;
            node = start;
            start = start->next;
            delete node;
        }
        end = nullptr;
    }

    T GetStart() const
    {
        if (isEmpty())
            throw "Empty queue";
        return start->element;
    }

    bool isEqual(GenericQueue<T> &q2)
    {
        bool equal = true;
        if (Size() != q2.Size())
            equal = false;

        Node *current1 = nullptr;
        Node *current2 = nullptr;

        current1 = start;
        current2 = q2.start;
        while (current1 != nullptr && current2 != nullptr && equal)
        {
            if (current1->element != current2->element)
                equal = false;
            current1 = current1->next;
            current2 = current2->next;
        }
        delete current1;
        delete current2;
        return equal;
    }

    int Size()
    {
        int count = 0;
        Node *current = start;
        while (current != nullptr)
        {
            count++;
            current = current->next;
        }
        return count;
    }

    ~GenericQueue() { Clear(); }
}

```

EXTRA:: Mi implementacion del TAD lista (int)

```

#include <iostream>
// template <class T>
class Node
{
protected:
    int data;
    Node *joint;
public:
    Node(int t)
    {
        data = t;
        joint = 0;
    };
    Node(int p, Node *n)
    {
        data = p;
        joint = n;
    };
    int NodeData() const { return this->data; };
    Node *NodeJoint() const { return this->joint; };
    void SetJoint(Node *sgte) { joint = sgte; };
};

class List
{
protected:
    Node* first;
public:
    List(){first = 0;};
    void CreateList(){
        int x;
        first = 0;
        do{
            std::cin >> x;
            if(x!=-1){
                first = new Node(x,first);
            }while(x!=-1);
        }
    }

    void InsertInHead(int entry)
    {
        Node* node;
        node= new Node(entry);
        node ->SetJoint(first);
        first = node;
    };
}

```

```
Node* Last(){
    Node* node = first;
    if(node == 0) throw "error, empty list";
    while(node->NodeJoint() != 0) node = node->NodeJoint();
    return node;
};
```

```
void InsertInTail(int entry){
    Node* node = this->Last();
    node ->SetJoint(new Node(entry));
};
```

```
void InsertBetween(Node* last, int entry){
    Node* node;
    node = new Node(entry);
    node ->SetJoint(last->NodeJoint());
    last->SetJoint(node);
}
```

EXTRA METHODS

```
bool IsPalindrome(string phrase)
{
    bool isPal = true;
    GenericStack<char> *stack = new GenericStack<char>();
    for (int i = 0; i < phrase.size(); i++)
        stack->Push(phrase[i]);

    for (int i = 0; i < phrase.size(); i++)
    {
        char last = stack->Top();
        if (phrase[i] == last)
            stack->Pop();
        else
            isPal = false;
    }
    delete stack;
    return isPal;
}
```

```
template <class T>
void CopyStack(GenericStack<T> *s1, GenericStack<T> *s2)
{
    int size = s1->Size();
    s2->Clear();
    for (int i = 0; i < size; i++)
        s2->Push(s1->Pop());
    s2->Reverse();
    s2->Print();
}
```

```
template <class T>
GenericQueue<T> *ExactCopy(GenericQueue<T> *queue)
{
    GenericQueue<T> *cloned = new GenericQueue<T>();
    while (!queue->isEmpty())
    {
        T data = queue->GetStart();
        cloned->Push(data);
        queue->Pop();
    }
    return cloned;
}
```